User Experience: Writing a web-enabled CICS/COBOL program
SHARE
August 2010

Craig Schneiderwent
State of Wisconsin
Department of Transportation
cschneidpublic@yahoo.com
#include <std/disclaimer.h>

Any opinions expressed are my own and do not necessarily reflect those of my employer. You wouldn't believe the amount of red tape that one statement gets me out of.

What You Will See

- Web-enabled CICS/COBOL code
- Other cool, webby stuff
  - COBOL to generate XML
  - COBOL to parse XML
- The COBOL is just snippets, an entire working application is available on the SHARE website with the materials for this session

This presentation grew out of a proof of concept application I wrote to help out our development staff.

The proof of concept application receives parameters and based on those will send HTML, XML, an HTML form, or an image from CICS back to the requesting web browser.

Main entry point in the application is J7200544.

Environment (at the Time)

- z/OS 1.6, CICS TS 2.2, DB2 7.1, Enterprise COBOL 3.4
- WebSphere Application Server (WAS) 5.x on Windows
- 20 years worth of CASE tool generated business logic in CICS
- CASE tool runtime uses CICS COMMAREAs to pass data out of CICS

Since then we've been migrated to WAS 6, z/OS 1.10, DB2 v8, and CICS TS 3.1. The CASE tool has gone through many name changes and several owners, it's currently called CA-Gen and is owned by CA.

These last two bullets were a real driving force behind the project.

What was required

Java application running in WAS on Windows needed > 32K of data in XML from CASE tool business logic

Given the capabilities of CICS and COBOL you're about to see, the client could as easily have been C/C++, C#, Perl, Ruby, etc. running on Linux, Unix, what have you. The server (CICS/COBOL) isn't sensitive to client programming language or operating environment.

The developers felt themselves between a rock and a hard place. Note the CASE tool uses commareas, thus the 32K problem.

This second point is pretty important. The techniques presented are standards-based and thus can be used with any client that conforms to the standards.

What we considered, rejected, and eventually decided on

- Access data directly from Java application
  - Would duplicate existing business logic
  - Not SOA-ish and we'd like to head in an SOA direction
- MQ
  - Additional piece of infrastructure
- Raw sockets
  - Would require us to invent a protocol
- HTTPS
  - Didn't require an additional piece of infrastructure
  - Is itself a protocol running over sockets

There was a <u>lot</u> of discussion on how to proceed. My personal vote was for MQ. With different people involved, or in a different IT shop, the decision might go differently.

Plugging my own work, file 788 on the CBTTape site (formerly known as the MA1K MQ SupportPac) would have made this trivial.

Proof of concept

- Applications staff wanted a working example
- CICS support staff knew it could be done, but this would be the first of its kind in our shop
- Code would have to not just work, but also serve as a guide

Just pointing the developers at the documentation and saying "Go forth and do good works" was not a viable option. They write code to implement business logic, this was outside of their problem space.

0001 Success

JA05-J7200544

- **Region:** DCICS
- **Method:** GET
- **Version:** HTTP/1.1
- **Path:** /WisDOT/HostSystemInformation
- **Query String:** t=html
- **Authentication:** BASICAUTH
- **SSL Type:** SSL
- **Client Address:** 131.48.35.23
- **Client Name:** dot.wi.gov
- **TCPIPSERVICE:** HTTPSSL
- **Server Address:** 166.188.54.152
- **Server Name:** dserverdot.wi.gov
- **Server Port:** 08164
- **User ID:** CWS1
- **User Name:** SCHNEIDERWENT, CRAIG
- **GMT:** Mon, 29 Jun 2009 13:24:04 GMT

The proof of concept application, prompted with a query string of "t=html" (without the quotes) returns HTML that renders something like this. The application also responds to query strings of "t=xml" and "t=imag", sending back XML and a graphic respectively.

Where to start?

- CICS manuals
  - Programming Guide
  - Programming Reference
  - Internet Guide
- RFCs <http://www.rfc-editor.org/>
- COBOL Manual
  - Programming Reference
  - Programming Guide
- Experimentation

The second bullet points to a searchable list of RFCs.

The URL for every RFC I mention in this presentation is on the last page, so don't worry if you miss one.

I found experimentation to be very important.

HTTPS in One Slide

- HTTP over SSL
- HTTP is defined in RFC 2616
- Plain text headers describing the message
- You can define your own headers
- Specifies "methods" or "verbs"
- SSL is Secured Sockets Layer
- Traffic over the socket is encrypted
- External to the application

The HTTP specification includes headers describing the message. You can retrieve these headers with a CICS API. Headers include "Accept-Charset", "Authorization", and "Content-type".

You can also retrieve which "method" or "verb" was used by the requester. Methods include GET, POST, PUT, DELETE, and several others.

For purposes of this project, I was interested in two of those request types: GET and POST.

You can see how these methods or verbs might mesh nicely with the logic of a service.

How does the code get invoked? (application programmer view)

An HTTP request arrives. The URL takes the form

scheme://hostName:port/path?queryString Where

scheme is http or https

hostName is something meaningful assigned by your network staff corresponding to your IP address

port is the port your TCPIPSERVICE definition specifies

path is something meaningful to identify your application to its users

queryString is optional parameters for your program

If you're the application programmer, you really only need to know that your code will be invoked when an HTTP request arrives. CICS will start your program and you can retrieve information about your invocation parameters via CICS APIs which we'll see shortly.

In our case the URL is something like https://cicsdotp.wi.gov:3090/DMVApps/driverInquiry.

Some of our apps use queryString, others do not.

Application Programmer View (contd.)

- querySthing may contain program parameters, but it is optional
- If the HTTP method for this request is a POST, you may also get some data in addition to/instead of the parameters

If there is data in the queryString, you are responsible for parsing it. It may take the form

fieldName1=value1+fieldName2=value2

or

value1&value2

There isn't a standard for this, it's up to you. Negotiate with the person writing the client code and come up with something you can both live with. Likewise with the POST data, perhaps you can pass XML between the client and the CICS server program, but you can just as easily do comma-delimited quotes-around-strings data streams.

How does the code get invoked? (system programmer view)

- TCPIPSERVICE definition in CICS specifies IP address and port (a socket)
- URIMAP definition specifies the path the incoming request will use in its URI and the tranID and program to be invoked
- Being the application programmer, I didn't have to worry about this stuff

Testing

Since I was writing a server program, I needed a client with which to test it.

The application is standards-based (HTTP, HTML, XML) so I could write the client in any language that understood those standards.

But there was no need for me to write a client, any current web browser also understands those standards.

