XML From RPG Using Free Tools

Session 500125
44CB

Presented by

Scott Klement

http://www.scottklement.com

"There are 10 types of people in the world.
Those who understand binary, and those who don't."

Why do I care about XML?

XML is important to businesses because:

- It is a standard for exchanging data between trading partners
- It is a standard for exchanging data between computer applications
- It can be processed by a program, but is readable to a human, which makes it versatile.
- It is one of the main technologies required for using Web Services
- Some newer APIs are using XML for input and output

How to do it in RPG?

While reviewing the syntax of XML in the next several slides, think to yourself…

- **"Could I write routines that output data in this format?"**
- **"If I had to write a program to read and process a file with this layout, how would I go about it?"**
- **"Could I make my read and process routines generic enough to be used for any XML document?"**

XML Syntax Review (1 of 3)

- **XML tags start with <TagName> and end with </TagName>**
- **They can have attributes   <TagName attribute="value">**
- **Tags can be nested in other tags   <tag1><tag2></tag2></tag1>**

```
<CustRec custno="1234">

  <Name>Fred's Pig Sprinklers, Inc.</Name>

  <Address>
    <Street>123 Main St.</Street>
    <City>Cleveland</City>
    <State>OH</State>
    <Zip>44145</Zip>
  </Address>

</CustRec>
```

XML Syntax Review (2 of 3)

Line breaks are not required, and can appear anywhere within the data, or can be omitted entirely. It's not always "one tag per line".

```
   <CustRec custno="4321">
   <Name>
      Bob's Shoe Emporium,Ltd.
   </Name>
   <Address>
     <Street>
       321 Sesame St.
     </Street>
     <City>New York</City><State>NY</State>   <Zip>12345</Zip>
   </Address></CustRec>
   <CustRec custno="5432"><Name>Big Al's
 Formula</Name><Address><Street>3067 W Thorncrest Dr</Street>
   .
   .
</CustFile>
```

XML Syntax Review (3 of 3)

- Tags can be intermixed with data. That data can span many lines.
- If a tag does not have a corresponding closing tag, it should be formatted as <TagName/>
- Tag names may not be unique throughout a document (for example, <title>)

```
 <Article>
   <author name="Scott Klement"/>
   <title>TCP/IP and Sockets</title>
   <chapter id="intro">
     <title>Chapter 1 - Introduction</title>
     <sect1>
       <title>TCP/IP Concepts and Terminology</title>
       <para>This section is an introduction to TCP/IP
         programming using a <I>Sockets API</I>.(Sockets can
         also be used to work with other network protocols,
          .
          .
       </para>
     </sect1>
  </chapter>
</Article>
```

What have you decided?

Can you write a program to output an XML document?

- **Output isn't very difficult because it doesn't usually change, and you control the format.**

Can you write a program to read and process one?

- **You probably could, but since you don't control how it's formatted when it's sent to you, it'd be difficult to ensure that the program would always work for every file.**

Can it be made generic enough to be re-used for any XML document?

- **It's probably possible, but it would be a lot of work! Why reinvent the wheel when there are free tools available?**

Writing XML w/Standard RPG

It's not usually that difficult to write XML output, since you usually have a specific format in mind. Since you control the format, rather than a 3rd party, you don't have to be ready to cope with any possible situation.

One way would be to write it to a flat file using standard RPG operations:

```
FXMLFILE   O    F  500        DISK
D xml             s            512A

 /free
        xml = '<?xml version="1.0">';
        except xmldata;;
        xml = '<CustFile>";
        except xmldata;
        xml = '  <CustRec>';
        except xmldata;
 /end-free

OXMLFILE   E            xmldata
O                       xml                 512
```

