

Introduction to XML (part 2)

by Rob Collins

Reprise

Hopefully, you will have already downloaded the software mentioned in my previous XML article. To remind you, you'll need two editors: one for plain text and one for XML. Recommended software: EditPad, postcardware from www.jgsoft.com for plain text editing; XML Spy from www.xmlspy.com (shareware, \$54); and Internet Explorer 5 (IE5) from a small web startup company somewhere in the States. Recommended book: "XML in Action" by William J. Pardi (Microsoft Press, ISBN 0-7356-0562-9).

Describing XML data

If an XML document conforms to the general way XML should be written, it is called "well-formed". The main point here is that the tags should be matching and correctly nested, as in my XML e-mail example:

```
<?xml version="1.0" encoding="UTF-8"?>
<MEMO>
  <TO>Steve</TO>
  <FROM>Rob</FROM>
  <CC>Joanna</CC>
  <SUBJECT>XML demo</SUBJECT>
  <BODY>Hello, World!</BODY>
</MEMO>
```

But how do we know if this is correct XML for the recipient to understand? Perhaps the parsing program is expecting

```
<MAIL_TO>Steve</MAIL_TO>
<MAIL_FROM>Rob</MAIL_FROM>
<COPY_TO>Joanna</COPY_TO>
```

in place of lines 3, 4 and 5 above. This would also be well-formed, but not necessarily correct. We need something to say which elements are permitted, and which are required. (<TO>Steve</TO> is an element.) When viewing XML in a browser, we merely need it to be well-formed, but for a program that uses the data, it must be "valid". This means that it matches some description of the data, for which there are two approaches.

The first way of validating XML was to use a Document Type Definition (DTD), as an XML optional extra. This was used originally to check SGML. Here is a DTD for the e-mail above, taken from 'XML in Action', p.49:

```
<?xml version="1.0"?>
<!DOCTYPE EMAIL [
  <!ELEMENT EMAIL (TO, FROM, CC, SUBJECT,
  BODY)>
  <!ELEMENT TO (#PCDATA)>
  <!ELEMENT FROM (#PCDATA)>
  <!ELEMENT CC (#PCDATA)>
  <!ELEMENT SUBJECT (#PCDATA)>
  <!ELEMENT BODY (#PCDATA)>
]>
```

However, some people have found it difficult to use DTDs because it requires developers to learn another syntax in which to write them, and this DTD language is not extensible.

On the principle that good compilers should be written in the language they're compiling, Microsoft has put forward an alternative approach that uses XML. It is called an XML schema, with the goals of being simple, extensible and

powerful enough to reflect the sort of data constraints found in modern relational databases, as well as written in XML. For example, with an XML schema, you can specify constraints that the data must contain a positive integer, or be in a certain range, or match a pattern. You can add your own properties, such as documentation, or links to external help. It's good, but not yet a widely implemented standard.

Here's part of a more powerful XML schema for the e-mail example. The content of an element can be *empty*, *textOnly*, *eltOnly* (=sub-elements) or *mixed* (=text and sub-elements).

```
<?xml version="1.0" encoding="UTF-8"?>
<Schema name="email">
  <AttributeType name="priority"
    dt:type="enumeration"
    dt:values="NORMAL LOW HIGH" />

  <ElementType name="to"
    content="textOnly" />
  <ElementType name="from"
    content="textOnly" />

  <ElementType name="cc"
    content="textOnly" />
  <ElementType name="subject"
    content="textOnly" />
  <ElementType name="body"
    content="textOnly" />

  <ElementType name="email"
    content="eltOnly">
    <attribute type="priority"
      default="NORMAL" />
    <element type="to" minOccurs="1"
      maxOccurs="*" />
    <element type="from" minOccurs="1"
      maxOccurs="1" />
    <element type="cc" minOccurs="0"
      maxOccurs="*" />
    <element type="subject" minOccurs="0"
      maxOccurs="1" />
    <element type="body" minOccurs="0"
      maxOccurs="1" />
  </ElementType>
</Schema>
```

