Session E-XML

Practical Uses for XML

Doug Hennig
Stonefield Software Inc.

Overview

XML can be a great way to store some types of data or transfer data from one application to another. This session discusses what XML is and presents various methods of generating and parsing XML. It also shows why XML is useful and demonstrates some practical examples of how I've used XML in my applications.

Introduction

I've seen a number of questions over the past several months on several VFP forums asking why someone should use XML. I've been using XML for a variety of purposes for several years so I decided an article on some of the practical uses of XML would be useful.

Before I get into some of the ways you can use XML, I'll first discuss what XML is and how to create and process XML. Part 2 of this article in the next issue discusses why you might want to use XML and shows some of the uses I've made of it in various applications.

What is XML?

XML is an acronym for eXtensible Markup Language. It provides a mechanism for specifying content in a consistent, human-readable format (well, somewhat readable, depending on the complexity of the XML) that can also be parsed by software. Its consistency is maintained by two requirements: that the XML be well-formed and valid.

Well-formed XML

Well-formed XML means it follows certain rules. One rule is that an XML document contains only a single "root" element. That means the content of the document must be between start and end "tags" for this element. A start tag is a name surrounded by "<"and ">" (for example, "<MyTag>") and an end tag is similar but "/" precedes the name (for example, "</MyTag>"). Another rule is that every start tag must either have a matching end tag or have "/" before ">" (such as <MyTag/>) to indicate that it's an empty tag. For example, this is well-formed XML:

```
<tags>
  <mytag>Some text</mytag>
  <mytag>Some other text</mytag>
</tags>
```

This is not well-formed because it contains more than one root element.

```
<mytag>Some text</mytag>
<mytag>Some other text</mytag>
```

This is also not well-formed because not all start tags have matching end tags:

```
<tags>
  <mytag>Some text
  <mytag>Some other text
</tags>
```

Although this looks well-formed, it isn't because XML is case-sensitive, and the case of the start and end tags doesn't match:

```
<tags>
  <mytag>Some text</MyTag>
  <mytag>Some other text</MyTag>
</Tags>
```

Elements are like structured statements in VFP, such as IF...ENDIF, in that they can contain other elements (you can have nested elements just like you can have IF statements inside other IF statements) and they have to be ended in the opposite order they were started. The following is not well-formed because the end tag for tag1 appears before that of tag2 so they aren't properly nested:

```
<tag1><tag2>Some text</tag1></tag2>
```

White space—spaces, tabs, carriage returns, and so forth—between tags is irrelevant. From an XML parser perspective, the following two XML fragments are identical, although the first is obviously easier to read:

```
<tags>
  <mytag>Some text</mytag>
  <mytag>Some other text</mytag>
</tags>

<tags><mytag>Some text</mytag>
<mytag>Some other text</mytag></tags>
```

In addition to text between a start and end tag, elements can also have attributes. These are specified as name-value pairs in the start tag. Attribute values must be delimited with either single or double quotes and each attribute name can only appear once in an element. For example:

```
<tags>
  <mytag Attribute1="some value"
    Attribute2="some value">Some text</mytag>
  <mytag Attribute1="some value"
    Attribute2="some value">Some text</mytag>
</tags>
```

If an element has only attributes and no text, it's usually specified as an empty tag since the end tag isn't required. For example:

```
<tags>

  <mytag Attribute1="some value" Attribute2="some value"/>

  <mytag Attribute1="some value" Attribute2="some value"/>

</tags>
```

Although you can have elements with both text between tags and attributes, it's common to use one or the other. XML in which elements have only text and no attributes is called "element-centric"or "element-based" and XML with elements with only attributes is called "attribute-centric" or "attribute-based."

Since "<," ">," and "/" are special characters in XML, they can't be used in attribute values or text. There are several other characters that also can't be used in XML, including "&" and the single and double quotes. These special characters must be represented with the "entity references" shown in **Table 1**.

Table 1. Entity references for special characters in XML.

Character	Entity Reference
>	&gt;
<	&lt;
&	&amp;
'	&apos;
"	&quot;

For example, the first element is not well-formed because of the "&" but the second is fine:

```
<mytag>Tips & Tricks</mytag>

<mytag>Tips &amp; Tricks</mytag>
```

If XML contains non-ASCII characters, such as the accented character in "Berglunds snabbköp," you must either encode the data as UTF-8 when you create the XML or specify the encoding used for the data in a processing instruction in the XML. To encode in UTF-8, use STRCONV(expression, 9). To specify the encoding, add the following to the start of the XML:

```
<?xml version="1.0" encoding="Windows-1252"?>
```

You can, of course, specify other encoding; for English VFP developers, Windows-1252 is the likely choice.

Failing to either encode the data or include a processing instruction will cause most XML parsers to give an error when you try to load the XML. Although I'll discuss creating and parsing XML later in this article, you can see this problem now. Run XMLDOMGenerateElements.PRG, included in the source code for this article, to create an XML file named Customers.XML from the sample Northwind Customers table, then type the following code to load the XML into the Microsoft XML DOM object:

```
loXML = createobject('MSXML2.DOMDocument')

loXML.async = .F.

loXML.load('customers.xml')

messagebox(loXML.parseError.reason)
```

This displays the error message "An invalid character was found in text content," indicating a problem parsing a non-ASCII character in the XML. That's because the data in Customers.XML is Windows-1252 but no encoding instruction is included in the XML and the XML DOM object expects UTF-8 in the absence of such an instruction.

Valid XML

Just because XML is well-formed doesn't mean it's valid. Valid XML ensures expected elements and attributes are found and they contain valid values. It's similar to the difference between having a DBF that you can open because it isn't corrupted (that is, it's "well-formed") and ensuring it has the expected structure (your code will likely crash if it opens an order table instead of a customer table and tries to access the city field). Like a DBF header that describes the structure of the table, the structure of an XML document can be described with either a Document Type Definition (DTD) or schema. DTDs are an older mechanism; schemas are the preferred approach.