CICS APIs (representative but non-comprehensive)

- EXTRACT TCPIP
- WEB EXTRACT
- WEB READ HTTPHEADER
- WEB RECEIVE
- DOCUMENT CREATE
- DOCUMENT INSERT
- WEB SEND

These are the CICS APIs mentioned earlier. There are a lot more than this, but this is what I used.

EXTRACT TCPIP is used to get information about the TCPIP "environment" such as the client IP address, client DNS name, your IP address, your DNS name, the port you're communicating on, whether SSL is in effect or not, and so forth. I found this information useful in creating meaningful error messages.

WEB EXTRACT is used to get the URL you were invoked with, and most importantly the query string - those parameters that come after the question mark.

WEB READ HTTPHEADER is used to get specific information from the HTTP headers. You ask for a particular header by name and you get the value or a "not found" condition is raised. HTTP headers are listed in RFC 4229, along with pointers to the RFC where each header is defined.

WEB RECEIVE is used to get any data that may have been sent if you were invoked via an HTTP POST. This is analogous to doing a RETRIEVE if you were invoked via an EXEC CICS START.

This is pretty much the order in which you would use these APIs, and in fact this is a basic outline of a CICS HTTP server application. Get some information about your environment, get your calling parameters, do your business logic (probably between WEB RECEIVE and DOCUMENT CREATE), create your document, put your reply in the document and send it back to the requester.

Enough of this nonsense, let's look at the code

- Much of this is RFCs coded in COBOL
- Personal philosophy: If you *must* abend, don't *just* abend
- I prefer to catch bad EIBRESPs
- Some EIBRESPs are only "sorta" bad, you really must catch them
- Yes, it's more code, tricky code to get right, but your poor coworker carrying the pager will thank you

As the first bullet indicates, much of what's in this proof-of-concept application is RFCs implemented in COBOL. This really was a matter of reading what is required of an HTTP server application and writing COBOL to do those things.

Much of the rest of this is preparatory to your seeing the source code and wondering why I didn't just code NOHANDLE.

Some people don't code RESP or NOHANDLE on their EXEC CICS calls, allowing CICS to abend their application if something goes wrong. I think this is a fine thing to do under certain circumstances. An example of when I don't think it's fine is in the WEB READ HTTPHEADER API, where a "not found" condition is raised if the specified header is not present. You're going to see that the HTTP request might specify a client code page, and it might not. The RFC states that the server application must honor the code page if it is specified. So a "not found" condition is only "sorta" bad. You really must catch this one.

With a 3270 application, when something goes wrong you can display an error message and the end user can call a help desk to describe what they were doing and what the error message is. We're talking about a server application here. If something goes wrong, you don't have a good place to display that error message. So I decided to use the WRITE OPERATOR API followed by the ABEND API with a unique user abend code within the application.

None of this is based on rules or principles, just on painful experience.

```
        EVALUATE WS-RESP
            WHEN DFHRESP( NORMAL )
                CONTINUE
            WHEN OTHER
                INITIALIZE CICS-API-FAILED
                MOVE 'WEB-READ-HTTPHEADER' TO CICS-API-FAILED
                MOVE '8070'            TO CICS-API-FAILED-LOC
                PERFORM 8900-GET-CICS-RESP-MNEMONIC
                PERFORM 8930-MAKE-HTTP-HDR-ERR-SUFX
        END-EVALUATE
        EVALUATE WS-RESP
            WHEN DFHRESP( NORMAL )
                CONTINUE
            WHEN DFHRESP( NOTFND )
                IF WS-RESP2 = 1
*                   Requested header was not found
                    SET HTTP-HDR-NOT-FND TO TRUE
                ELSE
                    PERFORM 8910-MAKE-CICS-ERR-MSG
                    PERFORM 9100-WTO
                END-IF
            WHEN OTHER
                PERFORM 8910-MAKE-CICS-ERR-MSG
                PERFORM 9100-WTO
        END-EVALUATE
```

What you will see in the proof of concept code is similar to this. For each EXEC CICS do-something call, there is a check for each RESP and/or RESP2 combination that is only "sorta" bad and an attempt is made to continue. For RESPs that are just plain bad, an error message is constructed and written to the system log. This is usually followed by a user abend. Hopefully that message written to the system log contains enough information to debug the problem without diving into the core dump.

```
01  WORK-AREAS.
    05  WS-RESP                PIC S9(008) COMP-5 VALUE +0.
    05  WS-RESP2               PIC S9(008) COMP-5 VALUE +0.
    05  WS-AUTH-CVDA           PIC S9(008) COMP-5 VALUE +0.
    05  WS-CLNT-NM-LN          PIC S9(008) COMP-5 VALUE +0.
    05  WS-CLNT-ADDR-LN        PIC S9(008) COMP-5 VALUE +0.
    05  WS-SRVR-NM-LN          PIC S9(008) COMP-5 VALUE +0.
    05  WS-SRVR-ADDR-LN        PIC S9(008) COMP-5 VALUE +0.
    05  WS-SSL-TY-CVDA         PIC S9(008) COMP-5 VALUE +0.
    05  WS-CLNT-NM             PIC X(080) VALUE SPACES.
    05  WS-CLNT-ADDR           PIC X(015) VALUE SPACES.
    05  WS-SRVR-NM             PIC X(080) VALUE SPACES.
    05  WS-SRVR-ADDR           PIC X(015) VALUE SPACES.
    05  WS-TCPIP-SRVC-NM       PIC X(008) VALUE SPACES.
    05  WS-PORT-NB             PIC X(005) VALUE SPACES.

    MOVE LENGTH OF WS-CLNT-NM   TO WS-CLNT-NM-LN
    MOVE LENGTH OF WS-CLNT-ADDR TO WS-CLNT-ADDR-LN
    MOVE LENGTH OF WS-SRVR-NM   TO WS-SRVR-NM-LN
    MOVE LENGTH OF WS-SRVR-ADDR TO WS-SRVR-ADDR-LN

    EXEC CICS
        EXTRACT TCPIP
        AUTHENTICATE(WS-AUTH-CVDA)
        CLIENTNAME(WS-CLNT-NM)
        CNAMELENGTH(WS-CLNT-NM-LN)
        CLIENTADDR(WS-CLNT-ADDR)
        CADDRLENGTH(WS-CLNT-ADDR-LN)
        SERVERNAME(WS-SRVR-NM)
        SNAMELENGTH(WS-SRVR-NM-LN)
        SERVERADDR(WS-SRVR-ADDR)
        SADDRLENGTH(WS-SRVR-ADDR-LN)
        SSLTYPE(WS-SSL-TY-CVDA)
        TCPIPSERVICE(WS-TCPIP-SRVC-NM)
        PORTNUMBER(WS-PORT-NB)
        RESP(WS-RESP)
        RESP2(WS-RESP2)
    END-EXEC
```

I do suggest that, if your application is known to only execute via an SSL connection, check for that. If you find SSL is not in effect, spit out an error message to the log and respond with some HTML or XML indicating the problem. Depending on your situation this might indicate an application under development is attempting to make use of your server application, or it could mean someone is trying to break into your system.