Writing XML w/IFS APIs

Output using RPG operations can be awkward. It's a somewhat easier using the IFS APIs because you can write many lines of XML and output them all at once.

```
      /free
        fd = open('/home/scottk/xml/test.xml'
                 : O_WRONLY+O_CREAT+O_TRUNC+O_CCSID
                 : M_RDWR
                 : 819);

        xml =
          '<?xml version="1.0">'
        + '<CustFile>'
        + '  <CustRec custno="' + %trim(CustNo) + '">'
        + '    <Name>' + %trim(Name) + '</Name>'
        + '    <Address>'
       ... And so forth ...

        callp write(fd: %addr(xml)+2: %len(xml));
        callp close(fd);
    /end-free
```

Writing XML w/Free Tools

Even with the IFS APIs, it can be awkward. You have to get all of the quotes in the right place, and maintenance can be awkward. One of the easiest ways is to use a free tool.

CGIDEV2 is a free tool from IBM. It's completely written in RPG and it's aimed at RPG programmers who want to output HTML code. Since XML is very similar to HTML, it works nicely for XML as well.

CGIDEV2 lets you create templates where you put the XML code. These templates can be written by you or by a colleague who is well versed in XML code and not so well versed in RPG code.

The templates consist of "sections", which are groups of XML code that you write out at once, and "variables", which are strings that are replaced with variable data from your RPG program.

Sample CGIDEV2 Template

Sections start with /$ (for example /$FileHeading)

Substitution variables are formatted: /%VarName%/

```
/$FileHeading
<?xml version="1.0"?>
<CustFile>

/$CustRec
  <CustRec custno="/%Custno%/">
     <name>/%Name%/</name>
     <Address>
       <Street>/%Street%/</Street>
       <City>/%City%/<City>
       <State>/%State%/</State>
       <Zip>/%ZipCode%/</Zip>
     </Address>
  </CustRec>

/$FileFooter
</CustFile>
```

Code for use with CGIDEV2

The key to writing XML with CGIDEV2 is to use the WrtHtmlToStmf() procedure to write the results. (Don't use `wrtsection('*fini')`)

```
getHtmlIfsMult( '/myTemplates/CustXml.tmpl' );

wrtsection('FileHeading');

setll *start custmas;
read custmas;

dow not %eof(custmas);
    updHtmlVar('Custno'     : %char(CustNo) );
    updHtmlVar('Name'       : Name    );
    updHtmlVar('Street'     : Addr    );
    updHtmlVar('City'       : City    );
    updHtmlVar('State'      : State   );
    updHtmlVar('ZipCode'    : %editw(ZipCode: '     -    '));
    wrtsection('CustRec');
    read custmas;
enddo;

wrtsection('FileFooter');

WrtHtmlToStmf('/export/custmas.xml': 819);
```

More CGIDEV2 notes

CGIDEV2 doesn't understand numbers. You'll need to use the `%EDITC, %EDITW` or `%CHAR` BIFs to convert them to strings before writing them to the file. (Easy to do!)

`updHtmlVar()` loads the variable values into CGIDEV2's memory.

`wrtsection()` merges the variable values into the XML code and then writes it to an output buffer in memory.

`WrtHtmlToStmf()` writes the output buffer to a stream file on disk.

Because CGIDEV2 works with everything in memory, it's not suitable for very large documents. The IFS API example does not have this problem, however.

There is no validation being done when the document is written. That's not usually a problem because the document layout doesn't change. Once you've completed the testing phase of your project, the document should be correct, so validating it each time may not be necessary.

Parsing XML

What are your options for parsing XML?

- Roll your own (good luck!)
- Buy IBM's XML Toolkit (DOM & SAX) (5733-XT1)
- Use the SAX parser built in to COBOL (V5R3) or RPG (V5R4)
- Use the free Java ones
- Call the Java methods directly from RPG
- Purchase a 3rd party tool.
- Use the open source Expat tool

What is Expat?

Expat is a service program that was written in C by James Clark who was the technical lead on the XML Working group of W3C when the XML specification was produced.

Expat is very fast, very reliable and robust.

It's the underlying XML parser for the *Mozilla* project, *OpenOffice, Perl's XML::Parser*, as well as Scott's own *HTTPAPI* project.

