Web data formats in Lazarus/FPC

Michaël Van Canneyt

June 30, 2018

Abstract

These days it is hard to avoid contact with popular web data formats as XML and - more recently - JSON. Fortunately, Lazarus and FPC are equipped with a set of tools to handle and output these formats. An overview.

1 Introduction

In an age where words as SOA, Webservices and SOAP and AJAX are considered to be standard tools of every programmer, it is hard to avoid the building blocks on which these concepts are built. The data format XML is a corner stone on which these technologies are built: a textual data encoding and structuring format. It’s highly standardized by the W3 consortium, which has made it the format of choice for transferring data. FPC/Lazarus comes with a set of tools that implement some of the W3 consortium’s standards and recommendations: DOM, XPath.

As XML standards become more and more elaborate - and hence cumbersome - JSON (JavaScript Object Notation) emerged as a second standard for data format: it is suitable in the first place for transferring data to a web-browser, where the javascript can be readily parsed and used as a native way of reading the data: the JavaScript engine itself can be used to parse the data and transform it into Javascript Objects, Arrays and associative arrays.

While the JSON notation for data is not really useful outside a browser, a web server application should be able to produce it, so the browser can read it. Other than the format’s (extremely simple) specifications, there are little standards to be implemented, so the FPC implementation of JSON was created so the data that needs to be sent is easy to create and handle.

In this article the implementations available in FPC/Lazarus for handling XML and JSON will be discussed; The XML and JSON formats themselves will not be discussed: just the way these formats can be treated.

2 XML: The DOM way

The W3 consortium has made a specification to represent a XML document in memory: the DOM (Document Object Model)[1]. Free Pascal contains a DOM Level 1 implementation (with some DOM level 2 extensions) in the dom unit. To work with the DOM is easy, but requires the document to be completely in memory at all times.

The DOM represents an XML document as a tree of nodes (of class TDOMNode): All parts of the document are described using a TDOMNode descendent. Which part this is, can be seen from the NodeType property of the TDOMNode class: each descendent has its own
Table 1: The main types and their nodetypes

<table>
<thead>
<tr>
<th>Class</th>
<th>NodeType</th>
<th>Data it represents</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDOMElement</td>
<td>ELEMENT_NODE</td>
<td>an XML tag.</td>
</tr>
<tr>
<td>TDOMAttr</td>
<td>ATTRIBUTE_NODE</td>
<td>an attribute of an XML tag.</td>
</tr>
<tr>
<td>TDOMText</td>
<td>TEXT_NODE</td>
<td>the text enclosed in XML tags.</td>
</tr>
<tr>
<td>TDOMComment</td>
<td>TEXT_NODE</td>
<td>a XML comment.</td>
</tr>
<tr>
<td>TDOMCDATASection</td>
<td>CDATA_SECTION_NODE</td>
<td>a CDATA section.</td>
</tr>
</tbody>
</table>

value of NodeType. The main TDOMNode descendents with their NodeTypes can be found in table[1]

Each node can have a number of child nodes, and the tree of nodes can be traversed completely with the following function methods of TDOMNode:

**FirstChild** returns the first child of the TDOMNode.

**LastChild** returns the last child of the TDOMNode.

**NextSibling** returns the next node at the same level of the current TDOMNode.

**PreviousSibling** returns the previous node at the same level of the current TDOMNode.

The complete XML document is represented with the TDOMDocument class. It contains the DocumentElement property, which is of type TDOMElement, and contains the instance of the first element in the XML document. The TDOMNode class contains a OwnerDocument property which points to the TDOMDocument instance of which it is a part. The TXMLODocument descendent has some additional properties to describe the encoding, style sheet etc. In the subsequent, the TXMLODocument will be used, but most of the methods are actually defined in TDOMDocument.

