Web Services from RPG using HTTPAPI

Session 500124 **33MA**

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 Web Services?

A web service provides the ability to call a program (or procedure) over the Web.

- How is this different from the Web applications I see all the time?
- Couldn't I just automate those?
- What good is a Web service?

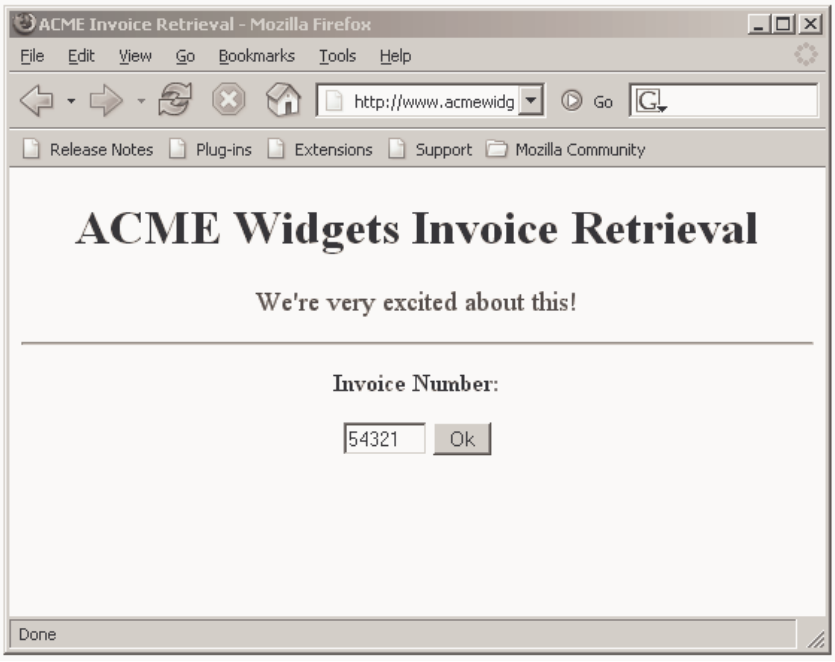
I'll answer these questions, but first some background . . .

Web Applications

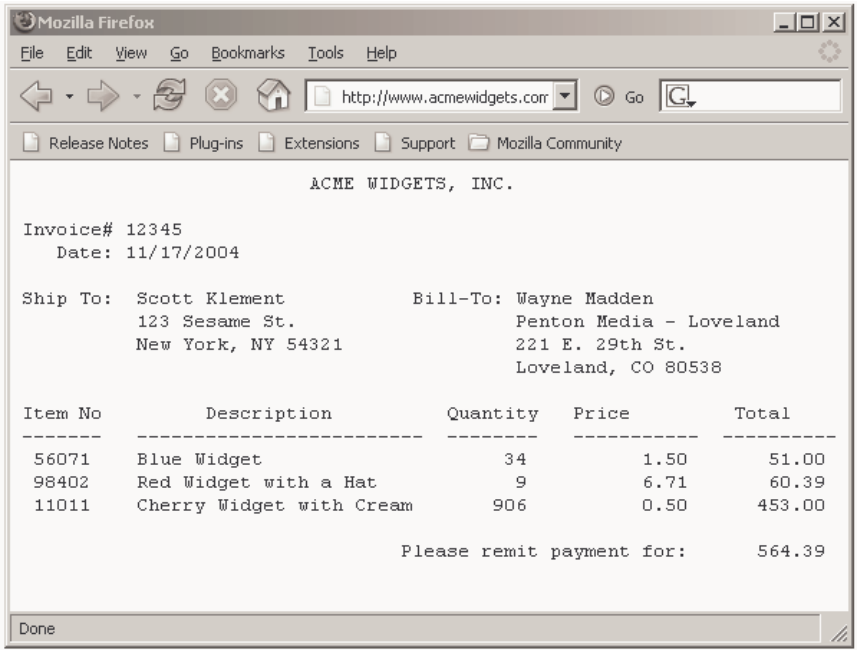
In a typical web application...

- A Web browser displays a web page containing input fields
- The user types some data or makes some selections
- The browser sends the data to a web server which then passes it on to a program
- After processing, the program spits out a new web page for the browser to display

Web Enabled Invoice



Web Enabled Invoice



An idea is born

Eureka! Our company could save time!