It's a stream-oriented parser, similar to SAX. You feed it an XML document, as a stream, and it will parse it to determine "events".

Scott has put together a package that you can download for free from his web site. It contains the Expat service program (precompiled), the source code, a CL program that compiles it, RPG prototypes in a /COPY member, and some sample RPG programs.

http://www.scottklement.com/expat/

Expat Concepts

When you use Expat:

- Your program is responsible for reading the file, not Expat!
- You read the file, and feed the data (bit by bit) into Expat's parser.

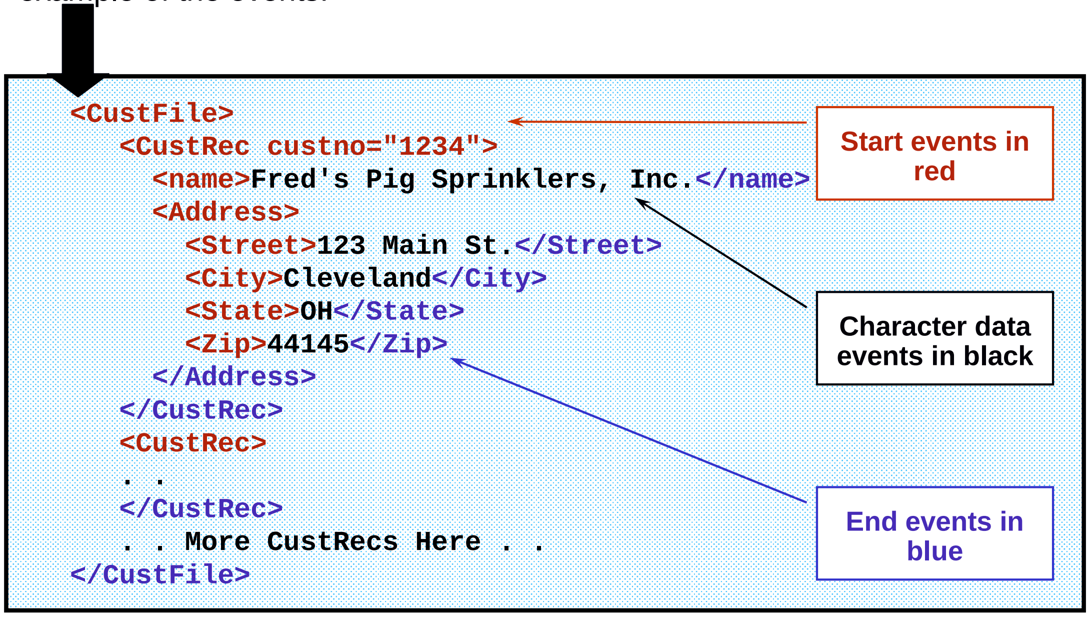
Tip: This means you can parse XML from any input source!

Events:

- Expat reads through the XML as your program feeds it in.
- It scans the data for "events".
  - ✓Start of XML element <MyTag>
  - ✓End of XML element </MyTag>
  - ✓Character data
- Each time an event is found, it calls one of your subprocedures.
- You write these subprocedures, and they process the events as required for your business goals.

Expat Scans For Events

Expat looks for start/end XML tags, these constitute "events". Here's an example of the events:



```
<CustFile>
   <CustRec custno="1234">
     <name>Fred's Pig Sprinklers, Inc.</name>
     <Address>
       <Street>123 Main St.</Street>
       <City>Cleveland</City>
       <State>OH</State>
       <Zip>44145</Zip>
     </Address>
   </CustRec>
   <CustRec>
   . .
   </CustRec>
   . . More CustRecs Here . .
</CustFile>
```

Expat Overview

Using Expat Requires These Steps:

- **Open the file that you want to parse**
- **Create a work space for the XML Parser**
- **Register *event handlers***
- **Feed the data to the XML parser (in a loop)**
- **Free up the work space**

Event Handler:

- **Subprocedures that you write**
- **Called by Expat when certain events occur**
  - ✓ Start of XML element <MyTag>
  - ✓ End of XML element </MyTag>
  - ✓ Character data
- **Data is passed in UCS2 Unicode.**
- **Must keep track of position in document**
- **Must decide what to do with the data that's read**

Example of Parsing Cust XML

The goal of this example is to parse the XML that we just wrote using CGIDEV2 and use it to print some address labels.