Since the TXMLODocument instance is the owner of all the nodes, it is also responsible for creating the nodes. It therefore has the following methods to create nodes:

function CreateElement(const tagName: DOMString): TDOMElement;

function CreateTextNode(const data: DOMString): TDOMText;

function CreateComment(const data: DOMString): TDOMComment;

function CreateCDATASection(const data: DOMString): TDOMCDATASection;

function CreateAttribute(const name: DOMString): TDOMAttr;

After creating a node, it can be inserted in the document tree with one of the following methods of TDOMNode:

function InsertBefore(NewChild, RefChild: TDOMNode): TDOMNode;

function ReplaceChild(NewChild, OldChild: TDOMNode): TDOMNode;

function RemoveChild(OldChild: TDOMNode): TDOMNode;

function AppendChild(NewChild: TDOMNode): TDOMNode;

Armed with the above definitions a simple application can be made to view and edit an XML document. It’s only logical to represent an XML document as a tree structure, with elements and texts as the nodes in the tree.

### 3 A simple XML viewer

The XML viewer application consists therefore of a single form with a menu that contains the usual file menu items (new, open, save and 'save as'), and a toolbar with the same
actions as the menu. The rest of the form is filled with a TTreeView instance (TVXML) which will show the XML document, and a TPanel instance (PDetails) which will be used to display details of the current node.

To fill the treeview with the contents of a TXMLDocument, the following code is used:

```delphi
procedure TMainForm.ShowDocument(AXML: TXMLDocument);
begin
  With TVXML.Items do
  begin
    BeginUpdate;
    Try
      Clear;
      If Assigned(AXML.DocumentElement) then
      begin
        ShowNode(Nil,AXML.DocumentElement);
        TVXML.Selected:=TVXML.Items.GetFirstNode;
      end;
    Finally
      EndUpdate;
    end;
  end;
end;
```

The code is quite simple: it clears the treeview, and then lets the `ShowNode` method do the actual work, starting from the `DocumentElement` node. The `ShowNode` is a simple recursive procedure:

```delphi
procedure TMainForm.ShowNode(AParent: TTreeNode; E: TDomNode);
Var
  N : TTreeNode;
  D : TDomNode;
begin
  N:=TVXML.Items.AddChild(AParent,E.NodeName);
  N.Data:=E;
  D:=E.FirstChild;
  While (D<>Nil) do
  begin
    Case D.NodeType of
      ELEMENT_NODE : ShowNode(N,D);
      TEXT_NODE    : ShowNode(N,D);
    end;
  D:=D.NextSibling;
end;
```

The first thing that is done is creating a node in the treeview to show the TDOMNode. The `nodeName` property is used as the text for the node. For a TDOMElement instance, this is the tag name. For a TDOMText instance, this is the fixed text #text. After this, the children of the node are traversed using `FirstChild` and `NextSibling`, and they added to the newly created node, by calling `ShowNode`. Note that `ShowNode` is only called for child nodes of type ELEMENT_NODE and TEXT_NODE. This means that attributes are not shown in the tree.
The above 2 short procedures are enough to show the XML document in a TreeView. To display the contents of a node, the panel is used. When a node is selected in the tree, the OnSelectionChanged event is fired:

```pascal
procedure TMainForm.TVXMLSelectionChanged(Sender: TObject);
begin
  ClearNodeData;
  If Assigned(TVXML.Selected) and Assigned(TVXML.Selected.Data) then
    ShowNodeData(TDomNode(TVXML.Selected.Data))
  else
    PDetails.Caption:=SSelectNode;
end;
```

The ClearNodeData clears the contents PDetails panel by deleting all controls in it:

```pascal
procedure TMainForm.ClearNodeData;
begin
  With PDetails do
    While (ControlCount>0) do
      Controls[0].Free;
end;
```

