Interfacing PowerBuilder with the Web

PART 1 OF 2

Developing an XML parser using PowerBuilder

JP Morgan in London recently used PowerBuilder to implement an XML interface between a Web-based bond trading system and one of their back-office systems. This is the first of two articles that describe how this system was developed.

Some Background

Trading bonds in Europe was a very old-fashioned process involving many manual processes and disparate systems. When a dealer wanted to purchase a bond they would contact a bank, such as JP Morgan, by telephone or fax. The details of the trade would be written down and then entered into a back-office system. The dealers, banks, and clearinghouses all had computer systems but little information was communicated electronically. The same deal might be entered into three different systems before it was settled.

The major players in the European bond market (JP Morgan, Deutsche Bank, and Citibank) invited a British software house called Capital Net to develop a system to automate the market. The result was a Web-based B2B bond trading system called IssueLink that links dealers, banks, and clearing systems.

Participants can access IssueLink via a browser or by developing an XML interface to their own systems. JP Morgan decided to pursue the XML option. The core component in the interface is an NT Service written in PowerBuilder. The service runs 24/7, listening for messages from IssueLink and parsing them when they arrive. Depending on the type of message, different actions are performed, such as validating the trade, updating a back-office system, or notifying a participant. Some incom-

ing messages may trigger outgoing messages that are processed by IssueLink.

In the nine months since it first went live, the interface with IssueLink has processed trades worth $20,000,000,000. Under the old manual system trades could take over an hour to complete. Over a quarter of the trades now take less than 15 minutes from when the dealer enters the trade until it's added to JP Morgan's back-office system. The quickest trade took only 1 minute 8 seconds.

What Is XML?

Most readers will be familiar with XML, but for those who aren't I'll provide a quick introduction. XML stands for Extensible Markup Language. Like HTML, XML uses tags to delimit the boundaries of data. The basic format is "<name>" to indicate the start of the data and "</name>" to indicate the end of the data. Unlike HTML, which has a fixed set of tags, the tags in XML are user-defined and can be given names that are appropriate for the data. The term *extensible* describes this ability to define your own tags.

An XML document describes the structure of the data as well as its content, but doesn't contain any information on how the data should be presented. You can use Extensible Stylesheet Language Transformations (XSLT) to transform the structure of XML documents and Cascading Style Sheets (CSS) to format them.

Because the names of data elements relate to real-world objects, it's possible to determine the purpose of the data without having to refer to anything other than the XML file. Because of this, XML is said to be self-describing.

XML documents can contain nested data such as "an employee has an address and that address has a house number and a street" as well as repeating data such as "an invoice is for one or more items." It's difficult to represent nested and repeating data in traditional flat files.

XML version 1.0 became a World Wide Web Consortium (W3C) recommendation in February 1998. Many software vendors, including Sybase and Microsoft, are XML-enabling their applications. XML is platform- and vendor-independent, and XML documents are text-based, which means they can be handled by all operating systems. Because of these characteristics, an XML message generated on a Palm Pilot could be interpreted on an IBM mainframe.

The following is a simple XML file that contains data relating to a presentation at TechWave.

```
<?xml version="1.0"?>
<presentation code="ID352">
<!--Example XML file-->
<title>A PB / XML Messaging System</title>
<presenter>Paul Donohue</presenter>
<audience>PowerBuilder Developers</audience>
<time>13:30</time>
<date>2001-08-13</date>
</presentation>
```

This example shows the three parts of an XML document that I'll discuss in this article.

1. ***Elements are the core of an XML document:*** They consist of a start and an end tag with the data between these tags. Elements may also have child elements or attributes. In the example, presentation, title, and time are among the elements.
2. ***Attributes are also name/value pairs: However,*** the name and data are separated by an equal sign. They belong to an element but are not elements on their own, rather they add some sort of qualification to the element they belong to. All attributes are strings and enclosed in single or double quotes. In the example there's only one attribute, code.
3. ***Comments are used to annotate XML documents:*** They begin with "<!--" and end with "-->". The example has one comment – "Example XML file."