The XML data will be parsed and loaded into the following array:

```
D count           s              10I 0 inz(0)

D custrec         ds                   qualified
D                                      dim(100)
D   custno                        4P 0
D   name                         30A
D   street                       30A
D   city                         15A
D   state                         2A
D   zip                           9P 0 inz(0)
```

"Count" will contain the number of customer records found.

"custrec" will contain all of the addresses loaded from the XML document.

Opening the File

This example uses the IFS APIs, but you could also read it from a “normal” file if you like, or a variable, or a parameter... whatever is appropriate for your business needs.

```
fd = open( '/home/klemscot/custmas.xml'
         : O_RDONLY );

if (fd < 0);
   // open() failed. Handle error here.
endif;
```

Since Expat does not support EBCDIC (and EBCDIC would be an unusual character set for XML, in any case!), you'll want to make sure that the data in the file is in an ASCII or Unicode character set.

Some cases, it's smart to use O_TEXTDATA is used to convert the data to Unicode as it's read, that way i5/OS takes care of translating the character set instead of Expat.

More About Encodings

Expat only supports the following encodings:

- ISO-8859-1 ASCII (default)
- UTF-8 subset of Unicode
- UCS2 (UTF-16) subset of Unicode
- US-ASCII

If one of the above encodings was used, Expat can automatically determine the encoding from the "encoding" attribute of the <?xml> tag. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
```

No matter which encoding was used for the input data, Expat always sends you the output in UCS2 Unicode.

This helps simplify your event handlers! They only need to know how to handle UCS2, rather than every possible encoding!

Create a Work Space

Expat needs an internal work space to use when processing the document. If you're familiar with object-oriented programming, you can think of it as an "XML Parser Object". Every document needs it's own work space.

```
D XML_ParserCreate...
D                  PR                  ExtProc('XML_ParserCreate')
D                                      like(XML_Parser)
D    encoding                     *    value options(*string)
```

At this time, you can also tell Expat how the data is encoded (UTF-8), or specify *OMIT to let Expat figure it out from the document itself...

```
D p                s                   like(XML_Parser)
. . .
  p = XML_ParserCreate(XML_ENC_UTF8);
-- or --
  p = XML_ParserCreate(*OMIT);
  if (p = *NULL);
     callp close(fd);
     // unable to create work space, tell user!
  endif;
```

Register Event Handlers

This is how you tell Expat which subprocedures to call for each event.

```
// Call the "start" subprocedure for each XML Start Element Event:

XML_SetStartElementHandler( p : %paddr(start) );

// Call the "end" subprocedure for each XML End Element Event

XML_SetEndElementHandler( p : %paddr(end) );

// Call the "chardata" subprocedure for character data events

XML_SetCharacterDataHandler( p : %paddr(chardata) );
```

Subprocedures (like everything else) are just bytes in memory. You can get their address in memory with the **`%paddr()`** BIF. Expat will call a subprocedure at the address you provide.

Feed the Parser

The data from the XML document must be fed to the XML_Parse() API, one chunk at a time. The API will scan the data for XML tags and will call the event handlers as needed. Pass done=1 to indicate that there’s no more data.