The controls in the PDetails panel are created by the code in the ShowNodeData call, where the path to the current node is also constructed and displayed in the status bar:

```pascal
procedure TMainForm.ShowNodeData(N : TDomNode);
Var
  P : TDomNode;
  S : String;
begin
  PDetails.Caption:='';
P:=N;
S:='';
While (P<>Nil) and Not (P is TXMLDocument) do
begin
  If S<>'' then
    S:='/'+S;
  S:=P.NodeName+S;
P:=P.ParentNode;
end;
SBXML.SimpleText:=Format(SCurrentNode,[S]);
Case N.NodeType of
  TEXT_NODE : ShowTextData(N as TDomText);
  ELEMENT_NODE : ShowElementData(N as TDomElement);
end;
end;
```

As can be seen, for a text node, the real work happens in the ShowTextData call:

```pascal
procedure TMainForm.ShowTextData(N : TDomNode);
```
Var
  M : TMemo;
begin
  DataTopLabel(SNodeText);
  M:=TMemo.Create(Self);
  M.Parent:=PDetails;
  M.Lines.Text:=N.NodeValue;
  M.Align:=alClient;
end;

The DataTopLabel shows a label aligned to the top of the PDetail panel. The rest of
the code shows a TMemo in the rest of the panel, and sets its text to the NodeValue
property of the TDOMNode. For a text node, this is the actual text of the node.

The ShowElementData call used to show a TDOMElement is slightly more compli-
cated, as it has to show all attributes of the element:

procedure TMainForm.ShowElementData(E : TDomElement);
Var
  L : TLabel;
  G : TStringGrid;
  I : Integer;
  N : TDomNode;

begin
  DataTopLabel(Format(SNodeData,[E.NodeName]));
  G:=TStringGrid.Create(Self);
  G.Parent:=PDetails;
  G.Align:=alClient;
  G.RowCount:=2;
  G.ColCount:=2;
  G.Cells[0,0]:=SAttrName;
  G.ColWidths[0]:=120;
  G.Cells[1,0]:=SAttrValue;
  G.RowCount:=1+E.Attributes.Length;
  G.Options:=G.Options+[goColSizing];
  If (G.RowCount>0) then
    G.FixedRows:=1
  else
    G.FixedRows:=0;
  For I:=1 to E.Attributes.Length do
    begin
      N:=E.Attributes[i-1];
      G.Cells[0,I]:=N.NodeName;
      G.Cells[1,I]:=N.NodeValue;
    end;
end;

As can be seen, it also shows a label at the top of the panel, and in the rest of the panel, a
stringgrid with 2 columns is created, and filled with the attributes of the element.

Now that the data in a XML document can be shown, all that needs to be done is to be able
to read one from disk. The DOM specification doesn’t specify how to read an XML docu-
The FPC implementation to read a XML document is in a separate unit, XMLRead. To read a complete XML document, it contains a single overloaded call:

```pascal
procedure ReadXMLFile(out ADoc: TXMLDocument;
    const AFilename: String);
procedure ReadXMLFile(out ADoc: TXMLDocument;
    var f: TStream);
```

As can be seen, a XML document can be read from stream, file, or pascal text file. The ADoc parameter should be Nil on entry. When the document is read completely, the parameter will contain a TXMLDocument instance. The programmer is responsible for freeing this instance when he is done with it. In case an error is encountered during the parsing of the document, an exception will be raised, and no document is returned.

In case not a complete XML document must be read, but just a part of it, the ReadXMLFragment must be used:

```pascal
procedure ReadXMLFragment(AParentNode: TDOMNode;
    const AFilename: String);
procedure ReadXMLFragment(AParentNode: TDOMNode;
    var f: TStream);
```

These procedures will read a XML fragment - it must be one or more complete elements and all elements below it - and will insert it as children of AParentNode.

Armed with these methods, the code to read a file from disk is rather simple:

```pascal
procedure TMainForm.OpenFile(AFileName: String);
Var
    ADoc : TXMLDocument;
begin
    ReadXMLFile(ADoc,AFileName);
    If Assigned(FXML) then
        FreeAndNil(FXML);
    FXML:=ADoc;
    SetFileName(AFileName);
    ShowDocument(FXML);
end;
```

If the call to ReadXMLFile was succesful, then the current XML document (stored in FXML) is freed, and then replaced with the newly read document. Finally, the ShowDocument call is used to show the document in the tree. The SetFileName call stores the filename, and shows it in the form title bar.