If you're not familiar with CVDAs (CICS Value Data Areas), they're explained in the CICS Application Programming Reference. They're just numeric values that have a specific meaning, much like EIBRESP.

```
01  WORK-AREAS.
    05  WS-RESP               PIC S9(008) COMP-5 VALUE +0.
    05  WS-RESP2              PIC S9(008) COMP-5 VALUE +0.
    05  WS-HTTP-QRY-STRN.
        10                    PIC X(050) OCCURS 10.


 8040-GET-HTTP-QRY-STRN.
*
* Obtain the http query string - the bit that comes after
* the question mark.
*
    MOVE LENGTH OF WS-HTTP-QRY-STRN TO WS-HTTP-QRY-STRN-LN
    EXEC CICS
        WEB EXTRACT
        QUERYSTRING(WS-HTTP-QRY-STRN)
        QUERYSTRLEN(WS-HTTP-QRY-STRN-LN)
        RESP(WS-RESP)
        RESP2(WS-RESP2)
    END-EXEC
```

This is how you get your query string, containing your invocation parameters. Note that the length of the receiving field is part of the API. You've no doubt heard of "buffer overflows" or "stack smashing" in reference to hacking incidents. CICS won't let your data overflow the target field so long as you set the length correctly. This is a nice safety feature.

You could get fairly elaborate in your error handling, adding a specific error message to be returned if you get a LENGERR because the parameters were too verbose.

WEB EXTRACT does more than just get your query string, I chose to code extracting the path and http method as separate calls.

```
 01   CONSTANTS.
      05   HTTP-CHARSET-HDR       PIC X(014)
           VALUE 'Accept-Charset'.
 01   WORK-AREAS.
      05   WS-HTTP-HDR-TO-RTV-LN  PIC S9(008) COMP-5 VALUE +0.
      05   WS-HTTP-HDR-BUFR-LN    PIC S9(008) COMP-5 VALUE +0.
      05   WS-HTTP-HDR-TO-RTV     PIC X(050) VALUE SPACES.
      05   WS-HTTP-HDR-BUFR.
           10                     PIC X(050) OCCURS 5.
      05   WS-HTTP-CLNT-CHARSET   PIC X(040) VALUE SPACES.


 1040-GET-CLNT-CD-PG.
*     We must specify the client's code page when we send
*     a response.  The code page is present in the http
*     protocol header, so we will retrieve it from there.
*
      INITIALIZE
          WS-HTTP-HDR-BUFR
          WS-HTTP-HDR-TO-RTV
      MOVE HTTP-CHARSET-HDR TO WS-HTTP-HDR-TO-RTV
      MOVE LENGTH OF HTTP-CHARSET-HDR TO WS-HTTP-HDR-TO-RTV-LN
      MOVE LENGTH OF WS-HTTP-HDR-BUFR TO WS-HTTP-HDR-BUFR-LN
      PERFORM 8070-WEB-READ-HDR
      IF HTTP-HDR-NOT-FND
*         Since there doesn't appear to be a client code page
*         in the protocol header, we'll use an innocuous default.
*         This default comes from section 3.7.1 "Canonicalization
*         and Text Defaults" of RFC 2616 "Hypertext Transfer
*         Protocol -- HTTP/1.1"
          INITIALIZE WS-HTTP-CLNT-CHARSET
          MOVE 'ISO-8859-1' TO WS-HTTP-CLNT-CHARSET
      ELSE
          PERFORM 8080-UNSTRN-CHARSET
      END-IF
      .
```

Notice how the http header whose value we want to read gets moved in, along with its length and the length of the buffer we want to read that value into. CICS is again trying to save us from buffer overflows.

As you can see, if the header is not found, meaning a client code page was not provided in the request, we use a harmless default which comes from RFC 2616.

```
01  WORK-AREAS.
    05  WS-RESP                PIC S9(008) COMP-5 VALUE +0.
    05  WS-RESP2               PIC S9(008) COMP-5 VALUE +0.
    05  WS-HTTP-HDR-TO-RTV-LN  PIC S9(008) COMP-5 VALUE +0.
    05  WS-HTTP-HDR-BUFR-LN    PIC S9(008) COMP-5 VALUE +0.
    05  WS-HTTP-HDR-TO-RTV     PIC X(050) VALUE SPACES.
    05  WS-HTTP-HDR-BUFR.
        10                     PIC X(050) OCCURS 5.


8070-WEB-READ-HDR.
    INITIALIZE HTTP-HDR-NOT-FND-SW
    EXEC CICS
        WEB READ
        HTTPHEADER(WS-HTTP-HDR-TO-RTV)
        NAMELENGTH(WS-HTTP-HDR-TO-RTV-LN)
        VALUE(WS-HTTP-HDR-BUFR)
        VALUELENGTH(WS-HTTP-HDR-BUFR-LN)
        RESP(WS-RESP)
        RESP2(WS-RESP2)
    END-EXEC
    INITIALIZE HAVE-ERR-MSG-SUFX-SW
    EVALUATE WS-RESP
        WHEN DFHRESP( NORMAL )
            CONTINUE
        WHEN OTHER
            INITIALIZE CICS-API-FAILED
            MOVE 'WEB-READ-HTTPHEADER' TO CICS-API-FAILED
            MOVE '8070'               TO CICS-API-FAILED-LOC
            PERFORM 8900-GET-CICS-RESP-MNEMONIC
            PERFORM 8930-MAKE-HTTP-HDR-ERR-SUFX
    END-EVALUATE
    EVALUATE WS-RESP
        WHEN DFHRESP( NORMAL )
            CONTINUE
        WHEN DFHRESP( NOTFND )
            IF WS-RESP2 = 1
*               Requested header was not found
                SET HTTP-HDR-NOT-FND TO TRUE
            ELSE
                PERFORM 8910-MAKE-CICS-ERR-MSG
                PERFORM 9100-WTO
            END-IF
        WHEN OTHER
            PERFORM 8910-MAKE-CICS-ERR-MSG
            PERFORM 9100-WTO
    END-EVALUATE
```

This is one of those situations where WS-RESP not being equal to NORMAL is only "sorta" bad.