The schema for an XML file describes the elements in that file and what they contain. Here's part of the schema for XML for the Northwind Customers table:

```
<xsd:element name="customers" minOccurs="0" maxOccurs="unbounded">

  <xsd:complexType>

    <xsd:sequence>

      <xsd:element name="customerid">

        <xsd:simpleType>

          <xsd:restriction base="xsd:string">

            <xsd:maxLength value="5"/>

          </xsd:restriction>

        </xsd:simpleType>

      </xsd:element>
```

This indicates there’s an element called “customers” that can appear from zero to an unlimited number of times and it contains several elements, including “customerid” which holds 5-character values. The syntax for schemas is beyond the scope of this article; search the Internet for “xml schema” and you’ll find plenty of reference sites with detailed information.

A schema can either be included in the XML file (an “inline” schema) or in a separate file which is then referenced in the XML file.

There’s a lot more to XML you could learn, such as namespaces, XQuery, and so on, but this overview covers the basics of what we’ll need for the rest of this article and the next one. There are many books and Web sites devoted to XML, so feel free to dig in if you want to know more.

Creating XML

There are several ways you can create XML.

Text commands

Since XML is just text, the simplest way to create it is using VFP’s text-handling commands and functions. This code outputs XML from customer information stored in a collection:

```
lcXML = '<customers>'

for each loCust in loCustomers

  lcXML = lcXML + ;

    '<customer>' + ;

      '<company>' + loCust.cCompanyName + '</company>' + ;

      '<contact>' + loCust.cContactName + '</contact>' + ;

      '<city>' + loCust.cCity + '</city>' + ;

    '</customer>'

next loCustomer

lcXML = lcXML + '</customers>'
```

Text merge is often used instead of string concatenation to create XML. This code produces the same output but is easier to type and to read:

```
lcXML = '<customers>'

for each loCust in loCustomers

  text to lcXML additive noshow textmerge

  <customer>
```

```
    <company><<loCust.cCompanyName>></company>
    <contact><<loCust.cContactName>></contact>
    <city><<loCust.cCity>></city>
    </customer>

    endtext
  next loCust
  lcXML = lcXML + '</customers>'
```

CURSORTOXML()

For data in a VFP table or cursor, CURSORTOXML() is a one-line solution as long as the XML you need to create is formatted the simple way CURSORTOXML() generates it. Here's an example:

```
use home() + 'samples\northwind\customers'
cursortoxml('customers', 'lcXML')
```

Note the unusual syntax: rather than returning an XML string, CURSORTOXML() places the XML into the location specified in the second parameter; in this case, the variable lcXML. You can output to a file by specifying the filename for the second parameter and adding 512 to the fourth "flags" parameter.

The XML generated looks like this:

```
<?xml version = "1.0" encoding="Windows-1252"
  standalone="yes"?>
<VFPData>
  <customers>
    <customerid>BONAP</customerid>
    <companyname>Bon app</companyname>
    <contactname>Laurence</contactname>
    <contacttitle>Owner</contacttitle>
    <address>12, rue des Bouchers</address>
    <city>Marseille</city>
    <postalcode>13008</postalcode>
    <country>France</country>
    <phone>91.24.45.40</phone>
    <fax>91.24.45.41</fax>
  </customers>
</VFPData>
```

In addition to being a single line of code, CURSORTOXML() has several advantages over text commands:

- It automatically replaces special XML characters with their entity references. For example, it outputs “Split Rail Beer & Ale” as “Split Rail Beer &amp; Ale.”
- It automatically adds a processing instruction indicating the encoding and it can optionally UTF-8 encode the data (add 48 to the fourth parameter).
- It can generate a schema for the XML, either included in the XML or output to a separate file.
- It can generate XML that’s element-based (the default or specify 1 for the third parameter) or attribute-based (pass 2 for the third parameter). You’d have to rewrite code that generates element-based XML with text commands to use attributes instead.

CURSORTOXML() is best suited to cases where you have simple, non-hierarchical XML that’s processed by a VFP application.

XMLAdapter

XMLAdapter is a relatively new base class in VFP. It can generate more complex XML than CURSORTOXML(), including hierarchical XML from related tables. XMLAdapter is like a container class; although it has properties and methods of its own, it also contains other objects; in this case, XMLTable objects. Each XMLTable object reprnesents a VFP cursor. XMLTable is itself a container of XMLField objects. By manipulating the properties of the objects in this hierarchy, you have a lot of control over the XML XMLAdapter generates.

Listing 1 shows an example, taken from XMLAdapterGenerateOrders.PRG included in the Subscriber Downloads, that generates hierarchical XML of customers and their orders. The XML looks like this (with some elements removed for brevity):

```
<customers>
  <customerid>BONAP</customerid>
  <companyname>Bon app'</companyname>
  <orders>
    <orderid>10331</orderid>
    <customerid>BONAP</customerid>
    <employeeid>9</employeeid>
    <orderdate>1996-10-16</orderdate>
  </orders>
</customers>
```

Listing 1. XMLAdapter can generate more complex XML than CURSORTOXML() , such as hierarchical XML from related tables.

```
local loXML as XMLAdapter, ;
  loCustomers as XMLTable, ;
  loOrders as XMLTable

* Open the Customers and Orders tables and relate them so
XMLAdapter
* knows how to nest them in a hierarchy.

close tables all
use home() + 'samples\northwind\customers'
use home() + 'samples\northwind\orders' order CustomerID in 0
set relation to CustomerID into Orders

* Create an XMLAdapter and add the Customers and Orders tables to
it.
* Nest Orders under Customers.

loXML       = createobject('XMLAdapter')
loCustomers = loXML.AddTableSchema('customers', .T.)
loOrders    = loXML.AddTableSchema('orders',    .T.)
loCustomers.Nest(loOrders)

* Tell XMLAdapter to nest and format the XML.

loXML.RespectNesting  = .T.
loXML.FormattedOutput = .T.
```

```
* Output and view the XML.

loXML.ToXML('customerorders.xml', '', .T.)
modify file customerorders.xml nowait
```

I must admit I haven't used XMLAdapter much. It's really a wrapper around the Microsoft XML DOM object and I find that object more straightforward to use.