Similar to reading, a TXMLDocument can be written to file using the procedures in the XMLWrite unit:

```pascal
procedure WriteXMLFile(doc: TXMLDocument;
    const AFile: Text);
procedure WriteXMLFile(doc: TXMLDocument;
    var AFile: Text);
```

The code in the XML explorer is then a simple 2 lines:
procedure TMainForm.SaveToFile(AFileName : String);
begin
  WriteXMLFile(FXML,AFileName);
  SetFileName(AFileName);
end;

This is all that is needed to make the XML viewer application functional. When running, it should look like figure 1 on page 7.

To demonstrate the manipulation capabilities of the DOM model, some code will be added to add text nodes or element nodes to the DOM Tree. To this end, a new menu item and toolbar button is created, which will create a new element relative to the current element. The code executed by the menu item is as follows:

procedure TMainForm.ANewElementExecute(Sender: TObject);

Var
  P,N : TDomNode;
  PT : TTreeNode;

begin
  P:=Nil;
  PT:=TVXML.Selected;
  If Assigned(PT) then
    P:=TDOMNode(PT.Data)
  else
    P:=FXML;

  N:=TDOMNode.Create;
  N.Name:="\";
  N.NodeType:=nodeElement;
  N.Parent:=P;
  P:=N;
end;
N:=NewElement(P,PT);
If (N<>Nil) then
begin
  ClearNodeData;
  ShowNodeData(N);
end;
end;

At first, the current dom node is determined. If none is found (e.g. when the document is empty), then the document itself is set as the current dom node (TXMLDocument is also a descendent of TDOMNode). The NewElement call does the actual work. If it returns a node, then the node data is shown.

The real work happens in the NewElement function:

Function TMainForm.NewElement(P : TDomNode;
  PT : TTreeNode) : TDomNode;

Var
  N,NN : TDomNode;
  NT : integer;
  NL : TNodeLocation;
  S : String;
  TN : TTreeNode;

begin
  Result:=Nil;
  With TNewNodeForm.Create(Self) do
  try
    HaveParent:=P.ParentNode<>Nil;
    If (ShowModal<>mrOK) then
      Exit;
    NT:=NodeType;
    NL:=NodeLocation;
    S:=NodeText;
    finally
      Free;
    end;
  end;

This first part only collects data: it shows a dialog (TNewNodeForm, its details are not relevant to the discussion) which asks the type of node that should be created, where it should be created, and what text it should contain. With this data, the actual work is done.

At first a node is created. The TXMLDocument calls are used for this:

Case NT of
  ELEMENT_NODE : N:=FXML.CreateElement(S);
  TEXT_NODE : begin
    N:=FXML.CreateTextNode(S);
    NL:=nlLastChild;
  end;
end;
If (P.NodeType=TEXT_NODE) then
nl:=nlReplaceCurrent;

When the node is created, the nodelocation is used to determine where in the DOM tree the new node should be inserted. Not all combinations are possible: a text node is always
a leaf node (i.e. it must always be the only child node from its parent). Some checks are done for this.

After this, the node is inserted in the DOM tree, and at the same time the treeview is updated:

```pascal
Case NL of
  nlFirstChild :
    begin
      NN:=P.FirstChild;
      If (NN<>Nil) then
        P.InsertBefore(N,NN)
      else
        P.AppendChild(N);
      TN:=TVXML.Items.AddChildFirst(PT,N.NodeName);
    end;
  nlLastChild :
    begin
      P.AppendChild(N);
      TN:=TVXML.Items.AddChild(PT,N.NodeName);
    end;
  nlBeforeCurrent :
    begin
      P.ParentNode.InsertBefore(N,P);
      TN:=TVXML.Items.Insert(PT,N.NodeName);
    end;
  nlReplaceCurrent :
    begin
      P.ParentNode.ReplaceChild(N,P);
      PT.Text:=N.NodeName;
      TN:=PT;
    end;
end;
TN.Data:=N;
TVXML.Selected:=TN;
Result:=N;
end;
```

Code to modify the properties of a node (text or attributes) is left up to the interested reader.