- Automatically download the invoice in a program.
- Read the invoice from the download file, get the invoice number as a substring of the 3rd line
- Get the date as a substring of the 4th line
- Get the addresses from lines 6-9

Problem: The data is intended for people to read. Not a computer program!

- Data could be moved, images inserted, colors added
- Every vendor's invoice would be complex & different

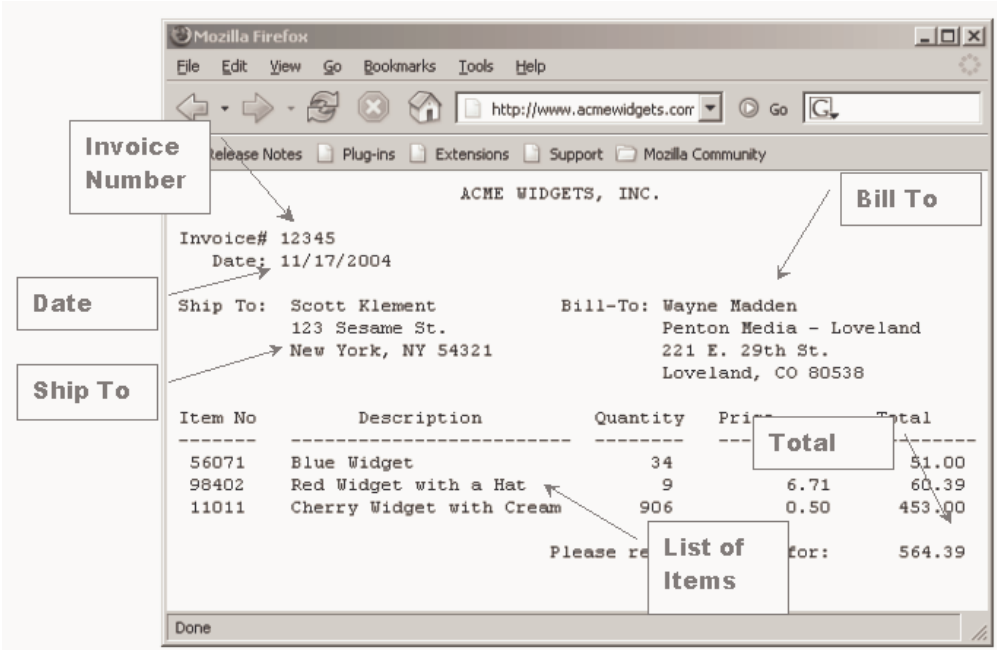
Need to Know "What"

What you want to know is *what* things are, rather than:

- Where they sit on a page.
- What they look like

The vendor needs to send data that's "marked up."

"Marked Up" Data



"Marked Up" Data with XML

```
<invoice>
  <remitto>
    <company>Acme Widgets, Inc</company>
  </remitto>
  <shipto>
    <name>Scott Klement</name>
    <address>
      <addrline1>123 Sesame St.</addrline1>
      <city>New York</city>
      <state>NY</state>
      <postalCode>54321</postalCode>
    </address>
  </shipto>
  <billto>
    <name>Wayne Madden</name>
    <company>Penton Media - Loveland</company>
    <address>
      <addrline1>221 E. 29th St.</addrline1>
      <city>Loveland</city>
      <state>CO</state>
      <postalCode>80538</postalCode>
    </address>
  </billto>
```

"Marked Up" Data with XML

```
 <itemlist>
    <item>
      <itemno>56071</itemno>
      <description>Blue Widget</description>
      <quantity>34</quantity>
      <price>1.50</price>
      <linetotal>51.00</linetotal>
    </item>
    <item>
      <itemno>98402</itemno>
      <description>Red Widget with a Hat</description>
      <quantity>9</quantity>
      <price>6.71</price>
      <linetotal>60.39</linetotal>
    </item>
    <item>
      <itemno>11011</itemno>
      <description>Cherry Widget</description>
      <quantity>906</quantity>
      <price>0.50</price>
      <linetotal>453.00</linetotal>
    </item>
  </itemlist>
  <total>564.39</total>
</invoice>
```

What is a Web Service?

A "program call" (or subprocedure call) that works over the Web.