Microsoft XML DOM

The MS XML DOM object provides a flexible mechanism for generating XML. One of the nice things about the XML DOM is that it provides an object-oriented front-end to XML. The most frequently used methods for generating XML are createElement, which creates a node object, and appendChild, which adds a node object to the XML. Use appendChild on the XML DOM object itself to add the root node and on a node to add an element to that node.

The code shown in **Listing 2**, which was taken from XMLDOMGenerateElements.PRG, shows how to generate element-based XML from the Northwind Customers table using the XML DOM.

Listing 2. This code generates element-based XML using the MS XML DOM object.

```
use home() + 'samples\northwind\customers'
loXML = createobject('MSXML2.DOMDocument')

* Create the root "customers" element and add it to the XML.

loRoot = loXML.createElement('customers')
loXML.appendChild(loRoot)

* Go through each customer and create the "customer" element.

scan
  loCust = loXML.createElement('customer')

* Create the "company" element and add it to the customer element.

  loNode = loXML.createElement('company')
  loNode.text = trim(COMPANYNAME)
  loCust.appendChild(loNode)

* Create the "contact" element and add it to the customer element.

  loNode = loXML.createElement('contact')
  loNode.text = trim(CONTACTNAME)
  loCust.appendChild(loNode)

* Create the "city" element and add it to the customer element.

  loNode = loXML.createElement('city')
  loNode.text = trim(CITY)
  loCustomer.appendChild(loNode)

* Add the customer element to the root customers element.

  loRoot.appendChild(loCust)
endscan

* View the XML.

strtofile(loXML.XML, 'customers.xml')
modify file customers.xml nowait
```

This may seem like a lot of code but it's very straightforward and easy to read. Also, the code is simpler if you want attribute-based XML instead; the following, taken from XMLDOMGenerateAttributes.PRG, replaces the code inside the SCAN loop:

```
loCust = loXML.createElement('customer')
loCust.setAttribute('company', trim(COMPANYNAME))
loCust.setAttribute('contact', trim(CONTACTNAME))
loCust.setAttribute('city', ;  trim(CITY))
```

```
loRoot.appendChild(loCust)
```

Note that while CURSORTOXML() by default adds white space to the XML (tabs to indent elements and carriage returns/line feeds, or CRLF, after each element), the XML DOM does not. To add white space using the XML DOM, use the createTextNode method to create a node with the desired text and add it to the appropriate node. For example, the code in **Listing 3**, taken from XMLDOMGenerateElementsWhiteSpace.PRG, is similar to the code in XMLDOMGenerateElements.PRG but indents each element and places it on its own line.

Listing 3. This code is similar to that in Listing 2 but adds white space to the XML.

```
#define ccCRLF chr(13) + chr(10)

#define ccTAB  chr(9)

scan

  loCust = loXML.createElement('customer')


* Add a CRLF and two tabs before each company element.


  loText = loXML.createTextNode(ccCRLF + ccTAB + ccTAB)

  loCust.appendChild(loText)


* Create the "company" element and add it to the customer element.


  loNode = loXML.createElement('company')

  loNode.text = trim(COMPANYNAME)

  loCust.appendChild(loNode)


*** additional code adding contact and city elements removed for
brevity


* Add a CRLF and one tab after the city element (that is, before
the

* closing customer tag).


  loText = loXML.createTextNode(ccCRLF + ccTAB)
  loCust.appendChild(loText)

* Add a CRLF and one tab before each customer element.

  loText = loXML.createTextNode(ccCRLF + ccTAB)
  loRoot.appendChild(loText)

* Add the customer element to the root customers element.

  loRoot.appendChild(loCust)
endscan

* Add a CRLF after the last customer element (ie. before the
closing
* customers tag).

loText = loXML.createTextNode(ccCRLF)
loRoot.appendChild(loText)
```

The XML DOM can generate hierarchical XML, something CURSORTOXML() can't do. The code in **Listing 4**, taken from XMLDOMGenerateOrders.PRG, generates XML containing both customers and their orders, formatted like this:

```
<customer company="Alfreds Futterkiste"
  contact="Maria Anders" city="Berlin">
  <order orderid="10643" orderdate="08/25/1997"
reqdate="09/22/1997"/>
  <order orderid="10692" orderdate="10/03/1997"
reqdate="10/31/1997"/>
</customer>
```

Listing 4. The XML DOM can generate hierarchical XML, such as order child elements for each customer. Note: code adding white space to the XML is omitted for brevity.

```
close tables all
use home() + 'samples\northwind\customers'
use home() + 'samples\northwind\orders' in 0
loXML = createobject('MSXML2.DOMDocument')

* Create the root "customers" element and add it to the XML.

loRoot = loXML.createElement('customers')
loXML.appendChild(loRoot)

* Go through each customer and create the "customer" element.

scan
  loCust = loXML.createElement('customer')

* Set the attributes.

  loCust.setAttribute('id',      trim(CUSTOMERID))
  loCust.setAttribute('company', trim(COMPANYNAME))
  loCust.setAttribute('contact', trim(CONTACTNAME))
  loCust.setAttribute('city',    trim(CITY))

* Get the orders for the selected customer, then go through each
order
* and create the "order" element.

  liCustID = CUSTOMERID
  select * from Orders ;
    where CustomerID = liCustID ;
    into cursor CustOrders
  scan
    loOrder = loXML.createElement('order')
```

```
* Set the attributes.

    loOrder.setAttribute('orderid',   ORDERID)
    loOrder.setAttribute('orderdate', ORDERDATE)
    loOrder.setAttribute('reqdate',   REQUIREDDATE)

* Add the order element to the customer element.

    loCust.appendChild(loOrder)
  endscan

* Add the customer element to the root customers element.

  loRoot.appendChild(loCust)
endscan

* View the XML.

strtofile(loXML.XML, 'customerorders.xml')
modify file customerorders.xml nowait
```

Note the XML generated by the XML DOM doesn't contain an encoding instruction. As I demonstrated earlier, that's a problem, so use the createProcessingInstruction method:

```
loNode = loXML.createProcessingInstruction('xml', 'version="1.0" ' + ;
  'encoding="Windows-1252"')
loXML.appendChild(loNode)
```

An oddity about the XML DOM is that it strips the encoding information from the XML property; the instruction statement appears as "<?xml version = "1.0"?>. " Instead, use the save method of the XML DOM object to write the XML to a file, in which case the encoding instruction is included.