To navigate and search for nodes through the XML document tree, the W3 created the XPath specification[2]. The XPath unit that comes with Free Pascal contains an implementation of this specification. The main function of this unit is the `EvaluateXPathExpression` call:

```pascal
function EvaluateXPathExpression(
  const AExpressionString: DOMString;
  AContextNode: TDOMNode): TXPathVariable;
```

This function will evaluate an XPath expression, starting at AContextNode, and returns the result as a TXPathVariable instance. The TXPathVariable class is the base class for all XPath results, there are a lot of descendents of this class, each representing a possible result of the query (which can be a number, text, set of nodes etc.)

A complete discussion of the XPath is out of the scope of this article, but a small example will be given to show how it can be used to quickly locate a node in the tree: For this, a
'Go to node' menu item is made below the 'Edit' menu. When clicked, the following code is executed:

```pascal
procedure TMainForm.AGotoExecute(Sender: TObject);

Var
  S : String;
  V : TXPathVariable;
  N : TNodeSet;

begin
  If Assigned(FXML.DocumentElement) then
    S:='/'+FXML.DocumentElement.NodeName
  else
    S:='';
  If InputQuery(SGoto,SSpecifyPath,S) then
    begin
      V:=EvaluateXPathExpression(S,FXML);
      try
        If Not (V is TXPathNodeSetVariable) then
          ShowMessage(SErrNotaNode)
        else
          begin
            N:=V.AsNodeSet;
            If (N.Count<>1) then
              ShowMessage(SErrSingleNode)
            else
              GotoNode(TDOMNode(N.Items[0]));
          end;
      finally
        V.Free;
      end;
    end;
end;
```

First, the root location for the search operation is determined (this could be extended for instance to search starting at the current node). This value is used to query the user for the path to look for. The path is then passed on to the `EvaluateXPathExpression` call, together with the XML document as the root of the search operation. If the result is of type `TXPathNodeSetVariable`, then the evaluation of the XPath expression resulted in one or more DOM nodes. The set of result nodes is returned as the `AsNodeSet` property, which is a `TList`. If the list contains a single node, then it is passed to the `GotoNode` call, which will display the node in the tree.

The XML viewer can be used to inspect for instance the Lazarus project information file (.lpi) which is in XML format. The following path:

```xml
/CONFIG/ProjectOptions/Units/Unit1/Filename
```

Will expand the node that contains the filename of the first unit in the project. The XPATH expression

```xml
/CONFIG/ProjectOptions/Units/*/Filename
```

Would result in a node set with the names of all units in the project. Since only a single node can be displayed in the tree, this would result in an error message.
4 JSON: the data structures

The JSON (JavaScript Object Notation) specification [3] is much more simple than the XML specifications: it fits on a single sheet of paper. The specification is geared towards use in browsers: indeed, JSON was conceived to quickly send structured data to a browser, without the need for complicated DOM structures and parsers. Instead, the built-in JavaScript engine is used to convert the JSON data to a JavaScript object, ready to be used in the browser. This simplicity and speed probably accounts for its quick adoption by web-application programmers - especially in non-corporate settings.

The following is an example of JSON data:

```
{"addressbook": { "name": "Mary Lebow", 
  "address": { "street": "5 Main Street", 
    "city": "San Diego, CA", 
    "zip": 91912 }, 
  "phoneNumbers": ["619 332-3452","664 223-4667"] }
}
```

Which would in XML be represented for instance as

```
<addressbook>
  <name>Mary Lebow</name>
  <address><street>5 Main Street</street>
    <city zip="91912">San Diego, CA</city>
  </address>
  <phoneNumbers><phone>619 332-3452</phone>
    <phone>664 223-4667</phone>
  </phoneNumbers>
</addressbook>
```

(example taken from [4]).