```
dou (done = 1);
       len = read(fd: %addr(Buff): %size(Buff));

       if (len < 1);
          done = 1;
       endif;

       if (XML_Parse(p: Buff: len: done) = XML_STATUS_ERROR);
           callp close(fd);
           errorMsg = 'Parse error at line '
              + %char(XML_GetCurrentLineNumber(p)) + ': '
              + %str(XML_ErrorString(XML_GetErrorCode(p)));
           // Here, you'll show the error message to the
           // user, or write to QSYSOPR, etc...
       endif;

enddo;
```

This is where you input the XML data, and it will call your handlers.

Free Up the Work Space

Now that the parsing is done, clean up the objects that were used to do the parsing:

- Free up the Expat work space
- Close the stream file

```
// Free up Expat work space
XML_ParserFree(p);

// close stream file
callp close(fd);
```

Tracking Position

The way I keep track of where I am in the document is by implementing a stack. Each time the start element handler is called, it pushes a new item onto the stack. Each time the end element handler is called, it pops the top item off of the stack.

```
<CustFile>
   <CustRec custno="1234">
     <name>Fred's Pig Sprinklers, Inc.</name>
     <Address>
       <Street>123 Main St.</Street>
       <City>Cleveland
        . . . and so on . . .
```

/CustFile/CustRec/Address/City

/CustFile/CustRec/Address

/CustFile/CustRec

/CustFile

Adding to a Stack in RPG

Implementing the stack in code means:

- Keep track of the current depth
- Maintain a global array that contains the XPath at the given depth
- Maintain a global array that contains the value of the element at the given depth

At the top of my program, in the global D-specs, I have the following:

```
D depth           s              10I 0
D stack           s             512A   varying dim(32)
D stackval        s             512A   varying dim(32)
```

In the start element handler, I add each new element to the stack:

```
depth = depth + 1;
stackval(depth) = '';

if (depth = 1);
   stack(depth) = '/' + %char(elemName);
else;
   stack(depth) = stack(depth-1) + '/' + %char(elemName);
endif;
```

Tip: The ***`%CHAR()`*** *BIF converts the UCS2 data to EBCDIC.*

Removing from the Stack

I remove each element from the stack when the end element handler is called, simply by reducing the current depth.

```
depth = depth - 1;
```

Since the stack is global, I can check it from anywhere in my handlers. Any time I need to know where I am in the document, I can check the current stack entry as follows:

```
if ( stack(depth) = '/CustFile/CustRec' );
   // do something that should be done for the CustRec element.
endif;
```

Start Element Handler (1 of 3)

This handler is called by XML_Parse() for the start elements.

A start element event handler must accept parameters in the following format, but the subprocedure name can have any name you like:

```
D start           PR
D   userdata                         *    value
D   elem                        16383C    options(*varsize)
D   attr                             *    dim(32767)
D                                         options(*varsize)
```

Userdata = It's possible to supply data to Expat that it will pass back to all of your handlers. I did not use it in this example, so it will be *NULL.

Elem = The element name in UCS2 Unicode (RPG data type C). The length can vary -- you'll know the end of the data by searching for a "null" (a hex u'0000' character)

Attr = Array of UCS2 strings, all are null-terminated. They represent the XML attributes specified on the start tag. A null pointer signals the end of the attribute list.

Start Element Handler (2 of 3)

Usually, the Start Element Handler has two important goals:
- To keep track of the position in the document (create an Xpath)
- To parse the attributes

Before it can do those things, it needs to find the end of the string by searching for the u'0000' character. I like to use **%subst()** to move it to a VARYING field for future use.

```
P start           B
D start           PI
D   userdata                     *    value
D   elem                    16383C    options(*varsize)
D   attr                         *    dim(32767)
D                                     options(*varsize)

D elemName        s            500C   varying

/free
    // Copy everything before the u'0000' (null terminator)
    // to the elemName variable:
    len = %scan(u'0000': elem) - 1;
    elemName = %subst(elem:1:len);
```

Start Element Handler (3 of 3)

After translating to EBCDIC, the start element handler adds the new element to the stack (as discussed earlier) and then checks to see if this is a new CustRec element. If so, it'll start a new array element.