- Very similar in concept to the CALL command.

```
CALL PGM(EXCHRATE) PARM('us' 'euro' &DOLLARS &EUROS)
```

- Runs over the Web, so can call programs on other computers anywhere in the world.
- Works on intranets as well

Imagine what you can do....

Imagine these scenarios...

Imagine some scenarios:

- You're writing a program that generates price quotes. Your quotes are in US dollars. Your customer is in Germany. You can call a program that's located out on the Internet somewhere to get the current exchange rate for the Euro.
- You're accepting credit cards for payment. After your customer keys a credit card number into your application, you call a program on your bank's computer to get the purchase approved instantly.
- You've accepted an order from a customer, and want to ship the goods via UPS. You can call a program running on UPS's computer system and have it calculate the cost of the shipment while you wait.
- Later, you can track that same shipment by calling a tracking program on UPS's system. You can have up-to-the-minute information about where the package is.

These are not just dreams of the future. They are a reality today with Web services.

Examples of Web Services

United Parcel Service (UPS) provides web services for:

- Verifying Package Delivery
- Viewing the signature that was put on a package
- Package Time-in-Transit
- Calculating Rates and Services
- Obtaining correct shipping information (zip codes, etc.)

- FedEx provides web services as well.
- United States Postal Service
- Amazon.com

- Validate Credit Cards
- Get Stock Quotes
- Check the Weather

How do they work?

A “program call” over the Web.

- You make an XML document that specifies the program (or “operation”) to call, as well as it’s input parameters.
- You use the HTTP protocol (the one your browser uses to download web pages) to send that XML document to a Web server.
- The Web server runs a program on it’s side, and outputs a new XML document containing the output parameters.

SOAP and WSDL

Although there's a few different ways of calling web services today, things are becoming more and more standardized. The industry is standardizing on a technology called SOAP.

SOAP = Simple Object Access Protocol

SOAP is an XML language that describes the parameters that you pass to the programs that you call. When calling a Web service, there are two SOAP documents -- an input document that you send to the program you're calling, and an output document that gets sent back to you.

The format of a SOAP message can be determined from another XML document called a WSDL (pronounced "wiz-dull") document.

WSDL = Web Services Description Language

A WSDL document will describe the different "programs you can call" (or "operations" you can perform), as well as the parameters that need to be passed to those operations.

WSDL

```
<definitions>

    <types>
       definition of types........
    </types>

    <message>
       definition of a message....
    </message>

    <portType>
       definition of a port.......
    </portType>

    <binding>
       definition of a binding....
    </binding>

    <service>
       a logical grouping of ports...
    </service>

</definitions>
```

<types> = the data types that the web service uses.

<message> = the messages that are sent to and received from the web service.

<portType> = the operations (or, “programs/procedures” you can call for this web service.

<binding> = the network protocol used.

<service> = a grouping of ports. (Much like a service program contains a group of subprocedures.)

Sample WSDL

```
<definitions name="CurrencyExchangeService">
   <message name="getRateRequest">
      <part name="country1" type="xsd:string"/>
      <part name="country2" type="xsd:string"/>
   </message>
   <message name="getRateResponse">
      <part name="Result" type="xsd:float"/>
   </message>
   <portType name="CurrencyExchangePortType">
      <operation name="getRate">
         <input message="tns:getRateRequest" />
         <output message="tns:getRateResponse" />
      </operation>
   </portType>
   <binding name="CurrencyExchangeBinding"
            type="tns:CurrencyExchangePortType">
      <soap:binding style="rpc"
                    transport="http://schemas.xmlsoap.org/soap/http" />
      <operation name="getRate">
         <soap:operation soapAction=""/>
      </operation>
   </binding>
   <service name="CurrencyExchangeService">
      <documentation>
         Returns the exchange rate between the two currencies
      </documentation>
      <port name="CurrencyExchangePort">
         <soap:address location="http://services.xmethods.net:80/soap"/>
      </port>
   </service>
</definitions>
```

Note: I removed the namespace identifiers and encodings to simplify the document a little bit.

Read it from the bottom up!

SOAP

Here's the skeleton of a SOAP message:

```
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
       soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding" >

    <soap:Header>
        (optional) contains header info, like payment info or authentication info
          (crypto key, userid/password, etc)
    </soap:Header>

    <soap:Body>
        . . .
        Contains the operation name (i.e. the "procedure to call") as
        well as parameter info. These are application defined.
        . . .
        <soap:Fault>
          (optional) error info.
        </soap:Fault>
        . . .
    </soap:Body>

</soap:Envelope>
```

Sample SOAP Documents

Again, I've removed the namespace and encoding information to keep this example clear and simple. (In a real program, you'd need those to be included as well.)