If WS-RESP = NOTFND and WS-RESP2 = 1 it means the specified header, in this case the client code page, is not present.

```cobol
 01  WORK-AREAS.
     05  WS-RESP                 PIC S9(008) COMP-5 VALUE +0.
     05  WS-RESP2                PIC S9(008) COMP-5 VALUE +0.
     05  WS-DOC-TOKN             PIC X(016) VALUE SPACES.

 8050-DOC-CRTE.
*
* Create the document into which we will insert the http
* protocol header(s) and html/text/xml that constitute
* the reply to be sent to the requestor.
*
* If you take a look at 8090-WEB-SEND, you'll see that we use
* the WS-DOC-TOKN to indicate which document we want to send
* as a reply.
*
     EXEC CICS
         DOCUMENT CREATE
         DOCTOKEN(WS-DOC-TOKN)
         RESP(WS-RESP)
         RESP2(WS-RESP2)
     END-EXEC
```

We have to create a document in which to place the reply we're sending back to the requester. The document token acts much like a file handle, it's just a way for CICS to identify which document you want to work with.

```
01  CONSTANTS.
    05  HOST-CD-PG              PIC X(008) VALUE '037     '.
01  WORK-AREAS.
    05  WS-RESP                 PIC S9(008) COMP-5 VALUE +0.
    05  WS-RESP2                PIC S9(008) COMP-5 VALUE +0.
    05  WS-RPLY-BUFR-LN         PIC S9(008) COMP-5 VALUE +0.
    05  WS-DOC-TOKN             PIC X(016) VALUE SPACES.
01  WS-RPLY-BUFR.
    05  PIC X(050) OCCURS 40960.

8060-DOC-ISRT.
*
* Insert the content of WS-RPLY-BUFR into the reply document.
*
* This example application only does one insert per reply.  The
* CICS documentation indicates one can do multiple inserts, with
* each being added to the "bottom" of the document.  One can
* also create "bookmarks" within the document and insert at a
* bookmark location.
*
    IF BINARY-CONTENT
        EXEC CICS
            DOCUMENT INSERT
            DOCTOKEN(WS-DOC-TOKN)
            BINARY(WS-RPLY-BUFR)
            LENGTH(WS-RPLY-BUFR-LN)
            RESP(WS-RESP)
            RESP2(WS-RESP2)
        END-EXEC
    ELSE
        EXEC CICS
            DOCUMENT INSERT
            DOCTOKEN(WS-DOC-TOKN)
            TEXT(WS-RPLY-BUFR)
            LENGTH(WS-RPLY-BUFR-LN)
            HOSTCODEPAGE(HOST-CD-PG)
            RESP(WS-RESP)
            RESP2(WS-RESP2)
        END-EXEC
    END-IF
```

And this is how the reply gets into the document. You can get quite fancy with these documents, creating bookmarks and document templates. Unfortunately I can't speak to such fanciness, what you're looking at is the extent of my experience.

Notice the specification of the HOSTCODEPAGE parameter on this API. This is how CICS knows from what code page to convert this document to the requester's client code page.

In my shop we use code page 37, yours might be different.

Also note that the host code page isn't specified when inserting binary content. This is content we don't want to have code page conversion performed on, like maybe a graphic.

The code on the SHARE website includes logic to place a graphic which resides in Working-Storage into the reply document.

```
*      SHARE LOGO
       05  PIC X(100)
           VALUE           X'FFD8FFE000104A464946000102000006400640000
-                          'FFEC00114475636B79000100040000003C0000FF
-                          'EE002641646F62650064C0000000010300150403
-                          '060A0D000004BB0000079C00000B1C00000F18FF
-                          'DB00840006040404050406050506090605060906090B'.
       05  PIC X(100)
           VALUE           X'080606080B0C0A0A0B0A0A0C100C0C0C0C0C0C10
-                          '0C0E0F100F0E0C1313141413131C1B1B1B1C1F1F
-                          '1F1F1F1F1F1F1F1F010707070D0C0D181010181A
-                          '1511151A1F1F1F1F1F1F1F1F1F1F1F1F1F1F1F1F
-                          '1F1F1F1F1F1F1F1F1F1F1F1F1F1F1F1F1F1F1F1F'.
       05  PIC X(100)
           VALUE           X'1F1F1F1F1F1F1F1F1F1F1F1F1F1FFFC2001108007B
-                          '007D03011100021101031101FFC400D400010001
-                          '050100000000000000000000000030204050607
-                          '010101010101010100000000000000000000001
-                          '0203040510000104010304010304030000000000'.
       05  PIC X(100)
           VALUE           X'000300010204052011121030130614405015702 1
-                          '33353132341100010204020507 0A050500000000
-                          '000001110200211203310410417122132 05161A1
-                          '325223308191B1D1E142627214 50C1B23373F182
-                          '92632412000105000000000000000000000000000'.
       05  PIC X(100)
           VALUE           X'40507080112113010002020103030402 03010100
-                          '0000000100112131411051613071A1208191C1F0
-                          'B14050D17060FFDA000C0301000 2110311000001
-                          'EA800000044608D70B837D0000010B38ED73C46B
-                          '96999EDA867B460DD93A116C6CA002CAF3D4FAF9'.
```

A graphic looks like this in Working-Storage. There's nothing special about the size of the PICTURE clause, other than the 160 byte limitation imposed by COBOL.

This does go on for a bit, the whole copybook is 234 lines long.

```
01  HTML-RPLY.
    05                    PIC X(006)  VALUE '<html>'.
    05                    PIC X(002)  VALUE X'0D25'.
    05                    PIC X(017)  VALUE '<h1 align=center>'.
    05                    PIC X(002)  VALUE X'0D25'.
    05  RPLY-STUS-CD      PIC X(050)  VALUE SPACES.
    05                    PIC X(002)  VALUE X'0D25'.
    05                    PIC X(005)  VALUE '</h1>'.
    05                    PIC X(017)  VALUE '<h2 align=center>'.
    05                    PIC X(002)  VALUE X'0D25'.
    05  RPLY-TRANID       PIC X(004)  VALUE SPACES.
    05                    PIC X(001)  VALUE '-'.
    05  RPLY-PGM          PIC X(008)  VALUE SPACES.
    05                    PIC X(002)  VALUE X'0D25'.
    05                    PIC X(005)  VALUE '</h2>'.
    05                    PIC X(002)  VALUE X'0D25'.
    05                    PIC X(004)  VALUE '<br>'.
    05                    PIC X(002)  VALUE X'0D25'.
    05                    PIC X(006)  VALUE '<body>'.
    05                    PIC X(002)  VALUE X'0D25'.
    05                    PIC X(004)  VALUE '<ul>'.
    05                    PIC X(002)  VALUE X'0D25'.
    05                    PIC X(004)  VALUE '<li>'.
    05                    PIC X(004)  VALUE '<b> '.
    05                    PIC X(008)  VALUE 'Region: '.
    05                    PIC X(004)  VALUE '</b>'.
    05  RPLY-APPL-ID      PIC X(008)  VALUE SPACES.
```