```
// ----------------------------------------------------------
// Maintain stack
// ----------------------------------------------------------
depth = depth + 1;
stackval(depth) = '';

if (depth = 1);
   stack(depth) = '/' + %char(elemName);
else;
   stack(depth) = stack(depth-1) + '/' + %char(elemName);
endif;

// ----------------------------------------------------------
// If this is the start of a new customer record,
//  move to the next element of the customer record array
// ----------------------------------------------------------
if ( stack(depth) = '/CustFile/CustRec' );
    count = count + 1;
endif;
```

Parsing Attributes (1 of 2)

Attributes are passed to the start element handler as an array of pointers. The pointers point to C-style null-terminated strings.

The elements of the array are in the UCS2 encoding.

The array alternates between the attribute name and it's value.

The first one is an attribute name. The next is it's value. Then comes the next attribute name, and so on, until one of the pointers is *NULL. A *NULL denotes the end of the list.

Parsing Attributes (2 of 2)

```
D attrData        s          16383C   based(p_Attr)
D attrName        s            100C   varying
D attrVal         s            200C   varying

    x = 1;
    dow attr(x) <> *NULL;

       p_Attr = attr(x);
       len = %scan(U'0000': attrData) - 1;
       attrName = %subst(attrdata:1:len);

       p_Attr = attr(x+1);
       len = %scan(U'0000': attrData) - 1;
       attrVal = %subst(attrData:1:len);

       if (stack(depth) = '/CustFile/CustRec' )
            and attrName = %ucs2('custno');
           custrec(count).custno = %dec(%char(AttrVal):4:0);
       endif;

       x = x + 2;
    enddo;
```

```
<CustRec custno="1234">
```

Character Data Handler (1 of 2)

The character data handler will be called for any data that's not part of an XML tag (the black data).

Your character data handler must accept the following parameters:

```
D chardata        PR
D   UserData                      *   value
D   string                   16383C   const options(*varsize)
D   len                         10I 0 value
```

UserData = same as the one of the start and end element handlers.

String = The character data in UCS2 Unicode. Unlike previous examples, u'0000' does not signal the end, instead you must check the *len* parameter.

Len = length of the string parameter, in characters (not bytes!)

Character Data Handler (2 of 2)

```
P chardata        B
D chardata        PI
D   UserData                     *   value
D   string                  16383C   const options(*varsize)
D   len                        10I 0 value

 /free
    stackval(depth) = stackval(depth)
                    + %char(%subst(string:1:len));
 /end-free
P                 E
```

Tip: The length of the data will never be longer than the buffer you used when you fed the data into Expat. Use a 16383 or smaller buffer to ensure that the *len* parameter isn't too long for the procedure to handle.

In this example, the data is only saved to the stack. The end element handler map it into the data structure array, that way it's clear that the entire element has been received.

End Element Handler (1 of 2)

When the end element handler is called, you know that there's no more data for this element. Therefore, you do the following in the end element handler:

- Save the "stackval" value to the array of customer addresses, depending on the current position in the document.
- Pop the top element from the stack.

The end element handler always has the following parameters:

```
D end             PR
D   UserData                     *   value
D   elem                    16383C   const options(*varsize)
```

UserData = Same as the one for the start element handler.

Elem = element name in UCS2 Uncode. It's terminated by u'0000' like the element name from the start element handler. (We don't need this, though, since the element name is on our stack!)

End Element Handler (2 of 2)

Map the elements into their places in the array.