There are several benefits of using the XML DOM over text commands:

- As with CURSORTOXML(), it automatically replaces special XML characters with their entity references.
- It automatically generates both opening and closing tags.
- You get IntelliSense on the XML DOM and element objects if you declare them in a LOCAL statement.

Obviously, using the XML DOM is more work than using CURSORTOXML(). In addition to coding each element to be output, you also have to generate a schema manually if you wish to have one. However, it's a lot more flexible and can generate any type of XML you need. In addition, you can write wrapper classes that take care of some of this work for you. For example, I created a subclass of Custom called SFXML that generates and parses attribute-based XML, including adding an encoding instruction and optionally adding white space. Due to space limitations, I won't show the code for this class here, but notice how much simpler SFXMLGenerate.PRG, shown in **Listing 5**, is than XMLDOMGenerateAttributes.PRG.

Listing 5. A custom wrapper class, such as SFXML, can handle much of the mundane work of using the XML DOM.

```
close tables all

use home() + 'samples\northwind\customers'

loXML = newobject('SFXML', 'SFXML.vcx')


* Set the root and element node names and flag that we should add
white

* space.


loXML.cRootNode      = 'customers'

loXML.cElementNode   = 'customer'

loXML.lAddWhiteSpace = .T.


* Go through each customer and create an element for it.


scan

  loCust = loXML.CreateNode()


* Set the attributes.


  loCust.setAttribute('company', trim(COMPANYNAME))
  loCust.setAttribute('contact', trim(CONTACTNAME))

  loCust.setAttribute('city',    trim(CITY))


* Add the customer element to the XML.


  loXML.SaveNode(loCust, .T.)

endscan


* View the XML.


loXML.SaveXML('customers.xml')

modify file customers.xml nowait
```

Parsing XML

As with creating XML, there are a variety of mechanisms for parsing XML.

Manual parsing

The VFP STREXTRACT() function seems like it was added to the language specifically for parsing HTML and XML because it returns the text between delimiters. For example, this code retrieves the company names from XML formatted as "<companyname>Name of company</companyname>":

```
for lnI = 1 to occurs('<companyname>', lcXML)

  lcCompany = strextract(lcXML, '<companyname>', '</companyname>',
lnI)

next lnI
```

However, this gets tedious very quickly, especially as the XML gets more complex. Parsing attribute-based XML is particularly nasty. Even worse is XML that has optional attributes or elements; your code has to account for missing information when parsing the strings. For example, RSS, used for blogs and news aggregation, is XML. The guid element has an optional isPermaLink attribute, so that element may appear as "<guid>" (that is, with the ">" following "guid") or "<guid isPermaLink="true">", so retrieving the text between the element's tags is a little complex:

```
lcGUID = strextract(lcXML, '<guid>', '</guid>')

if empty(lcGUID)

  lcGUID = strextract(lcXML, '<guid ', '>', 1, 4)
  lcGUID = strextract(lcXML, lcGUID, '</guid>')
endif empty(lcGUID)
```

XMLTOCURSOR()

The companion for CURSORTOXML(), XMLTOCURSOR(), does a fine job of reading XML into a cursor as long as the XML is relatively simple. One tip for using XMLTOCURSOR(): to ensure data types are handled properly when no schema is available, create a cursor with the appropriate structure first, specify its name for the second parameter, and add 8192 to the third “flags” parameter. Otherwise, a field in the XML that contains integer values may be converted to logical values when XMLTOCURSOR() creates the cursor. Of course, if the XML has a schema, this isn’t an issue because VFP creates a cursor with the correct structure in that case.

This code reads the Customers.XML file created by the sample CURSORTOXMLGenerateElements.PRG into a cursor named TEMP:

```
xmltocursor('customers.xml', 'temp', 512)
browse
```

The third parameter indicates that the first is the name of a file. To parse the XML from a string instead, pass 0 or omit the third parameter. There are other parameters you can use as well; see the VFP help topic for this function for details.

XMLTOCURSOR() is pretty good at determining a schema for simple, non-hierarchical XML if one isn’t provided. For example, the code above can also be used on Customers.XML generated by the XMLDOMGenerateElements.PRG and XMLDOMGenerateAttributes.PRG sample programs, indicating that it can handle both element and attribute based XML. However, for more complex XML, you’ll likely use the XML DOM.

XMLAdapter

The XMLAdapter class can both generate and parse XML. Parsing XML follows a path opposite to generating it: load the XML, then call the ToCursor method of each table object to create a cursor. Here’s an example, taken from XMLAdapterParseOrder.PRG, which loads the XML generated by XMLAdapterGenerateOrders.PRG and converts it back into cursors:

```
loXML = createobject('XMLAdapter')
loXML.LoadXML('customerorders.xml', .T.)

* Create a cursor for each table.

for each loTable in loXML.Tables
  loTable.ToCursor()
```

```
    browse
    use
  next loTable
```

Unlike XMLTOCURSOR(), XMLAdapter doesn't work well without a schema, so you'll likely want to use this class with XML generated by XMLAdapter or by Visual Studio Datasets.

XML DOM

The load and loadXML methods of the XML DOM object load XML from a file and a string, respectively. By default, these methods return before the XML DOM finishes parsing the XML, which can be a problem if you load a large XML document and then try to process it before it has finished loading. To prevent this, set the async property to .F. first.

Parsing XML using the XML DOM object means learning some new syntax called XPath for locating nodes. There are many Web sites available with information on XPath, but **Table 2** shows the most commonly used syntax.

Table 2. Commonly used XPath syntax for specifying nodes.