HTML looks just like HTML. The x'0D25' literals translate into the familiar CRLF sequence, for readability if you need to look at the raw HTML on the receiving end.

```
05                     PIC X(006)  VALUE '<body>'.
05                     PIC X(006)  VALUE '<form '.
05                     PIC X(007)  VALUE 'action='.
05   FORM-RPLY-URL     PIC X(053)  VALUE SPACES.
05                     PIC X(014)  VALUE 'method="POST" '.
05                     PIC X(013)  VALUE 'name="form1">'.
05                     PIC X(002)  VALUE X'0D25'.
05                     PIC X(003)  VALUE '<b>'.
05                     PIC X(012)  VALUE 'Reply Type: '.
05                     PIC X(004)  VALUE '</b>'.
05                     PIC X(002)  VALUE X'0D25'.
05                     PIC X(007)  VALUE '<input '.
05                     PIC X(012)  VALUE 'type="text" '.
05                     PIC X(009)  VALUE 'size="4" '.
05                     PIC X(012)  VALUE 'name="type" '.
05                     PIC X(014)  VALUE 'maxlength="4">'.
05                     PIC X(002)  VALUE X'0D25'.
05                     PIC X(004)  VALUE '<br>'.
```

You can send HTML forms if you want, and again this is just HTML coded as constants in Working-Storage surrounding the data to be sent. This is just a snippet from the proof of concept application, you can see where there's a text field defined.

```
 01  WORK-AREAS.
     05  WS-RESP              PIC S9(008) COMP-5 VALUE +0.
     05  WS-RESP2             PIC S9(008) COMP-5 VALUE +0.
     05  WS-DOC-TOKN          PIC X(016) VALUE SPACES.
     05  WS-HTTP-CLNT-CHARSET PIC X(040) VALUE SPACES.

 8090-WEB-SEND.
*
* Send the reply constructed in the document referenced by
* WS-DOC-TOKN back to the requester.
*
     EXEC CICS
         WEB SEND
         DOCTOKEN(WS-DOC-TOKN)
         CLNTCODEPAGE(WS-HTTP-CLNT-CHARSET)
         RESP(WS-RESP)
         RESP2(WS-RESP2)
     END-EXEC
```

The client code page is specified here on the WEB SEND API. So you have two pieces, the host code page on the DOCUMENT INSERT and the client code page on the WEB SEND. There's your "from" and "to" for the conversion.

What about security?

- SSL
- BASICAUTH

SSL you're probably familiar with as a user. Secured Sockets Layer encrypts all traffic between client and server. When you're shopping online you probably see a little "lock" icon appear somewhere in your browser's window indicating an SSL conversation is taking place.

SSL can be set up with RACF, or I presume any of the other security products out there.

BASICAUTH is kind of interesting. A server can require that the client include an Authentication header with its HTTP request. It does this by responding to HTTP requests without such a header with a 401 status code. HTTP status codes are specified in RFC 2616, 401 means "Unauthorized." The usual response from a client, say a web browser, is to pop up a modal dialog box asking for a logon ID and password. The client then creates an Authentication header per RFC 2617 and includes it in its request which it resends to the server. The server validates the logon ID and password and proceeds accordingly.

This works, BASICAUTH can be set on the TCPIPSERVICE definition by your CICS sysprog.

This is all completely outside the CICS application. No application server logic need be coded to do any of this.

Content of Communication

- We decided to send XML back and forth between client and server
- XML has been around for some years now and seems to have gained an avid following on the distributed platforms

Creating XML in COBOL, the code

```
XML GENERATE
        WS-BUFFER FROM FancyOutput
        COUNT IN WS-BUFFER-SIZE
        ON EXCEPTION PERFORM 9010-BAD-XML-GEN
END-XML
```

It turns out the code is well-nigh trivial, thanks to our friends in COBOL development.

Creating XML in COBOL, Working-Storage

```
01  WORK-AREAS.
    05  WS-BUFFER-SIZE PIC S9(008) COMP-5 VALUE +0.
    05  FancyOutput.
        10  NbOfItems PIC 9(002) DISPLAY VALUE 0.
        10  ItemList
            OCCURS 0 TO 99 DEPENDING NbOfitems.
            15  ItemID PIC X(008).
            15  ItemQuantity PIC 9(004).
            15  ItemPrice PIC 9(004).99.
            15  ItemDescription PIC X(080).
        10  CustomerID PIC X(008).
        10  CustomerAddressLine OCCURS 2 PIC X(040).
        10  CustomerCity PIC X(020).
        10  CustomerState PIC X(002).
        10  CustomerZip PIC X(005).
        10  OrderTotal PIC 9(008).99.
01  WS-BUFFER PIC X(102400) VALUE SPACES.
```

The variable names from Working-Storage are used to construct the XML "begin" and "end" tags. Mixed case is not required, but the people who spec this stuff out seem to like it, so here it is for illustration.

Creating XML in COBOL, What you get

```
<FancyOutput><NbOfItems>3</NbOfItems><ItemList><It
emID>ABCDEFGH</ItemID><ItemQuantity>1</ItemQuantit
y><ItemPrice>0001.23</ItemPrice><ItemDescription>D
escription 01</ItemDescription></ItemList><ItemLis
t><ItemID>BCDEFGHI</ItemID><ItemQuantity>2</ItemQu
antity><ItemPrice>0001.99</ItemPrice><ItemDescript
ion>Description 02</ItemDescription></ItemList><It
emList><ItemID>CDEFGHIJ</ItemID><ItemQuantity>6</I
temQuantity><ItemPrice>0003.50</ItemPrice><ItemDes
cription>Description 03</ItemDescription></ItemLis
t><CustomerID>CUST0001</CustomerID><CustomerAddres
sLine>123 Main Street</CustomerAddressLine><Custom
erAddressLine> </CustomerAddressLine><CustomerCity
>AnyTown</CustomerCity><CustomerState>WI</Customer
State><CustomerZip>53700</CustomerZip><OrderTotal>
00000026.21</OrderTotal></FancyOutput>
```

And, after an XML GENERATE, you get something that looks like this.