The JSON specification only describes the textual data format, it does not give any specifications about how to handle the data. Since it is JavaScript, there is no need, as the data translates directly to JavaScript objects and arrays. For other languages - such as Object Pascal - there is no specification. The implementation provided with Free Pascal is therefore not the implementation of a standard, but rather an implementation which is believed to be easy for the quick creation and examination of data. The fpjson unit contains the JSON implementation for Free Pascal.

The basic object structure for the JSON implementation is shown in figure 2 on page 12. The basic class is TJSONData. It has the following properties:

- **JSONType** an enumerated which contains the data type for this object.
- **IsNull** True if the value is NULL.
- **AsBoolean** The value of the data as a boolean.
- **AsInteger** The value of the data as an integer.
- **AsFloat** The value of the data as a double or extended.
- **AsString** The value of the data as a string.
- **Count** The number of items contained in this data item. This will obviously be only nonzero for Objects and Arrays.
**Items** This array property gives access to the items contained in this data item. Each item is a TJSONData descendent instance. Note that the items can be of different types: an array can contain a mix of integers, strings, null values.

**AsJSON** returns the contents of the value (and its sub-values, if it is an array or object) as a JSON string.

The descendents of this class - one per JSON data type, as can be seen in figure 2 on page 12 - store the actual data. Their constructor takes the data value as a parameter.

To create a JSON string value, one would execute the following code:

```pascal
J:=TJSONString.Create('A string');
```

The constructors of the TJSONObject and TJSONArray optionally take an array of const, and will fill the object or array with the values in the array

```pascal
Ja:=TJSONArray.Create([Nil, True, 1, 2.3, 'String']);
Jo:=TJSONObject.Create(['a',1,'b',ja]);
```

Note that the TJSONObject constructor needs its arguments in pairs: A name (always a string) and a value (almost any valid JSON base type). From the example, it’s also visible that the array can also contain other TJSONData instances.

The array type TJSONArray can also be manipulated with simple add calls:

```pascal
function Add(Item : TJSONData): Integer;
function Add(I : Integer): Integer;
function Add(S : String): Integer;
function Add: Integer;
function Add(F : TJSONFloat): Integer;
function Add(B : Boolean): Integer;
function Add(AnArray : TJSONArray): Integer;
```
function Add(AnObject: TJSONObject): Integer;
Procedure Delete(Index : Integer);
Procedure Remove(Item : TJSONData);

The Add calls take care of creating the correct TJSONData descendent. The meaning of the Delete and Remove calls should be clear. Similar calls exist for the TJSONObject class, but they take an additional argument: the name of the element to add.

Armed with all these calls, the above JSON data can be formed as follows;

Procedure TestLong;

Var
    JAB, J, JA : TJSONObject;
    JT : TJSONArray;

begin
    JA:=TJSONObject.Create;
    JA.Add('street','5 Main Street');
    JA.Add('City','San Diego, CA');
    JA.Add('Zip',91912);
    JT:=TJSONArray.Create;
    JT.Add('619 332-3452');
    JT.Add('664 223-4667');
    J:=TJSONObject.Create;
    J.Add('name','Mary Lebow');
    J.Add('address',JA);
    J.Add('phonenumbers',JT);
    JAB:=TJSONObject.Create(
        ['addressbook',J]);
    Writeln(JAB.AsJSON);
    JAB.Free;
end;

Or, the shorthand way:

Procedure TestShort;

Var
    JAB, J, JA : TJSONObject;
    JT : TJSONArray;

begin
    JA:=TJSONObject.Create([‘street’,'5 Main Street',
                            ‘City’,'San Diego, CA’,
                            ‘Zip’,91912]);
    JT:=TJSONArray.Create([‘619 332-3452’,'664 223-4667’]);
    J:=TJSONObject.Create([‘name’,'Mary Lebow’,
                           ‘address’,JA,’phonenumbers’,JT]);
    JAB:=TJSONObject.Create([‘addressbook’,J]);
    Writeln(JAB.AsJSON);
    JAB.Free;
end;

It can be done even shorter without the auxiliary variables, but then the code becomes rather
unreadable. A quick check of the output will convince anyone that the output is the same in both cases, and equal to the JSON sample presented above.