Input Message

```
<?xml version="1.0"?>
<SOAP:Envelope>
   <SOAP:Body>
     <n:getRate>
       <country1 xsi:type="xsd:string">US</country1>
       <country2 xsi:type="xsd:string">euro</country2>
     </n:getRate>
   </SOAP:Body>
</SOAP:Envelope>
```

Output Message

```
<?xml version="1.0"?>
<SOAP:Envelope>
   <SOAP:Body>
     <n:getRate>
       <Result>0.8358</Result>
     </n:getRate>
   </SOAP:Body>
</SOAP:Envelope>
```

HTTPAPI

Now that you know the XML data that needs to be sent and received, you need a method of sending that data to the server, and getting it back.

Normally when we use the Web, we use a Web browser. The browser connects to a web server, issues our request, downloads the result and displays it on the screen.

When making a program-to-program call, however, a browser isn't the right tool. Instead, you need a tool that knows how to send and receive data from a Web server that can be integrated right into your RPG programs.

That's what HTTPAPI is for!

- HTTPAPI is a free (open source) tool to act like an HTTP client (the role usually played by the browser.)
- HTTPAPI was originally written by me (Scott Klement) to assist with a project that I had back in 2001.
- Since I thought it might be useful to others, I made it free and available to everyone.

http://www.scottklement.com/httpapi/

More about HTTPAPI

How did HTTPAPI come about?

- I needed a way to automate downloading ACS updates from the United States Postal Service
- A friend needed a way to track packages with UPS from his RPG software
- Since many people seemed to need this type of application, I decided to make it publicly available under an Open Source license

Currency Exchange Example

I've shown you the sample WSDL and SOAP documents for XMethod.net's "Currency Exchange" demonstration web service.

Over the next several slides, we'll look at an RPG example that uses HTTPAPI to consume this Currency Exchange Service.

This type of program is often referred to as a *Web Service Consumer*.

In business, a customer that utilizes your product is referred to as a "consumer". For example, if my company makes sausage, and you buy one from the grocery store and eat it, you're the "end consumer."

This is analogous to a program that uses a web service. When the service is used, it's referred to as "consuming" the service. Therefore, a program that utilizes a Web service is a Web Service Consumer.

Web Service Consumer (1/4)

```
H DFTACTGRP(*NO) BNDDIR('LIBHTTP/HTTPAPI')

D EXCHRATE        PR                  ExtPgm('EXCHRATE')
D   Country1                    32A   const
D   Country2                    32A   const
D   Amount                      15P 5 const
D EXCHRATE        PI
D   Country1                    32A   const
D   Country2                    32A   const
D   Amount                      15P 5 const

 /copy libhttp/qrpglesrc,httpapi_h

D Incoming        PR
D   rate                         8F
D   depth                       10I 0 value
D   name                      1024A   varying const
D   path                     24576A   varying const
D   value                    32767A   varying const
D   attrs                         *   dim(32767)
D                                     const options(*varsize)

D SOAP            s           32767A   varying
D rc              s              10I 0
D rate            s               8F
D Result          s              12P 2
D msg             s              50A
D wait            s               1A
```

A program that uses a Web Service is called a "Web Service Consumer".

The act of calling a Web service is referred to as "consuming a web service."

Web Service Consumer (2/4)

```
/free
  SOAP = '<?xml version="1.0" encoding="US-ASCII" standalone="no"?>'
       + '<SOAP-ENV:Envelope'
       + ' xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"'
       + ' xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"'
       + ' xmlns:xsd="http://www.w3.org/1999/XMLSchema">'
       + '<SOAP-ENV:Body>'
       + ' <n:getRate'
       + '   xmlns:n="urn:xmethods-CurrencyExchange"'
       + '   SOAP-ENV:encodingStyle='
              + '"http://schemas.xmlsoap.org/soap/encoding">'
       + '      <country1 xsi:type="xsd:string">'
       +          %trim(Country1) + '</country1>'
       + '      <country2 xsi:type="xsd:string">'
       +          %trim(Country2) + '</country2>'
       + ' </n:getRate>'
       + '</SOAP-ENV:Body>'
       + '</SOAP-ENV:Envelope>';

       rc = http_url_post_xml( 'http://services.xmethods.net:80/soap'
                             : %addr(SOAP) + 2
                             : %len(SOAP)
                             : *NULL
                             : %paddr(Incoming)
                             : %addr(rate)
                             : HTTP_TIMEOUT
                             : HTTP_USERAGENT
                             : 'text/xml'
                             : *blanks );
```

Constructing the SOAP message is done with a big EVAL statement.