These are not separate variables, this is one long stream of data.
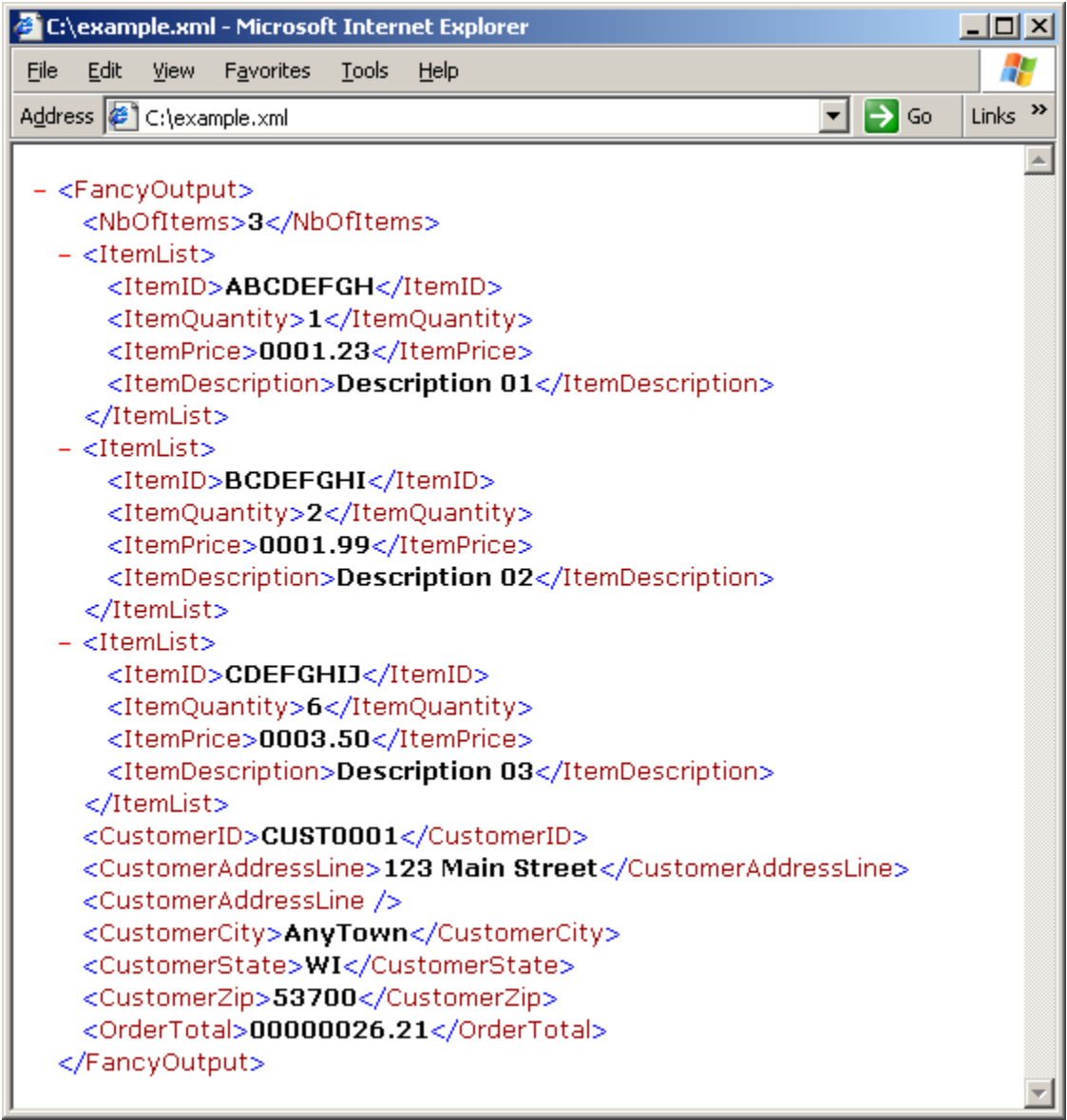
Thankfully, web browsers know all about XML and will format it nicely for display.

You can see how there's a nice match between COBOL's hierarchical structures in Working-Storage and XML's hierarchical organization of data.

All I did was cut the data stream out of a core dump and paste it into a file, then open the file with MS Internet Explorer.

Creating XML in COBOL, What you get

- Trailing spaces trimmed from character data elements
- Leading zeroes trimmed from integer numeric (apparently not from anything including a decimal point)
- XML tags are the same as the variable names from Working-Storage
- For Occurs Depending On (ODO) you only get as many as you specify in the object of the ODO
- For standard OCCURS you get as many as you specify, and they could be empty
- Again, thanks to our friends in COBOL development, you get pretty much what you'd expect

ODO Gotchas

- Anything following an ODO clause cannot be the subject of an INITIALIZE statement (compile error)
- The object of an ODO cannot itself be variably located (compile error)
- If you change the ODO object, the "template" for the subject and subsequent data items moves *but the data doesn't*
  - Runtime error if you're lucky, data corruption if you're not
  - I wasn't lucky
- Suggest making the ODO object part of the structure for which you're generating XML
- If there's a COBOL program receiving your XML they will thank you
- We had a Java program receiving the XML and they didn't want it so the ODO object got moved outside the structure

If you use the INITIALIZE statement, be advised you can't use it on an item containing an ODO clause nor can you use it on anything following an item containing an ODO clause within a given "01" level. This is a compile error.

The object of an ODO is the item specified in the DEPENDING phrase. If you say "A OCCURS 1 TO 26 DEPENDING Z" then Z cannot be variably located.

This third bullet, this is important. It's kind of like changing the register for a USING in Assembler. Unless you're real careful, nothing good is going to come after that.

The fourth bullet, if you put all your ODO objects at the beginning of the structure you're generating XML for, you have a nice neat data stream for a COBOL program to XML PARSE.

This won't work

```
01  BadArea.
    05  NbOfIncidents PIC 9(002).
    05  IncidentList
        OCCURS 0 TO 10 DEPENDING NbOfIncidents.
        10  IncidentType PIC X(008).
        10  NbIncidentReasons PIC 9(002).
        10  IncidentReason
            OCCURS 1 TO 10 DEPENDING NbIncidentReasons PIC X(008).
```

This has nothing to do with XML GENERATE, the COBOL language just doesn't support a structure of this type

See how the object of the IncidentReason item ODO is variably located? You might ask, "Well, who needs to do that?" Much to our surprise and dismay, we did.