Note that the last statement:

```
JAB.Free
```

Frees also all other created objects: a `TJSONArray` or `TJSONObject` owns the elements it contains.

Writing a `TJSONFile` is extremely simple, as can be seen from the testcode above: the `AsJSON` method returns a string that describes the JSON data in a valid JSON format. The only thing that needs to be done is to write this to a stream.

To read JSON data, the unit `jsonparser` contains a `TJSONParser` class. It contains a single call: `Parse`, which returns a `TJSONObject` instance corresponding to the JSON data it reads from a string or a stream which can be specified in the constructor of the class.

## 5 A JSON data viewer

Similar to the XML viewer, a JSON data viewer can be coded. The overall code and logic of the program is the same as the XML viewer. Only some detailed procedures are changed. For instance, the call to read the JSON data from a file reads as follows:

```pascal
procedure TMainForm.OpenFile(AFileName : String);
Var
    J : TJSONObject;
    F : TFileStream;
begin
    F:=TFileStream.Create(AFileName,fmOpenRead);
    Try
        With TJSONParser.Create(F) do
            try
                J:=Parse;
                Finally
                    Free;
                end;
            Finally
                F.Free;
            end;
    If J.JSONType<>jtObject then
        Raise Exception.Create(SErrNoJSONObject);
    If Assigned(FJSON) then
        FreeAndNil(FJSON);
    FJSON:=J as TJSONObject;
    SetFileName(AFileName);
    ShowObject(FJSON);
end;
```

The `FXML` variable which existed in the XML viewer has been replaced by a `FJSON` variable, of type `TJSONObject`. Note the check whether the object returned by the `TJSONParser` is a `TJSONObject`. The parser can also return a simple type, and this cannot be displayed as a tree.

14
The call to build up the treeview is very similar to its XML pendant, a simple recursive algorithm:

Function TMainForm.ShowNode(AParent: TTreeNode; AName : String; J: TJSONData) : TTreeNode;

Var
  O : TJSONObject;
  I : Integer;

begin
  If Not (J.JSONType in [jtArray,jtObject]) then
     AName:=AName+' : '+J.AsJSON;
  Result:=TVJSON.Items.AddChild(AParent,AName);
  Result.Data:=J;
  If (J.Count<>0) then
     begin
     if (j is TJSONObject) then
        begin
          O:=J as TJSONObject;
          For I:=0 to O.Count-1 do
            ShowNode(Result,O.Names[i],O.Items[i]);
        end
     else if (j is TJSONArray) then
        begin
          For I:=0 to J.Count-1 do
            ShowNode(Result,IntToStr(I),J.Items[i]);
        end;
      end;
  end;

Note that each node is shown with its value appended if it is a simple type. Array elements are displayed with their index as the node name, and object elements are displayed with their name.

Showing the data of a node in the details panel happens in the same way as for the XML viewer. The code below shows how this is done for a JSON array:

procedure TMainForm.ShowArrayData(A : TJSONArray);

Var
  G : TStringGrid;
  I : Integer;
  J : TJSONData;

begin
  DataTopLabel(SArrayData);
  G:=TStringGrid.Create(Self);
  G.Parent:=PDetails;
  G.Align:=alClient;
  G.FixedCols:=0;
  G.RowCount:=2;
  G.ColCount:=1;
  G.Cells[0,0]:=SElementValue;
  G.RowCount:=1+A.Count;
  G.Options:=G.Options+[goColSizing];
If (G.RowCount > 0) then
    G.FixedRows := 1
else
    G.FixedRows := 0;
For I := 1 to A.Count do
    begin
        J := A.Items[i - 1];
        If (J.JSONType in [jtArray, jtObject, jtNull]) then
            G.Cells[0, I] := J.AsJSON
        else
            G.Cells[0, I] := J.AsString;
        end;
end;