Syntax	Specifies
/Element	Find the specified element name under the root node
/Element1/Element2	Find Element2 as a child of Element1
/Element[@Attribute='Value']	Find the specified element name with the specified attribute containing the specified value

Two commonly used methods to locate nodes are selectNodes and selectSingleNode. As their names imply, selectNodes returns a collection of node objects and selectSingleNode returns a single node object. Once you have a node object, you can use its getAttribute property to read attribute information. For element-based XML, you can use selectNodes and selectSingleNode calls on nodes to find elements within those nodes.

For example, **Listing 6** shows code taken from XMLDOMParseOrders.PRG that loads the XML generated by XMLDOMGenerateOrders.PRG and retrieves information about orders and customers:

Listing 6. Use selectSingleNode and selectNode to find nodes in the XML.

```
local loXML as MSXML2.DOMDocument.3.0
loXML = createobject('MSXML2.DOMDocument')


* Load the XML.


loXML.async = .F.
loXML.load('customerorders.xml')

* Find order ID 10308; either of these works.

loNode = loXML.selectSingleNode("/customers/customer/order[" + ;
  "@orderid='10308']")
loNode = loXML.selectSingleNode("/customers/customer[@id='ANATR']"
+ ;
  "/order[@orderid='10308']")

* Get and display the attributes of that order.

lcOrdDate = loNode.getAttribute('orderdate')
ldOrdDate = cast(lcOrdDate as D)
lcReqDate = loNode.getAttribute('reqdate')
ldReqDate = cast(lcReqDate as D)
messagebox('Order 10308 was ordered on ' + transform(ldOrdDate) +
;
  ' and is required by ' + transform(ldReqDate))

* See how many customers we have in London.

loNodes = loXML.selectNodes("/customers/customer[@city='London']")
messagebox('There are ' + transform(loNodes.length) + ;
  ' customers in London.')

* Display the orders for the first one.

loNode   = loNodes.item(0)
loOrders = loNode.selectNodes('order')
lcOrders = ''
for each loOrder in loOrders
```

```
    lcOrders = lcOrders + iif(empty(lcOrders), '', chr(13)) + ;
        loOrder.getAttribute('orderid')
  next loOrder
  messagebox(loNode.getAttribute('company') +
    ' has the following orders:' + chr(13) + chr(13) + lcOrders)
```

As with generating XML, you can create a wrapper class for parsing specific XML. For example, I've created a class called SFRSS that parses RSS-formatted XML. The SFXML class I discussed earlier can parse attribute-based XML without you having to know XPath syntax; see SFXMLParse.PRG for an example.

Why XML?

I first heard about XML at a presentation Ken Levy did at a VFP conference years ago. My reaction at the time was "what's the big deal?" After all, XML is just text, and there are lots of ways you can format text.

Since then, I've come to realize that XML is special because it's consistently formatted text. That means you can easily generate it and easily parse it. Let's look at some of the advantages of XML over other types of formats.

Binary-formatted information is useful for many types of data structures; DBF files are an example of a binary format. However, binary information has a few drawbacks. First, creation and parsing must be completely custom code. Second, the format is proprietary and not easily shared. Finally, binary data is not human readable. Not that you want to read data structures often but it can be useful when something goes wrong or you want to debug creation or parsing. It's also useful if you need to edit the information using a different tool or API than normally used.

For example, the properties of objects in a database container are stored in the PROPERTIES memo of the DBC file. That memo contains binary information. Each property value is prefixed with a code that identifies the property and several bytes containing the length of the value. As a result, it isn't easy to change the value of a property by hacking the DBC. Why would you want to do that when you can use DBSETPROP()? DBSETPROP() works fine for most properties but some are considered to be read-only and can't be changed using that function. To change read-only values programmatically, you have to find where in the memo the value is stored by looking for the proper code, overwrite the current value with the new one, and adjust the length bytes if necessary. Any mistake makes the record in the DBC invalid. Now, suppose the properties were stored like this instead:

```
<properties>
  <property name="Caption">Some Caption</property>
  <property name="Default">150</property>
</properties>
```

Not only is this easily read, it would be very easy to make any necessary changes. Of course, this would also take up a lot more space than the binary version does, making the DCT file considerably larger. That's one of the downsides of XML.

Another drawback with binary information is that it's more difficult to move it from one application to another. For example, something as simple as passing an array from a VFP application to a COM object or .Net application requires some special tricks, while passing a string like XML is generally easy. Another example is sending information to a Web application, where HTML protocol and firewalls make it way more difficult to transfer binary information than to transfer text.

One benefit that binary format has over XML is speed. Since the code needed to parse the information is specific for the format, it can be highly optimized and, of course, there's less data to store and pass. Although XML parsers can be fast for small amounts of text, they can be very slow when the size gets into the megabytes.

Another oft-used format is name-value pairs, as commonly seen in INI files. Like XML, this format is also easily read and edited:

```
[Data]

Driver=SQL Server

Database=Northwind

UserName=sa

Password=$%#&$%
```

This is also easily parsed: you can use STREXTRACT(Text, Name + "=", CHR(13)), AT(Name + "=", Text) followed by SUBSTR(), or ALINES() and ASCAN(). You can also use the GetPrivateProfileString Windows API function.

While the name-value pair format is less wordy than XML, it suffers from a few limitations. First, it can't handle multiple instances of a name. This forces you to append an instance number or find some other means to create unique names. For example, suppose you have a TreeView control on a form and want to store the keys for expanded nodes when the user closes the form so when they open it again, you can restore the state of those nodes. Using name-value pairs, you'd have to do something like this:

```
[Nodes]

Expanded1=SomeKeyValue

Expanded2=SomeOtherKeyValue

Expanded3=YetAnotherKey
```

This format also isn't hierarchical in nature; there are only two levels: the section level (the name in square brackets) and the name-value pairs level. This is fine for very simple information but quickly becomes cumbersome for more complex data.

Let's look at some of the uses I've made of XML over the past several years.

Field properties

The flagship product for my company, Stonefield Software, is Stonefield Query, an end-user reporting application. Although Stonefield Query generates an FRX file for a report, it doesn't do so until the report is actually run. Instead, the report is stored as a definition for what fields appear in the report, how they're formatted, and how the report itself is formatted. This gives us a lot of flexibility; for example, we can automatically size columns based on the width of the text in those columns rather than the field's defined width in the table, making the report more compact and easier to read.