This routine tells HTTPAPI to send the SOAP message to a Web server, and to parse the XML response.

As HTTPAPI receives the XML document, it'll call the INCOMING subpocedure for every XML element, passing the "rate" variable as a parameter.

Web Service Consumer (3/4)

If an error occurs, ask HTTPAPI what the error is.

```
   if (rc <> 1);
      msg = http_error();
   else;
      Result = %dech(Amount * rate: 12: 2);
      msg = 'Result = ' + %char(Result);
   endif;

   dsply msg ' ' wait;

   *inlr = *on;

  /end-free

P Incoming        B
D Incoming        PI
D   rate                          8F
D   depth                       10I 0 value
D   name                      1024A   varying const
D   path                     24576A   varying const
D   value                    32767A   varying const
D   attrs                         *   dim(32767)
D                                     const options(*varsize)

  /free
      if (name = 'Result');
         rate = %float(value);
      endif;
  /end-free
P                 E
```

If no errors occur, then display the result on the screen.

This is called for every XML element in the response. When the element is a "Result" element, save the value, since it's the exchange rate we're looking for!

Web Service Consumer (4/4)

Here's a sample of the output from calling the preceding program:

```
                              Command Entry
                                                        Request level:   1
 Previous commands and messages:
    > call exchrate parm('us' 'euro' 185.93)
      DSPLY  Result = 155.53









Bottom
 Type command, press Enter.
 ===>

 F3=Exit    F4=Prompt    F9=Retrieve    F10=Include detailed messages
 F11=Display full       F12=Cancel     F13=Information Assistant  F24=More keys
```

What Just Happened?

HTTPAPI does not know how to create an XML document, but it does know how to parse one.

In the previous example:

- The SOAP document was created in a variable using a big EVAL statement.
- The variable that contained the SOAP document was passed to HTTPAPI and HTTPAPI sent it to the Web site.
- The subprocedure we called (**`http_url_post_xml`**) utilizes HTTPAPI's built-in XML parser to parse the result as it comes over the wire.
- As each XML element is received, the **`Incoming()`** subprocedure is called.
- When that subprocedure finds a **`<Result>`** element, it saves the element's value to the "rate" variable.
- When **`http_url_post_xml()`** has completed, the rate variable is set. You can multiply the input currency amount by the rate to get the output currency amount.

No! Let Me Parse It!

If you don't want to use HTTPAPI's XML parser, you can call the **`http_url_post()`** API instead of **`http_url_post_xml()`.**

In that situation, the result will be saved to a stream file in the IFS, and you can use another XML parser instead of the one in HTTPAPI.

```
. . .
  rc = http_url_post( 'http://services.xmethods.net:80/soap'
                    : %addr(SOAP) + 2
                    : %len(SOAP)
                    : *NULL
                    : '/tmp/CurrencyExchangeResult.soap'
                    : HTTP_TIMEOUT
                    : HTTP_USERAGENT
                    : 'text/xml'
                    : *blanks );
. . .
```

For example, you may want to use RPG's built in support for XML in V5R4 to parse the document rather than let HTTPAPI do it.

Is /FREE required?

In my examples so far, i've used free format RPG. It's not required, however, you can use fixed format if you prefer. Just use EVAL or CALLP statements.

```
. . .
 c                   eval      SOAP = '<?xml version="1.0">'
 c                             + '<SOAP-ENV:Envelope  . .
      . . . and so on . . .

 c                   callp     http_url_post_xml(
 c                               'http://services.xmethods.net:80/soap'
 c                             : %addr(SOAP) + 2
 c                             : %len(SOAP)
 c                             : *NULL
 c                             : %paddr(Incoming)
 c                             : %addr(rate)
 c                             : HTTP_TIMEOUT
 c                             : HTTP_USERAGENT
 c                             : 'text/xml'
 c                             : *blanks )
. . .
```