Parsing an XML File

Although it would be possible to read and write XML files using PowerBuilder's file functions, they're normally processed with a software component called a *parser*. Parsers fall into two groups: DOM and SAX.

DOM stands for Document Object Model. DOM parsers load an entire XML document into memory when the file is parsed. The data is placed into a tree of nodes that your application can manipulate. This is only suitable for small XML files because of the memory overhead. The methods described in this article use DOM.

SAX stands for Simple API for XML. SAX parsers are event-driven. As the document is parsed, you're notified when certain events occur. An event might be finding the start or end of an element. SAX can handle large files because they're not read into memory. This is useful if you want to process only a subset of the data in a file. The main drawback is

that there's no random access to the data within the document.

Why Use PowerBuilder for an XML Parser?

When you think of XML you probably think of the Internet and Java because these technologies have grown up together. However, there's no reason why you can't process XML messages using almost any language – especially if there's a parser available for your development tool. When I worked on my first XML project, my entire team consisted of PowerBuilder developers. Although some were familiar with Java, they didn't have enough experience to build a mission-critical system using Java. The project deadlines didn't allow any time for training so the team investigated whether we could develop in PB.

A development tool must meet three criteria before it can be used to develop a DOM-based XML system.

1. ***Be able to parse an XML file:*** PB has no low-level XML support built in; however, neither do C++ or Java. Most XML-based systems use a parser to process the XML data rather than perform low-level file operations. The XML parser we decided to use is an OLE object that PB can access easily.
2. ***Be able to manipulate the data in memory:*** When using a DOM parser you need to be able to manipulate the nodes of the XML message in memory. PB's memory management is not as advanced as C++'s; however, our chosen parser did most of the tricky work for us and provided simple methods to access the data. Using these methods and PB's arrays, structures, and nonvisual objects, we could do whatever we wanted with the data.
3. ***Be able to access the database:*** XML is a great technology for data interchange but at some point you probably need to store the data somewhere. PB's strong point is its database access, so it was simple to record details of the messages and to update our back-office systems.

The MS Redistributable Parser

There are many XML parsers available. Many are open source and almost all seem to have a weird name such as expat, XP, Xerces, hex, and Xparse. Microsoft has included a parser with Internet Explorer since version 4.7, called the Microsoft Redistributable Parser, and, as the name suggests, you can distribute the parser royalty-free. Although this parser is shipped as part of IE, it's a separate OLE object so any development tool that can access OLE objects can use it.

If you install IE on your computer, the parser will be registered automatically. The only drawback to this is that the parser that comes with IE may not be the latest version. Early versions don't support the final specification of XSLT so it's best to download the parser on its own from the Microsoft site. Once Internet Explorer is installed it will be associated with the XML extension. Double-click on an XML file in Windows Explorer and IE will display it with the structure and data color-coded.

Property	Description
Async	Specifies if asynchronous download is permitted. When set to TRUE control is returned before the XML document is loaded. (Recommendation = FALSE).
validateOnParse	Specifies whether the parser should validate the document against the Document Type Definition (DTD) when it's parsed. (Recommendation = TRUE).

TABLE 1 XML parser properties

How to Parse an XML File

Using the parser with PowerBuilder is similar to controlling any other OLE application such as Word or Excel. Once you get the hang of the syntax it's easy to get it working. The minimum code to do this is shown in Listing 1. All error checking has been removed for clarity.

CONNECTING

The OLE object variable acts as a reference to the XML parser. The ConnectToNewObject() function invokes the Microsoft XML parser. The parser that's shipped with IE5 is identified by the class name "Microsoft.XMLDOM". If you install a newer version, you'll have to use the class name "MSXML2.DOMDocument.3.0". Use PowerBuilder's IsValid() function to test if the ConnectToNewObject() function was successful.

Once you've connected to the OLE object you can access the parser's methods and properties using notation similar to "<ole_object_variable>.<method_or_property>". You can set many different attributes. Table 1 shows some useful attributes.