The workaround

```
01  Incident.
    05  NbOfIncidents    PIC 9(002).
    05  NbIncidentReasons PIC 9(002).
    05  IncidentList.
        10  IncidentType  PIC X(008).
        10  IncidentReason
            OCCURS 1 TO 10 DEPENDING NbIncidentReasons PIC X(008).
01  WS-INTERMEDIATE-BUFFER PIC X(10240).
01  WS-FINAL-BUFFER PIC X(102400).
77  WS-XML-LEN PIC 9(008) COMP-5.
77  WS-BUFFER-PTR PIC 9(008) COMP-5 VALUE 1.

    PERFORM NbOfIncidents TIMES
        move data to Incident structure
        XML GENERATE WS-INTERMEDIATE-BUFFER FROM Incident
            COUNT IN WS-XML-LEN
        END-XML
        STRING WS-INTERMEDIATE-BUFFER(1:WS-XML-LEN)
          INTO WS-FINAL-BUFFER
          POINTER WS-BUFFER-PTR
        END-STRING
    END-PERFORM
```

So we did this. Multiple XML GENERATEs into an intermediate buffer which is then STRINGed into the final buffer.

This won't work either

```
05  MyRecord01.
    10  Key01   PIC X(014) VALUE 'AEIOU123456789'.
    10  Value01.
        20  PIC X(001) VALUE '&'.
        20  PIC X(001) VALUE '<'.
        20  PIC X(001) VALUE '>'.
        20  PIC X(001) VALUE "'".
        20  PIC X(001) VALUE '"'.
    10  Value02 PIC X(008) VALUE '<MNOPQR>'.

XML GENERATE FINAL-XMLIZED-DATA
    FROM MyRecord01
    COUNT IN NB-XML-BYTES
    ON EXCEPTION PERFORM 9010-XML-GEN-ERR
END-XML

Results in

01 FINAL-XMLIZED-DATA
                   AN-GR
 02 FILLER            X(50)  OCCURS 20
    SUB(1)            DISP              '<MyRecord01><Key01>AEIOU123456789</Key01><Value02>'
    SUB(2)                             '&lt;MNOPQR&gt;</Value02></MyRecord01>              '
    SUB(3)                             '                                                  '
    SUB(4) to SUB(20) elements same as above.
```

This XML GENERATE in the middle is intended to give me all the fields in the MyRecord01 structure at the top. Notice how Value01 is missing from the results at the bottom? The documentation for XML GENERATE reads "unnamed elementary data items or elementary FILLER data items" will be ignored.

Those results at the bottom are from a call to CEE3DMP so I could examine Working-Storage contents.

Performance

- The more data elements in the structure being generated, the more CPU time it takes
- We decided to check on using STRING to create their XML to see if it performed better
- Wrote a benchmark program with different data scenarios using IBM Enterprise COBOL version 3
- Bottom line: STRING always outperforms XML GENERATE but *STRING doesn't do everything for you* that XML GENERATE does

Given the workaround a couple of slides back, people started looking at the CPU time for the application and wondering about using STRING instead of XML GENERATE.

It turns out these aren't really comparable, XML GENERATE does lots more for you than STRING.

STRING doesn't

- Trim *trailing* spaces
  - DELIMITED SPACES clips at the first space
- Do anything special with hex data values
  - XML GENERATE prefixes the tag name with "hex." if the data element contains hex data
  - XML GENERATE puts a character representation of the hex data in the target (0040, if the data element contains a hex null followed by a space)
- Automatically adjust for a change in the data structure for which XML is being created
- Do anything special with & < > ' " characters
  - XML GENERATE converts these to & < > ' " respectively
  - See http://www.w3.org/TR/2006/REC-xml11-20060816/#syntax

In fact, XML GENERATE does so much more for you I needed a whole extra slide to tell you about it.

That URL at the end points to the most recent XML 1.1 specification, part of which states that you're supposed to do this conversion.

STRING is faster, but at the price of significantly more maintenance.

What if I'm sent XML?

```
       1000-XML-PARSE.
           XML PARSE FINAL-XMLIZED-DATA
               PROCESSING PROCEDURE 2000-PARSE-PRCS
               ON EXCEPTION PERFORM 9030-XML-PARSE-ERR
           END-XML
           .
       2000-PARSE-PRCS.
       *    XML-EVENT is a COBOL special register, populated (in this
       *    case) by the XML parser.
           EVALUATE TRUE
             WHEN LENGTH OF XML-TEXT > 0
               DISPLAY
                 MYNAME SPACE XML-EVENT
                 ': XML-TEXT = |' XML-TEXT '|'
             WHEN LENGTH OF XML-NTEXT > 0
               DISPLAY
                 MYNAME SPACE XML-EVENT
                 ': XML-NTEXT = |' XML-NTEXT '|'
           END-EVALUATE

           EVALUATE XML-EVENT
             WHEN 'START-OF-ELEMENT'
                 MOVE XML-TEXT TO CURR-XML-TAG
                 MOVE 1 TO STRING-PTR
             WHEN 'CONTENT-CHARACTER'
             WHEN 'CONTENT-CHARACTERS'
                 PERFORM 2100-MOVE-TO-FLD
             WHEN 'EXCEPTION'
                 PERFORM 9020-XML-PARSE-ERR
             WHEN OTHER
                 CONTINUE
           END-EVALUATE
           .
```

The code in that first EVALUATE is useful for debugging purposes, or just gaining some insight into how the parser works with your code.

The real work begins in the second EVALUATE. The parser interacts with your code via Special Registers, variables that are implicitly defined. These special registers are XML-CODE, XML-EVENT, XML-TEXT, and XML-NTEXT. There is a control flow section in the Language Reference that describes this interaction.

Basically, when an event occurs, the code associated with the PROCESSING PROCEDURE phrase is executed. Either XML-TEXT or XML-NTEXT is set to the content in the XML document associated with the event. XML-NTEXT is for USAGE NATIONAL data. There are a whole bunch of XML-EVENTs defined in the COBOL Programming Language Reference in the section on Special Registers.

The START-OF-ELEMENT event occurs when, unsurprisingly, we get to a begin tag for an element. Notice that XML-TEXT is set to the name of the element.

Please take special note of the two checks for CONTENT-CHARACTER (singular) and CONTENT-CHARACTERS (plural). From examining the output generated from that first EVALUATE, it appears that the first event occurs when the parser encounters one of those special cases I mentioned previously, the ones that STRING doesn't handle. Ampersands and less-thans and greater-thans, oh my. The second event, CONTENT-CHARACTERS, appears to occur when a string of "normal" characters is encountered.