Report definitions are stored as records in a single table. Since there is usually more than one field in a report, how is the formatting for multiple fields stored in a single record? In earlier versions, we used to store the field definitions in an array, with one row per field in the report and each column containing specific formatting information. For example, column 1 is the field name, column 2 is the data type, and other columns contain font name, font size, bold and italic settings, and so on. Saving from an array to a table and reading from a table to an array is pretty easy using the SAVE TO and RESTORE FROM commands to convert the array to and from a binary format stored in a memo field.

However, arrays are cumbersome to use: we always had to remember which column contains which type of information when we wrote code to process the fields in a report. Also, to support exporting a report definition to send to someone else, we had use a different mechanism to write the array values to a text file. The code to parse those text files was pretty ugly too.

A few years ago, we changed how field information is stored internally, in the reports table, and in export files. Internally, field information is now stored as objects in a collection. An SFField object has properties about a field in a report, such as cFieldName, cFontName, nFontSize, and lBold. These objects are stored in a collection of fields for a report. Now, instead of using code like this:

```
lnField = ascan(laFields, lcFieldName, -1, -1, 1, 15)

lcFontName = laFields[lnField, 12]
```

we can use this more readable version:

```
loField = This.oFieldsCollection.Item(lcFieldName)

lcFontName = loField.cFontName
```

We now store the field formatting to a memo field in the reports table using code that saves property values in the field objects to XML. Although this is more code than a single SAVE TO statement, it's very readable:

```
lcXML = '<fields>'

for each loField in This.oFieldsCollection

  lcXML = lcXML + '<field>' + ;
```

```
    '<fieldname>' + loField.cFieldName + '</fieldname>'
  lcXML = lcXML + '<fontname>' + loField.cFontName + '</fontname>'
  lcXML = lcXML + '<fontsize>' + transform(loField.nFontSize) + ;
    '</fontsize>'
* other properties handled here
  lcXML = lcXML + '</field>'
next loField
lcXML = lcXML + '</fields>'
replace FIELDS with lcXML in REPORTS
```

Populating the field collection from the table is also easily done. In this case, because the XML is simple, we decided to let XMLTOCURSOR() handle the parsing for us:

```
xmltocursor(REPORTS.FIELDS)
scan
  lcField = trim(FIELDNAME)
  loField = This.oFieldsCollection.Add(lcField)
  with loField
    .cFontName = trim(FONTNAME)
    .nFontSize = FONTSIZE
* other properties handled here
  endwith
endscan
```

We gained several benefits using this approach. First, the XML is much more readable and can easily be modified directly if necessary, something that was impossible to do with the binary array information. Second, we use the same code whether the field information is saved to a memo in a table or an export text file. Because we also store sorting, grouping, and filtering information this way now, we use nearly identical code to handle that information as well.

Although we don't currently have plans to do so, we could take it to the next step and store the entire report definition as XML. This would result in a more hierarchical layout, so we'd use the Microsoft XML DOM to parse it rather than XMLTOCURSOR(). However, in that case, we could eliminate most of the code handling export files, since the XML containing a report definition *is* the export file.

Memberdata

One of the ways you can extend the capabilities of the reporting system in VFP 9 is through memberdata for report objects. Memberdata defines additional properties for an object beyond those built into VFP. For example, Stonefield Query supports rotation of report objects (something that's now supported in VFP 9 Service Pack 2) and dynamic font and color (for example, display negative numbers in red and positive numbers in black). These attributes are stored as memberdata.

Memberdata is stored as XML in the STYLE memo of a report object in the FRX file. A ReportListener subclass called SFReportListener reads the memberdata for each FRX record and uses it to decide how to process the object. For example, this member data tells the report listener to instantiate another ReportListener subclass, SFReportListenerDirective defined in SFReportListener.VCX. That class will change the foreground color of the field to red (RGB value 255) if TOTAL_PRICE is less than zero or leave it at -1 (meaning the default color) if not. Note the less-than sign appears as "&lt;" in the XML as "<" is a special character in XML.

```
<memberdata>
  <VFPData><reportdata name="listener"
    type="forecolor"
    class="SFReportListenerDirective"
    classlib="SFReportListener.vcx"
    expression="iif(total_price &lt; 0, 255, -1)" />
  </VFPData>
</memberdata>
```

If you have read anything about memberdata, you'll note that several of the default attributes are missing (script, execute, execwhen, and declasslib) and there's a new one, expression. One of the nice things about XML is the "X," which stands for "extensible." As long as the schema supports it (and in the case of memberdata, there isn't an enforced schema), you can discard unnecessary attributes and add new attributes as you see fit. The VFP XMLTOCURSOR() function automatically handles XML with differing attributes between elements by creating columns for each possible attribute and leaving those that are omitted in a particular element blank in that record in the resulting cursor. That means a report can have memberdata created with extensions from different developers, which might each use different attributes, and the code that processes the memberdata won't have a problem parsing it.

Here's how SFReportListener process this XML. The BeforeReport event, which fires before a report is output, goes through each record in the FRX file and calls the ProcessFRXRecord method for each one. That method uses XMLTOCURSOR() on the XML stored in the STYLE memo (if it isn't blank) to create a memberdata cursor. In this example, there's only one record, but if several extensions are applied to a field, there could be several records. ProcessFRXRecord goes through each record in the memberdata cursor, calling ProcessMemberData for each one. Here's the code for ProcessFRXRecord.

```
local lnSelect, ;
```

```
    lcCursor, ;
    loException as Exception, ;
    loData
  if not empty(STYLE)
    lnSelect = select()
    lcCursor = sys(2015)
    try
      xmltocursor(STYLE, lcCursor, 4)
    catch to loException
    endtry
    if used(lcCursor)
      scan
        This.ProcessMemberData()
      endscan
      use
    endif used(lcCursor)
    select (lnSelect)
  endif not empty(STYLE)
```