Handling Errors with HTTP API

Most of the HTTPAPI routines return 1 when successful

- **Although this allows you to detect when something has failed, it only tells you *that* something failed, not *what* failed**
- **The `http_error()` routine can tell you an error number, a message, or both**
- **The following is the prototype for the `http_error()` API**

```
D http_error      PR            80A
D   peErrorNo                   10I 0 options(*nopass)
```

The human-readable message is particularly useful for letting the user know what's going on.

```
        if ( rc <> 1 );
            msg = http_error();
            // you can now print this message on the screen,
            // or pass it back to a calling program,
            // or whatever you like.
        endif;
```

Handling Errors, continued…

The error number is useful when the program anticipates and tries to handle certain errors.

```
if ( rc <> 1 );

    http_error(errnum);

    select;
    when errnum = HTTP_NOTREG;
      // app needs to be registered with DCM
      exsr RegisterApp;
    when errnum = HTTP_NDAUTH;
      // site requires a userid/password
      exsr RequestAuth;
    other;
      msg = http_error();
    endsl;

endif;
```

These are constants that are defined in HTTPAPI_H (and included with HTTPAPI)

About SSL with HTTPAPI

The next example (UPS package tracking) requires that you connect using SSL. (This is even more important when working with a bank!)

HTTPAPI supports SSL when you specify "https:" instead of "http:" at the beginning of the URL.

It uses the SSL routines in the operating system, therefore you must have all of the required software installed. IBM requires the following:

- Digital Certificate Manager (option 34 of OS/400, 5722-SS1)
- TCP/IP Connectivity Utilities for iSeries (5722-TC1)
- IBM HTTP Server for iSeries (5722-DG1)
- IBM Developer Kit for Java (5722-JV1)
- IBM Cryptographic Access Provider (5722-AC3)

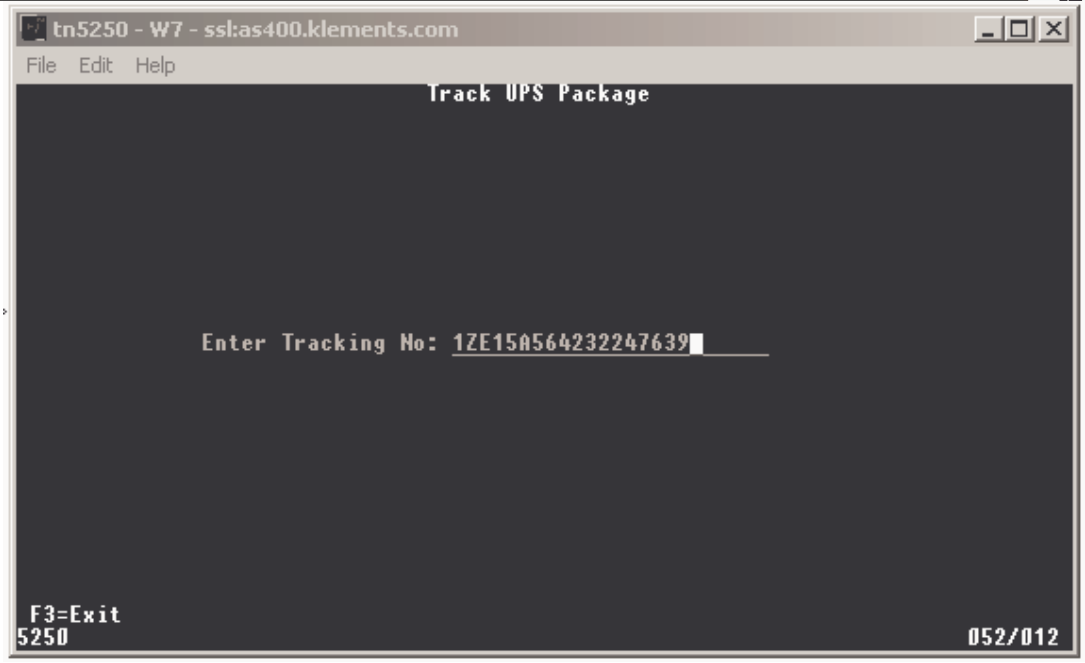
Because of import/export laws, 5722-AC3 is not shipped with OS/400. However, it's normally a no-charge item. You just have to order it separately from your business partner.