LOADING

The load() method loads and parses an XML document. If an error occurs while parsing the file, the load method returns false and the parseError attribute is populated. This structure will pinpoint the location of the error as well as the nature of the problem. The important properties of parseError are shown in Table 2.

Walking the Tree

Remember that DOM parsers load the entire XML file into memory, whereas SAX parsers are event-driven. One advantage of DOM is that you can go straight to the node you require using the selectSingleNode() method. This is fine if you know the structure of the XML files that you'll be processing. However, if you're unsure what the structure will be, you require a generic solution.

Once the file is in memory it's represented by a tree of nodes. Each node may have siblings (elements at the same level), children (elements at a lower level), or attributes. The term *walking the tree* refers to processing each node of the XML file starting at the root node and moving on to all its children, attributes, and siblings. Each time you reach a fork in the tree you go down one of the branches, forking as required until you reach the leaf nodes. At this point you work your way back up the tree, visiting any branches that haven't yet been processed. Although it sounds complicated, using recursion makes it simple.

In Listing 2 an XML file is loaded into a treeview using a recursive function called wf_parse_node. The arguments for this function are the XML node to parse and the treeview item to populate. Start wf_parse_node by giving it the root node of the XML file and the root item in the treeview. Use the "documentElement" attribute to find the XML root node.

Populating a treeview recursively is pretty cool, but if you're developing a business application you'll probably want to do something else with your XML file. The most likely alternatives are updating the database, invoking a business rule object, writing to a file, or sending an e-mail. The recursive method of walking the tree is very useful for finding the nodes you require to carry out this processing.

DISCONNECTING

When you've finished processing one or more XML files you need to disconnect from the browser and tidy up. The DisConnectObject() function will terminate your

Property	Description
ErrorCode	A number that identifies the type of error
Filepos	The absolute position of the error in the file
Line	The number of the line that contains the error
Linepos	The character position of the error (on the error line)
Reason	The reason for the error
SrcText	The text of the line that contains the error

TABLE 2 Important parseError properties

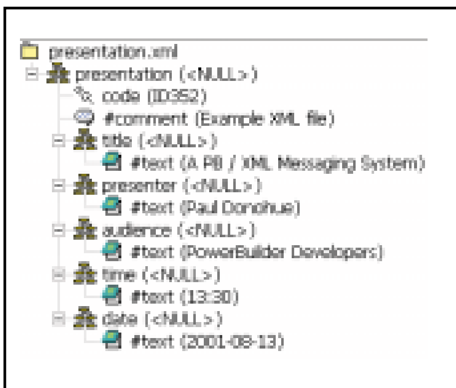

FIGURE 1 XML DOM tree

XML browser session. Make sure you destroy the OLE object variable when you're finished.

A Strange Thing About the XML DOM Tree

There's a feature of the DOM tree that can be confusing. If you use the sample code to parse the XML file shown at the beginning of this article, the results aren't what you would expect. The resulting DOM tree is shown in Figure 1. The labels show the node name and its value in brackets.

The root node is called *presentation* and has no value. It has one attribute node, a comment node, and five child element nodes. The attribute and comment nodes have values as you would expect, however, the element nodes don't have a value of their own. The element's value is stored in a text child node. This strange behavior is logical if you think of elements as containers. Elements may contain other elements, attributes, comments, or data (stored in a child node).

As a workaround to finding the value of an element, check if the first node in its child node list is of type "TEXT". In this case you can assume that the text node is the value for the element. A PowerBuilder example that demonstrates how to code for this is available on the ***PBDJ*** Web site, www.PowerBuilderJournal.com.

In Part 2 I'll discuss how JP Morgan was able to develop an NT Service using PowerBuilder. The service processes the XML messages from IssueLink.