ProcessMemberData checks whether the NAME field, which comes from the “name” attribute in the memberdata, is “listener.” If so, this memberdata specifies that another ReportListener subclass is to be used, so the code checks to see whether we’ve already instantiated the class specified in the CLASS field. You can store an object reference to another listener in the Successor property of a listener. As a result, you can form a chain of listeners: the first contains a reference to the second, the second to a third, and so on, until finally the Successor property of a listener contains .NULL. ProcessMemberData goes through the chain, checking whether the class of each listener is the desired class. If not, the code instantiates a new listener. Finally, ProcessMemberData calls the ProcessMemberData method of the specified listener, allow it to do something with the memberdata.

```
  if upper(NAME) = "LISTENER" and not empty(CLASS)
    lcClass     = upper(trim(CLASS))
    loParent    = This
    loSuccessor = This.Successor
    llHaveClass = .F.
    do while not isnull(loSuccessor)
      if upper(loSuccessor.Class) = lcClass
        llHaveClass = .T.
        exit
      endif upper(loSuccessor.Class) = lcClass
      loParent    = loSuccessor
      loSuccessor = loSuccessor.Successor
    enddo while not isnull(loSuccessor)
    if not llHaveClass
      lcLibrary = trim(CLASSLIB)
      loParent.Successor = newobject(lcClass, lcLibrary)
    endif not llHaveClass
    if pemstatus(loParent.Successor, 'ProcessMemberData', 5)
      loParent.Successor.ProcessMemberData()
    endif pemstatus(loParent.Successor...
  endif upper(NAME) = "LISTENER" ...
```

I won’t show the code for it, but the ProcessMemberData of SFReportListenerDirective, the class specified in the sample XML shown earlier, stores the values in the TYPE and EXPRESSION columns, which come from attributes in the memberdata with those names, in a collection, along with the record number of the object the memberdata is for in the FRX. Later, as the report is being run, SFReportListenerDirective checks whether it stored any information for the report object currently being rendered, and if so, makes any changes defined. In this example, the listener evaluates the saved expression and applies the resulting value to the foreground color for the object. The effect is that if TOTAL_PRICE is less than zero, the field appears in red.

How does the memberdata get into the STYLE memo of the FRX record in the first place? One way is to enter it in the properties dialog for the object. Double-click the object in the Report Designer to bring up the properties dialog, select the Other page, and click the Edit Settings button in the Run-time Extensions section of the page. This brings up a dialog with an editbox in which you can type the XML. Of course, that wouldn’t be very user-friendly, so we customized the properties dialog. These dialogs are VFP classes in ReportBuilder.APP. We subclassed the classes, added controls allowing the user to specify rotation and dynamic font and color styles, and created a new APP. We then set the _ReportBuilder system variable to the name of our replacement APP file.

APIs

We do a lot of work with the GoldMine customer relationship management (CRM) system. One of GoldMine's cool features is that you can communicate with the running application. For example, you can move to a particular record and display that record in the user interface. Early versions of GoldMine used DDE to send instructions to the application, but DDE is an old, clunky mechanism. Later versions added a COM interface that made it easier to do this. However, one thing that made the COM object awkward to deal with was name-value pairs.

Because the COM object methods are very flexible, they can accept a varying set of parameters. For example, you might want to move to a record by record number (numeric) or by company name (character). Also, sometimes some parameters are needed and other times not, depending on how you call the method. Although VFP applications can easily deal with varying data types for parameters (you simply use VARTYPE() and a CASE statement that deals with different data types) and optional parameters (you use PCOUNT() and use default values for unpassed parameters), this isn't allowed in COM. So, the GoldMine developers decided that rather than having a specific list of parameters, you'd pass a container of name-value pair objects. To support this, they added numerous methods for dealing with the container and the objects, such as create, copy, and delete methods. So now there's an entire set of methods to learn that have nothing to do with business logic functions but with how you *call* the business logic functions. Ugly, ugly, ugly.

Fortunately, they cleaned this all up in a newer release of GoldMine. Now, you simply pass XML to the methods. In fact, they even removed the myriad of business logic functions and replace them with a single one: ExecuteCommand. The XML you pass contains the name of the business logic function to execute and any parameters to pass to that function. ExecuteCommand returns XML, which contains not only an indicator of success or failure (and in the case of failure, an error code) but also any output values, of which there may be several.

For example, here's how you log into GoldMine so you can send it instructions:

```
loGoldMine  = createobject('GMW.UI')
loXML       = createobject('MSXML2.DOMDocument.3.0')
loXML.async = .F.
text to lcCommand noshow textmerge
<GMAPI call="Login">
<data name="User"><<lcUserName>></data>
<data name="Pass"><<lcPassword>></data>
</GMAPI>
endtext
lcReturn = loGoldMine.ExecuteCommand(lcCommand)
loXML.loadXML(lcReturn)
loNode = loXML.selectSingleNode('/GMAPI/status')
* status code -31703 means already running
* status code 1 means it worked

lcStatus = loNode.getAttribute('code')

llReturn = inlist(lcStatus, '1', '-31703')
```

Since a couple of object references are needed, and you have to login before you can do anything else, it seemed like a good idea to create a wrapper object for the GoldMine COM object. SFGoldMineAPI, contained in SFGoldMine.VCX included in this article's download, is based on Custom. Its Init method instantiates an MS XML DOM object into the oXML property which other methods will use to create and parse XML. The Connect method, which you must call before calling any other method and pass the user name and password, instantiates the GoldMine object into the oGoldMine property and calls the Login method, which has most of the code shown above.