UPS Example (slide 1 of 11)

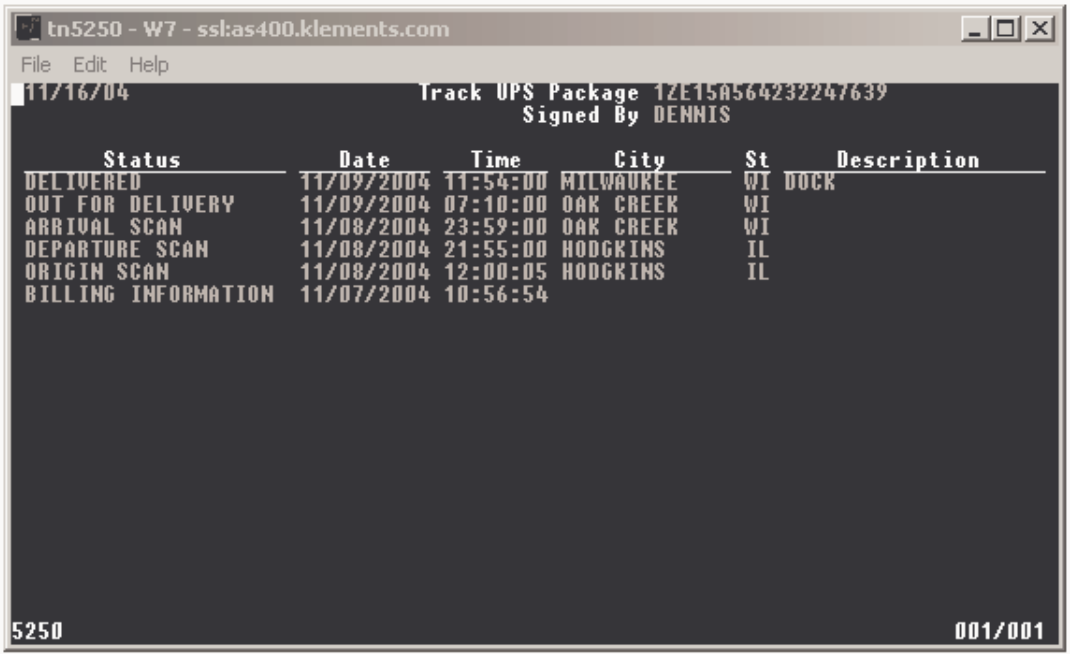
This demonstrates the "UPS Tracking Tool" that's part of UPS OnLine Tools. There are a few differences between this and the previous example:

- **You have to register with UPS to use their services (but it's free)**
- **You'll be given an access key, and you'll need to send it with each request.**
- **UPS requires SSL to access their web site.**
- **UPS does not use SOAP or (as far as I can tell) WSDL for their Web services.**
- **Instead, they provide you with documentation that explains the format of the XML messages.**
- **That document will be available from their web site after you've signed up as a developer.**

UPS Example (slide 2 of 11)



UPS Example (slide 3 of 11)



UPS Example (slide 4 of 11)

```
. . .

  D UPS_USERID      C                   '<put your userid here>'
  D UPS_PASSWD      C                   '<put your password here>'
  D UPS_LICENSE     C                   '<put your access license here>'

. . .

  d act             s             10I 0
  d activity        ds                  qualified
  d                                     dim(10)
  d   Date                         8A
  d   Time                         6A
  D   Desc                        20A
  D   City                        20A
  D   State                        2A
  D   Status                      20A
  D   SignedBy                    20A


. . .
      // Ask user for tracking number.
      exfmt TrackNo;
```

UPS provides these when you sign up as a developer.