Document Object Model

Having managed to avoid Microsoft's go-subs for years, I've now been forced to embrace their XML DOM. Details of the Document Object Model can be found in chapters 5 and 6 of "XML IE5", by Alex Homer, from Wrox Press Ltd (ISBN 1-861001-57-6). There are nodes, node lists, child nodes and lots of other stuff you can read about, or see the COM interface definitions by importing the XML type library from IE5 into Delphi. You do this by going to the Project menu, selecting *Import Type Library...*, and choosing the last of Microsoft's entries, which reads the interface from *msxml.dll* in your system directory. By default this will construct a source file *MSXML_TLB.pas* in your \Borland\Delphi5\Imports directory. Fortunately, we don't have to read 3,500 lines of interface

code: just pick out the interesting definitions shown bold in the code below, e.g. *documentElement*, *childNodes*, *nodeName*, *text*, *load*, *readyState*, *length*, *item*, *transformNode*.

XML Data Islands

In an HTML document, we can refer to XML data with an external file reference, or have XML embedded into the document, with a suitable ID label so as we can get at its contents. We could embed the e-mail XML as a data island, and use JavaScript to extract the data and insert into the HTML:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0
                                Transitional//EN">
<HTML> <HEAD>
<SCRIPT LANGUAGE="JavaScript"
FOR=window EVENT=onload> start(); </SCRIPT>
<SCRIPT LANGUAGE="JavaScript">
function start()
{
  var rootElem = email.documentElement;
  var
  rootLength = rootElem.childNodes.length;
  for (cl=0; cl<rootLength; cl++)
  {
    currNode = rootElem.childNodes.item(cl)
    switch (currNode.nodeName)
    {
      case "TO":
        todata.innerText=currNode.text;
        break;
      case "FROM":
        fromdata.innerText=currNode.text;
        break;
      case "SUBJECT":
        subjectdata.innerText=currNode.text;
        break;
      case "BODY":
        bodydata.innerText=currNode.text;
        break;
    } } }
</SCRIPT>
<TITLE>Untitled</TITLE>
</HEAD>
<BODY>
<XML ID="email">
  <MEMO>
    <TO>Steve</TO>
    <FROM>Rob</FROM>
    <CC>Joanna</CC>
    <SUBJECT>XML demo</SUBJECT>
    <BODY>Hello, World!</BODY>
  </MEMO>
</XML>

  <DIV ID="to" STYLE="font-weight:bold;
font-size:16">To:
  <SPAN ID="todata"
    STYLE="font-weight:normal">
  </SPAN>
</DIV><BR>

  <DIV ID="from" STYLE="font-weight:bold;
font-size:16">From:
  <SPAN ID="fromdata"
    STYLE="font-weight:normal">
  </SPAN>
</DIV><BR>

  <DIV ID="subject"
    STYLE="font-weight:bold;
font-size:16">Subject:
  <SPAN ID="subjectdata"
    STYLE="font-weight:normal">
  </SPAN>
</DIV><BR>

  <HR><SPAN ID="bodydata"
    STYLE="font-weight:normal"></SPAN><P>
</BODY>
</HTML>
```

External XML Data

Using XML data islands is convenient for small blocks of data, but breaks the idea of separating the format from the content. The real power can be seen when we have external file references. We can load an XML data file into an XML DOM ActiveX object, and walk the data tree, avoiding the hard-coding of headings into our script.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0
                                Transitional//EN">
<HTML> <HEAD>
<SCRIPT LANGUAGE="JavaScript"
FOR=window EVENT=onload>
  loadDoc(); </SCRIPT>
<SCRIPT LANGUAGE="JavaScript">
  var
  xmlDoc =
  new ActiveXObject( "microsoft.xmlDOM" );
  xmlDoc.load( "eg010 memo.xml" );

  function loadDoc()
  {
    if ( xmlDoc.readyState == "4" )
      start()
    else
      window.setTimeout( "loadDoc()", 4000 );
  }

  function start()
  {
    var newHTML = "";
    rootElem = xmlDoc.documentElement;
    for (el=0;
        el<rootElem.childNodes.length; el++ )
    {
      if (el != rootElem.childNodes.length
          - 1)
      {
        newHTML = newHTML +
          "<SPAN STYLE='font-weight:bold;' +
          'font-size:16'>" +
          rootElem.childNodes.item(el).nodeName +
          ": </SPAN><SPAN STYLE" +
          "'font-weight:normal'>" +
          rootElem.childNodes.item(el).text +
          "</SPAN><P>";
      }
      else
      {
        newHTML = newHTML +
          "<HR><SPAN STYLE=
          'font-weight:normal'>" +
          rootElem.childNodes.item(el).text +
          "</SPAN><P>";
      }
    }
    content.innerHTML = newHTML;
  }
</SCRIPT>

  <TITLE>e-mail demo</TITLE>
</HEAD>
<BODY>
  <DIV ID="content"> </DIV>
</BODY>
</HTML>
```