Moving to a particular record in GoldMine is very similar to how you'd do it in VFP: select the desired table, save the current index order, change to the one you want to seek on, seek the desired value, and restore the former index order. SFGoldMineAPI has wrapper methods for each of these operations: SelectTable, GetOrder, SetOrder, and Seek. Here's the code for GetOrder, for example:

```
local lcCommand, ;

  lcReturn, ;

  loNode, ;

  lcOrder

text to lcCommand noshow

<GMAPI call="RecordObj">

<data name="Command">GetOrder</data>

</GMAPI>

endtext

lcReturn  = This.oGoldMine.ExecuteCommand(lcCommand)

This.oXML.loadXML(lcReturn)

loNode  = This.oXML.selectSingleNode('/GMAPI/status')

lcOrder = loNode.text

return lcOrder
```

The PositionByAccountNumber method moves to the company record with the specified account ID. It uses the other methods to do the actual work.

```
lparameters tcAccountNo

local lcOrder, ;
```

```
  llReturn
with This
  .cErrorMessage = ''
  .SelectTable('Primary')
  lcOrder = .GetOrder()
  .SetOrder('CONTACC')
  llReturn = .Seek(tcAccountNo)
  .SetOrder(lcOrder)
endwith
return llReturn
```

RSS

We like to inform our users about tips and tricks and when new versions of our software are available. However, emailing newsletters is becoming less and less attractive as time goes on because of spam filters and what I call “inbox fatigue.”

As you likely are aware, RSS has been taking the world by storm over the past several years. Millions of blogs and news sites use RSS to publish content daily that subscribers all over the world can read. I’ve been blogging for several years and we launched a blog specific for Stonefield Query about a year ago.

A feature we’re planning on adding to a future release of Stonefield Query is a built-in news reader. Why bother, given that there are many news readers already available? The problem we have is that many users of our software don’t know what a blog or news reader is, or aren’t aware that we have a blog. By adding a news reader directly into our software, users will automatically be informed about the latest tips and tricks and new releases. (Of course, we’ll provide a way they can turn it off if they don’t want to be bothered.)

As you may be aware, an RSS feed is just an XML file with a particular schema sitting on a Web server somewhere. So, providing your own custom news reader simply means downloading the file, parsing the XML, and displaying the content.

Downloading the file requires using some functions in WinINet.DLL. GetHTTPFile.PRG takes care of that. Simply pass it the URL for the Web site and the name of the file to retrieve. However, this program has no error handling, doesn’t support user names, passwords, proxies, or anything else you might need for some Web sites, so I recommend using West Wind Internet Protocols, a utility from Rick Strahl (www.west-wind.com), as a much more complete solution.

Parsing the XML is straight-forward. We’ll use the MS XML DOM object to do the heavy lifting and store information about each article in the RSS in a table called RSS.

```
loXML       = createobject('MSXML2.DOMDocument.3.0')
loXML.async = .F.
loXML.loadXML(lcXML)

loNodes = loXML.selectNodes('/feed/entry')

for each loNode in loNodes

  lcTitle      = loNode.selectSingleNode('title').text

  lcContent    = loNode.selectSingleNode('content').text

  lcPublished  = loNode.selectSingleNode('published').text

  ltPublished  = ctot(lcPublished)

  lcUpdated    = loNode.selectSingleNode('updated').text

  ltUpdated    = ctot(lcUpdated)

  loCategories = loNode.selectNodes('category')

  lcCategories = ''

  for each loCategory in loCategories

    lcCategories = lcCategories + iif(empty(lcCategories), '', ', 
') + ;

      loCategory.getAttribute('term')

  next loCategory

  insert into RSS values (lcTitle, ltPublished, ltUpdated, 
lcContent, ;

    lcCategories)

next loNode
```

RSSReader.SCX is a simple demo of an RSS reader. It has textboxes to display the title, date/time published, date/time updated, and categories. It also has an instance of the Microsoft Web Browser control to display the HTML content. There are Previous and Next buttons to move to another record in the RSS table.

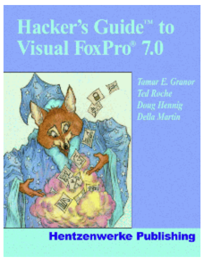
Summary

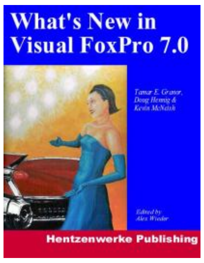
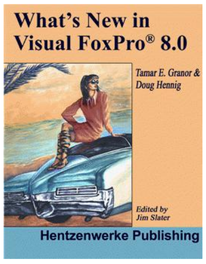
On the surface, XML looks like a simple technology. After all, it's just text. However, XML is a great solution for many types of problems, especially those involving data storage and data transmission. Take the time to investigate how XML can work for you.

Biography

Doug Hennig is a partner with Stonefield Systems Group Inc. and Stonefield Software Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT); the award-winning Stonefield Query;

the MemberData Editor, Anchor Editor, and CursorAdapter and DataEnvironment builders that come with Microsoft Visual FoxPro; and the My namespace and updated Upsizing Wizard in Sedna. Doug is co-author of the "What's New in Visual FoxPro" series, "The Hacker's Guide to Visual FoxPro 7.0," and the soon-to-be-released "Making Sense of Sedna and VFP 9 SP2." He was the technical editor of "The Hacker's Guide to Visual FoxPro 6.0" and "The Fundamentals." All of these books are from Hentzenwerke Publishing (http://www.hentzenwerke.com). Doug wrote over 100 articles in 10 years for FoxTalk and has written numerous articles in FoxPro Advisor and Advisor Guide. He currently writes for FoxRockX (http://www.foxrockx.com). He spoke at every Microsoft FoxPro Developers Conference (DevCon) since 1997 and at user groups and developer conferences all over the world. He is one of the organizers of the annual Southwest Fox conference (http://www.swfox.net). He is one of the administrators for the VFPX VFP community extensions Web site (http://www.codeplex.com/VFPX). He has been a Microsoft Most Valuable Professional (MVP) since 1996. Doug was awarded the 2006 FoxPro Community Lifetime Achievement Award (http://fox.wikis.com/wc.dll?Wiki~FoxProCommunityLifetimeAchievementAward).




What's New in Visual FoxPro 7.0
Tamar E. Granor, Doug Hennig & Kevin McNeish
Hentzenwerke Publishing

What's New in Visual FoxPro® 8.0
Tamar E. Granor & Doug Hennig
Edited by Jim Slater
Hentzenwerke Publishing