Resources

1. Birbeck, M., et al. (2001). *Professional XML*. Wrox Press. This is the best XML book I've come across.
2. *A commercial site with good XML news:* www.xml.com
3. *The XML industry portal:* www.xml.org
4. *The World Wide Web Consortium – good for pure XML information:* www.w3.org/xml
5. *Microsoft's XML page for developers – bit MS-centric but quite good:* http://msdn.microsoft.com/xml
6. *Information on Capital Net's IssueLink system:* www.capn.com/issuelink

Author Bio

Paul Donohue has 15 years' experience as a solution provider. He has worked with PowerBuilder since version 2 and is a Certified PowerBuilder Developer.

pbdj@pauldonohue.com

Listing 1

```
// Declare an OLE object as a reference to the parser
oleobject  lole_xml_document

// Identify the file to parse
string ls_filename = „C:\DEMO.XML„

// Create the OLE Object
lole_xml_document = CREATE oleobject

// Connect to the parser
// NOTE : This example uses the parser from IE5.
// Use the class name „MSXML2.DOMDocument.3.0%
// for the latest version.
lole_xml_document.ConnectToNewObject("Microsoft.XMLDOM")

// Load the file into memory (this will parse it)
lole_xml_document.load(ls_filename)

// *********************************************
// THE XML FILE HAS BEEN PARSED AND IS IN MEMORY
// YOU CAN INVOKE PARSER METHODS TO MANIPULATE IT
// *********************************************
wf_parse_node(this function has some arguments)

// Disconnect from the XML parser
lole_xml_document.DisConnectObject()

// Destroy the OLE object
DESTROY lole_xml_document
```

Listing 2

```
wf_parse_node(node, treeview)

// Arguments
oleobject  aole_node    // The node to process
long       al_treeview_parent    // The treeview item to add
nodes to

// local variables
string     ls_node_type    // The type of the current node
string     ls_node_name    // The name of the current node
string     ls_node_value    // The value of the current node
oleobject  lole_attribute_node_list    // The attributes for
this node
oleobject  lole_attribute_node    // An attribute for this node
oleobject  lole_child_node_list    // The child nodes for this
node
oleobject  lole_child_node    // The current child node
long       ll_max_attribute_nodes    // The number of attribute
nodes
long       ll_max_child_nodes    // The number of child nodes
long       ll_attribute_idx    // A counter
long       ll_child_idx    // A counter
string     ls_label    // The label for the treeview
long       ll_treeview    // The newly inserted treeview item

// Determine the node's name, type and value
ls_node_name = aole_node.nodename
ls_node_type = Upper(aole_node.nodetypestring)
ls_node_value = String(aole_node.nodevalue)

IF IsNull(ls_node_value) THEN
   ls_node_value = "<NULL>"
END IF

// Add this node to the treeview
// I am using a picture index of 1 however
// you could use the nodetypestring to determine
// the node,s type and use an appropriate picture
ls_label = ls_node_name + " (" + ls_node_value + ")"
ll_treeview = tv_xml.InsertItemLast (al_treeview_parent,
ls_label, )

// If this node has attributes process them
lole_attribute_node_list = aole_node.attributes

IF  IsValid(lole_attribute_node_list) THEN
      ll_max_attribute_nodes = lole_attribute_node_list.length
ELSE
      ll_max_attribute_nodes = 0
END IF

FOR ll_attribute_idx = 0 TO ll_max_attribute_nodes - 1
      lole_attribute_node =
lole_attribute_node_list.Item(ll_attribute_idx)
      wf_parse_node (ll_treeview, lole_attribute_node) /* RECUR-
SIVE */
NEXT

// If this node is an element and it has children process them
IF  ls_node_type = "ELEMENT" THEN
      lole_child_node_list = aole_node.childNodes

      IF  IsValid(lole_child_node_list) THEN
            ll_max_child_nodes = lole_child_node_list.length
      ELSE
            ll_max_child_nodes = 0
      END IF

      FOR  ll_child_idx = 0 TO ll_max_child_nodes - 1
             lole_child_node =
lole_child_node_list.Item(ll_child_idx)
             wf_parse_node(ll_treeview, lole_child_node) /* RECUR-
SIVE */
      NEXT

END IF
```


Download the Code!
The code listing for this article can also be located at
www.PowerBuilderJournal.com