XSL Style Sheets

As an alternative to loading data directly with JavaScript, we can use XSL (eXtensible Stylesheet Language), which like XML schemas, has the same structure and syntax as XML. An XML document can contain a reference to an XSL file, which contains all the formatting information, including which data to select, and in what order (not possible with Cascading Style Sheets). XSL uses pattern matching to apply the format, and this makes it a declarative rather than procedural language. We'll just look at a short example, again from "XML in Action", Ch. 8. The XML data is stored in a separate file:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl"
  href="eg152 plants.xsl" ?>
<CATALOG>
  <PLANT>
    <COMMON>Columbine</COMMON>
    <BOTANICAL>Aquilegia canadensis
      </BOTANICAL>
    <LIGHT>Mostly Shady</LIGHT>
    <PRICE>$3.20</PRICE>
    <AVAILABILITY>04/08/99</AVAILABILITY>
  </PLANT>
  <PLANT>
    <COMMON>Marsh Marigold</COMMON>
    <BOTANICAL>Caltha palustris</BOTANICAL>
    <ZONE>4</ZONE>
    <LIGHT>Mostly Sunny</LIGHT>
    <PRICE>$2.90</PRICE>
    <AVAILABILITY>01/09/99</AVAILABILITY>
  </PLANT>
</CATALOG>
```

The XSL here uses *for-each* and *value-of* to step through the data required:

```
<?xml version="1.0"?>
<xsl:template xmlns:xsl="uri:xsl">
  <xsl:for-each select="CATALOG/PLANT/NAME">
    <SPAN STYLE="font-weight:bold; font-size:20">
      <xsl:value-of select="COMMON"/>
    </SPAN>
    <SPAN STYLE="font-weight:bold">
      (<xsl:value-of select="BOTAN"/>)
    </SPAN>
    <P></P>
  </xsl:for-each>
</xsl:template>
```

The HTML document uses JavaScript to load the XML and XSL into two separate XML objects. The style sheet object is applied to the XML object using the *transformNode* method.

```
<HTML><HEAD>
<SCRIPT LANGUAGE="JavaScript"
  FOR="window" EVENT="onload">
var source =
  new ActiveXObject("Microsoft.xmlDOM");
source.load("eg152 plants.xml");
var style = new
  ActiveXObject("Microsoft.xmlDOM");
style.load("eg152 plants.xsl");
document.all.item
  ("xslContainer").innerHTML =
source.transformNode(style.documentElement);
</SCRIPT>

<TITLE>eg152 plants</TITLE></HEAD>
<BODY>
  <DIV ID="xslContainer"></DIV>
</BODY>
</HTML>
```

This finally produces the displayed data:

Columbine (Aquilegia canadensis)

Marsh Marigold (Caltha palustris)

So far in these articles, I hope you have understood something of what XML can do for you. They've been more of a taster than a tutorial, so the hard <work remains="for you"> <to do="Get"> </to> </work>.

Please send comments or questions to rob.collins@tcpq.com.

INPRISE/BORLAND ANNOUNCES BORLAND C++BUILDER 5

Revolutionizes C++ Development by Fusing C++ with the Latest Internet Standards

Inprise/Borland have announced Borland C++Builder 5. The company also announced it will soon offer its core C++ compiler technology, upon which C++Builder 5 is based, free for all developers. C++Builder 5 is scheduled to be available next month in three editions - *C++Builder 5 Enterprise*, *C++Builder 5 Professional*, and *C++Builder 5 Standard*.

"C++Builder 5 shakes-up the C++ landscape by fusing the Internet with C++ development," said Dale Fuller, interim president and CEO of Inprise/Borland. "This breakthrough release brings together Borland's industry standard C++ development environment with the latest Internet standards, XML and HTML 4, delivering the highest-performance web server applications. Inprise is committed to being a leading independent provider of rapid application development tools for all major platforms, including Linux, Windows, and Solaris."