As can be seen, the array is displayed in a grid with a single column: each element of
the array is in a row. If it is a simple type (boolean, number or string) then the value is
displayed. For complex types, the JSON representation is displayed.

To display objects in the details panel, the same logic is followed, but the grid contains 2
columns: the first one contains the name of each element in the object, the second displays
the value.

Last but not least, a new value can be added to the tree. The NewElement method of the
main form takes care of this:

Function TMainForm.NewElement(P : TJSONObject; PT : TTreeNode) : TJSONObject;

Var
    NT : TJSONType;
    EN, EV : String;
    TN : TTreeNode;
    I : Integer;

begin
    Result := Nil;
    With TNewElementForm.Create(Self) do
        try
            NeedName := P is TJSONObject;
            If (ShowModal <> mrOK) then
                Exit;
            NT := DataType;
            EN := ElementName;
            EV := ElementValue;
            finally
                Free;
            end;
        end;

In this first part, a dialog is displayed to query the user for the value which should be
added. Note that the NeedName property of TElementForm decides whether the user
must enter a name for the value or not: in case a new value is added to an object, the new
value must also get a name.

After all data was collected from the user, the new value can actually be created and added
to the parent object. Since the fjson unit contains 2 types for a JSON number value, some
logic is required to see whether a float or an integer value should be created:

Case NT of
jtNumber : if TryStrToInt(EV,I) then
    Result:=TJSONIntegerNumber.Create(i)
  else
    Result:=TJSONFloatNumber.Create(StrToFloat(EV));
jtNull : Result:=TJSONNull.Create;
jtString : Result:=TJSONString.Create(EV);
jtBoolean : Result:=TJSONBoolean.Create(StrToBool(EV));
jtArray : Result:=TJSONArray.Create;
jtObject : Result:=TJSONObject.Create;
end;
If P is TJSONObject then
  (P as TJSONObject).Add(EN,Result)
else if (P is TJSONArray) then
  (P as TJSONArray).Add(Result);
  TN:=ShowNode(PT,EN,Result);
  TVJSON.Selected:=TN;
end;

The above code shows the use of the constructors of all available JSON types, the last lines - as in the XML viewer - just show the newly added node.

There is no XPATH pendant for JSON, so a method to go to a certain JSON value is not available, but such a method could easily be constructed if required. With this the JSON data viewer is finished, and it should look more or less as in figure 3 on page 18, where the sample data presented above is displayed.

6 Conclusion

Despite the fact that Object Pascal is not a dynamically typed language and hence is not perceived to be a language suitable for the web, there are plenty of tools available to treat data formats which are commonly used to transfer data in web applications. The tools presented in this article are just the tools that are shipped with Free Pascal/Lazarus by default: many others are available, each with its own accents, but all alike in their overall structure. All these tools show that Object Pascal is perfectly capable of handling dynamical data, and has the added benefit of ensuring type safety in the code - something of great value to pascal programmers.

References

Figure 3: The JSON viewer application

```
L document: /home/michael/fpc/packages/fcl-json/tests/a.js

- addressbook
  - name: "Mary Lebow"
  - address
    - street: "5 Main Street"
    - city: "San Diego, CA"
    - zip: 91912
  - phonenumbers
    - 0: "619 332-3452"
    - 1: "664 223-4667"

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Mary Lebow</td>
</tr>
<tr>
<td>address</td>
<td>{ &quot;street&quot;: &quot;5 Main S</td>
</tr>
</tbody>
</table>
| phonenumbers | "619 332-3452", "6!  ```