Also note that the above paragraph describes the behavior of Enterprise COBOL 3.4. Enterprise COBOL 4.1 has a new compile option, XMLPARSE. This option can take one of two values, COMPAT or XMLSS. The behavior described in the above paragraph also holds true for XMLPARSE(COMPAT). It does not hold true for XMLPARSE(XMLSS). The difference is that the "CONTENT-CHARACTER" event will never occur with XMLPARSE(XMLSS).

```
   05  MyRecord01.
       10  Key01   PIC X(014) VALUE 'AEIOU123456789'.
       10  Value01.
           20  PIC X(001) VALUE '&'.
           20  PIC X(001) VALUE '<'.
           20  PIC X(001) VALUE '>'.
           20  PIC X(001) VALUE "'".
           20  PIC X(001) VALUE '"'.
       10  Value02 PIC X(008) VALUE '<MNOPQR>'.
   05  MyRecord02.
       10  Key01   PIC X(014) VALUE 'ZXYWV123456789'.
       10  Value01 PIC X(005). *Value01 of MyRecord01 is MOVEd into Value01 of MyRecord02
       10  Value02 PIC X(008) VALUE '<MNOPQR>'.
   05  MyRecord03.
       10  Key01   PIC X(014).
       10  Value01 PIC X(005).
       10  Value02 PIC X(008).
   05  STRING-PTR              PIC 9(008)  COMP-5 VALUE 0.
   05  CURR-XML-TAG           PIC X(030) VALUE SPACES.

   2100-MOVE-TO-FLD.
       EVALUATE CURR-XML-TAG
           WHEN 'Value01'
               STRING
                 XML-TEXT DELIMITED SIZE
                 INTO Value01 OF MyRecord03
                 POINTER STRING-PTR
                 OVERFLOW PERFORM 9040-STRING-OVERFLOW
               END-STRING
           WHEN 'Value02'
               STRING
                 XML-TEXT DELIMITED SIZE
                 INTO Value02 OF MyRecord03
                 POINTER STRING-PTR
                 OVERFLOW PERFORM 9040-STRING-OVERFLOW
               END-STRING
           WHEN OTHER
               CONTINUE
       END-EVALUATE
        .
```

In order to deal with the possibility that this code is processing one or more of those special characters, we have to keep track of where we are. At any time, the value in XML-TEXT may be one byte or many bytes and we may traverse this code more than once for a given value of CURR-XML-TAG.

I decided to handle this with a STRING statement using the POINTER phrase to keep track of my current position. I've also seen this done with reference modification.

Note that the value in CURR-XML-TAG was set to whatever was in XML-TEXT when we got the START-OF-ELEMENT event. In this case, the name of my Working-Storage field is the same as the name in CURR-XML-TAG, but it wouldn't have to be. That's just how I wrote this code.

Notice that Value01 of MyRecord02 consists of all special characters, and Value02 of MyRecord02 consists of a mix of special and ordinary characters.

If the XML document to be parsed contains the following

```
<MyRecord02><Key01>ZXYWV123456789</Key01><Value01>
&amp;&lt;&gt;&apos;&quot;</Value01><Value02>&lt;MN
OPQR&gt;</Value02></MyRecord02>
```

This is what we get from that debugging code a couple of slides back

```
XMLTEST4 START-OF-DOCUMENT              : XML-TEXT = |<MyRecord02><Key01>ZXYWV123
456789</Key01><Value01>&amp;&lt;&gt;&apos;&quot;</Value01><Value02>&lt;MNOPQR&gt
;</Value02></MyRecord02>

                                                  |

XMLTEST4 START-OF-ELEMENT              : XML-TEXT = |MyRecord02|
XMLTEST4 START-OF-ELEMENT              : XML-TEXT = |Key01|
XMLTEST4 CONTENT-CHARACTERS            : XML-TEXT = |ZXYWV123456789|
XMLTEST4 END-OF-ELEMENT                : XML-TEXT = |Key01|
XMLTEST4 START-OF-ELEMENT              : XML-TEXT = |Value01|
XMLTEST4 CONTENT-CHARACTER             : XML-TEXT = |&|
XMLTEST4 CONTENT-CHARACTER             : XML-TEXT = |<|
XMLTEST4 CONTENT-CHARACTER             : XML-TEXT = |>|
XMLTEST4 CONTENT-CHARACTER             : XML-TEXT = |'|
XMLTEST4 CONTENT-CHARACTER             : XML-TEXT = |"|
XMLTEST4 END-OF-ELEMENT                : XML-TEXT = |Value01|
XMLTEST4 START-OF-ELEMENT              : XML-TEXT = |Value02|
XMLTEST4 CONTENT-CHARACTER             : XML-TEXT = |<|
XMLTEST4 CONTENT-CHARACTERS            : XML-TEXT = |MNOPQR|
XMLTEST4 CONTENT-CHARACTER             : XML-TEXT = |>|
XMLTEST4 END-OF-ELEMENT                : XML-TEXT = |Value02|
XMLTEST4 END-OF-ELEMENT                : XML-TEXT = |MyRecord02|
```

The parser starts up and lets us know that via the START-OF-DOCUMENT event with XML-TEXT set to the contents of the document. Please note that there are some of those special characters included in the content.

Then we get a START-OF-ELEMENT event with XML-TEXT set to the name of the element, MyRecord02.

Then we get another START-OF-ELEMENT event with XML-TEXT set to the name of the element, Key01.

Then we get a CONTENT-CHARACTERS event with XML-TEXT set to at least some of the characters comprising the value of Key01. And then an END-OF-ELEMENT event with XML-TEXT set to the element name.

Now we get a START-OF-ELEMENT event with XML-TEXT set to Value01, the name of the element. This is followed by five CONTENT-CHARACTER events, each with XML-TEXT set to one of the special characters comprising the value of the element Value01. Followed by the END-OF-ELEMENT event for Value01.

Then we have our last START-OF-ELEMENT event, this one is for Value02. We get a CONTENT-CHARACTER event for the first byte, which is another one of those pesky special characters. Then we get a CONTENT-CHARACTERS event (note the plural, this is a different event) for the "normal" characters, followed by another CONTENT-CHARACTER (singular) event for the last byte which is again special. You can see why you can't just MOVE the XML-TEXT to your receiving field, sometimes you get the value of an element in separate chunks.

We finish up with two END-OF-ELEMENT events, one for Value02 and the last one for MyRecord02.

Try doing **that** with UNSTRING!

What You Have Seen

- CICS APIs to allow your COBOL code to be invoked and send a reply via HTTP(S)
- Generating XML in COBOL
- Parsing XML in COBOL

Questions?

Some RFCs of note

Hypertext Transfer Protocol -- HTTP/1.0
http://www.ietf.org/rfc/rfc1945.txt?number=1945

Hypertext Transfer Protocol -- HTTP/1.1
http://www.ietf.org/rfc/rfc2616.txt?number=2616

HTTP Header Field Registrations
http://www.ietf.org/rfc/rfc4229.txt?number=4229

HTTP Authentication: Basic and Digest Access
Authentication
http://www.ietf.org/rfc/rfc2617.txt?number=2617