```
select;
when stack(depth) = '/CustFile/CustRec/name';
    custrec(count).Name = stackval(depth);
when stack(depth) = '/CustFile/CustRec/Address/Street';
    custrec(count).Street = stackval(depth);
when stack(depth) = '/CustFile/CustRec/Address/City';
    custrec(count).City = stackval(depth);
when stack(depth) = '/CustFile/CustRec/Address/State';
    custrec(count).State = stackval(depth);
when stack(depth) = '/CustFile/CustRec/Address/Zip';
    if %scan('-':stackval(depth)) <> 0;
         stackval(depth) = %replace( ''
                                   : stackval(depth)
                                   : %scan('-': stackval(depth))
                                   : 1  );
    endif;
    custrec(count).Zip = %dec( stackval(depth) : 9 : 0 );
endsl;

depth = depth - 1;
```

Conclusion of Program

Since the Element Handlers are all called during the call to XML_Parse(), the parsing is complete once the work space has been cleared.

Now you can use the data in the CustRec array. In this case, it's used to print some address labels:

```
D pr              ds                   likeds(CustRec)
  .
  .
  for x = 1 to Count;
     pr = CustRec(x);
     except Label;
  endfor;
  .
  .
OQSYSPRT   E             Label          1  3
O                        pr.Name
O          E             Label          1
O                        pr.Street
O          E             Label          1
O                        pr.City
O                        pr.State            +1
O                        pr.Zip              +2 '     -    '
```

Built-In XML Support

COBOL XML Support

- Starting in V5R3, COBOL on the System i has built-in support for XML, similar to what Expat provides.
- Like Expat, it's a non-validating SAX-style parser.
- In V5R4, they added support for writing XML as well. (but, it very limited.)

RPG XML Support

- In V5R4, support similar to that of COBOL is included with ILE RPG.
- XML-INTO op-code loads XML from a string or file into variables.
- XML-SAX op-code calls a callback procedure much like Expat does.

The XML-INTO op-code makes things very easy, but doesn't work with name spaces, and requires a data structure to match the XML layout.

XML-SAX is very similar to the Expat solution, maybe a bit more complex.

More Information – CGIDEV2

CGIDEV2 is supported by IBM. Thedownload page for CGIDEV2 is
http://www-03.ibm.com/systems/services/labservices/library.html

Tutorials on Web programming with CGIDEV2 are available at:
http://www.easy400.net

Scott has written several articles about CGIDEV2 for his newsletter:

- **CGIDEV2 for XML**
  http://www.systeminetwork.com/article.cfm?id=51276

- **Web programming in RPG parts 1,2,3**
  http://www.systeminetwork.com/article.cfm?id=51135
  http://www.systeminetwork.com/article.cfm?id=51145
  http://www.systeminetwork.com/article.cfm?id=51209

- **CGIDEV2 for E-mail**
  http://www.systeminetwork.com/article.cfm?id=51238

More Information – Expat

Download Expat and sample code (including the CustList example) from Scott's Web site:

http://www.scottklement.com/expat/

XML Resources

- **Scott has written more about Expat in his newsletter.**
- **Latest "getting started" article:**
  http://www.systeminetwork.com/article.cfm?id=53061
- **Name space support:**
  http://systeminetwork.com/article/xml-namespaces-and-expat
- **Older Articles (from when Expat was returning UTF-8):**
  http://systeminetwork.com/article/club-tech-iseries-programming-tips-newsletter-23
  http://systeminetwork.com/article/club-tech-iseries-programming-tips-newsletter-25
  http://systeminetwork.com/article/club-tech-iseries-programming-tips-newsletter-26
  http://www.systeminetwork.com/article.cfm?id=50719

Expat's main site (no System i or RPG information):

http://www.libexpat.org

More Info – RPG XML Opcodes

I have written the following articles about RPG's opcodes:

XML-INTO:

http://systeminetwork.com/article/real-world-example-xml
http://systeminetwork.com/article/xml-maximum-length
http://systeminetwork.com/article/xml-read-xml-data-larger-65535
http://systeminetwork.com/article/xml-output-array-larger-16-mb

PTFs for IBM 6.1, offer Extended Support in XML-INTO:
by Barbara Morris of IBM
http://www-949.ibm.com/software/rational/cafe/docs/DOC-2975

XML-SAX:

http://www.scottklement.com/rpg/xml-sax

This Presentation

You can download a PDF copy of this presentation from:

http://www.scottklement.com/presentations/

Thank you!