UPS Example (slide 5 of 11)

```
postData =
   '<?xml version="1.0"?>'                                          +
   '<AccessRequest xml:lang="en-US">'                               +
     '<AccessLicenseNumber>' + UPS_LICENSE + '</AccessLicenseNumber>' +
     '<UserId>' + UPS_USERID + '</UserId>'                          +
     '<Password>' + UPS_PASSWD + '</Password>'                      +
   '</AccessRequest>'                                               +
   '<?xml version="1.0"?>'                                          +
   '<TrackRequest xml:lang="en-US">'                                +
     '<Request>'                                                    +
       '<TransactionReference>'                                     +
         '<CustomerContext>Example 1</CustomerContext>'             +
         '<XpciVersion>1.0001</XpciVersion>'                        +
       '</TransactionReference>'                                    +
       '<RequestAction>Track</RequestAction>'                       +
       '<RequestOption>activity</RequestOption>'                    +
     '</Request>'                                                   +
     '<TrackingNumber>' + TrackingNo + '</TrackingNumber>'          +
   '</TrackRequest>'                                                ;

 rc = http_url_post_xml('https://wwwcie.ups.com/ups.app/xml/Track'
                        : %addr(postData) + 2
                        : %len(postData)
                        : %paddr(StartOfElement)
                        : %paddr(EndOfElement)
                        : *NULL );
 if (rc <> 1);
    msg = http_error();
    // REPORT ERROR TO USER
 endif;
```

The StartOfElement and EndOfElement routines are called while http_url_post_xml is running

UPS Example (slide 6 of 11)

```
. . .
    for RRN = 1 to act;
        monitor;
          tempDate = %date(activity(RRN).date: *ISO0);
          scDate = %char(tempDate: *USA);
        on-error;
          scDate = *blanks;
        endmon;

        monitor;
          tempTime = %time(activity(RRN).time: *HMS0);
          scTime = %char(tempTime: *HMS);
        on-error;
          scTime = *blanks;
        endmon;

        scDesc = activity(RRN).desc;
        scCity = activity(RRN).city;
        scState = activity(RRN).state;
        scStatus = activity(RRN).status;

        if (scSignedBy = *blanks);
           scSignedBy = activity(RRN).SignedBy;
        endif;

        write SFLREC;
    endfor;
. . .
```

Since the StartOfElement and EndOfElement routines read the XML data and put it in the array, when http_url_post_xml is complete, we're ready to load the array into the subfile.

UPS Example (slide 7 of 11)

```
<?xml version="1.0" ?>
<TrackResponse>
  <Shipment>
. . .
    <Package>
      <Activity>
        <ActivityLocation>
          <Address>
            <City>MILWAUKEE</City>
            <StateProvinceCode>WI</StateProvinceCode>
            <PostalCode>53207</PostalCode>
            <CountryCode>US</CountryCode>
          </Address>
          <Code>AI</Code>
          <Description>DOCK</Description>
          <SignedForByName>DENNIS</SignedForByName>
        </ActivityLocation>
        <Status>
          <StatusType>
            <Code>D</Code>
            <Description>DELIVERED</Description>
          </StatusType>
          <StatusCode>
            <Code>KB</Code>
          </StatusCode>
        </Status>
       <Date>20041109</Date>
       <Time>115400</Time>
      </Activity>
```

This is what the response from UPS will look like.

HTTPAPI will call the `StartOfElement` procedure for every "start" XML element.

HTTPAPI will call the `EndOfElement` procedure for every "end" XML element. At that time, it'll also pass the value.

UPS Example (slide 8 of 11)

```
      <Activity>
        <ActivityLocation>
          <Address>
            <City>OAK CREEK</City>
            <StateProvinceCode>WI</StateProvinceCode>
            <CountryCode>US</CountryCode>
          </Address>
        </ActivityLocation>
        <Status>
          <StatusType>
            <Code>I</Code>
            <Description>OUT FOR DELIVERY</Description>
          </StatusType>
          <StatusCode>
            <Code>DS</Code>
          </StatusCode>
        </Status>
        <Date>20041109</Date>
        <Time>071000</Time>
      </Activity>
    . . .
    </Package>
  </Shipment>
</TrackResponse>
```

There are additional `<Activity>` sections and other XML that I omitted because it was too long for the presentation.

UPS Example (slide 9 of 11)

```
P StartOfElement  B
D StartOfElement  PI
D   UserData                    *   value
D   depth                     10I 0 value
D   name                    1024A   varying const
D   path                   24576A   varying const
D   attrs                       *   dim(32767)
D                                   const options(*varsize)
 /free

   if path = '/TrackResponse/Shipment/Package' and name='Activity';
      act = act + 1;
   endif;

 /end-free
P                 E
```

This is called during **`http_url_post_xml()`** for each start element that UPS sends. It's used to advance to the next array entry when a new package record is received.

UPS Example (slide 10 of 11)

```
P EndOfElement    B
D EndOfElement    PI
D   UserData                    *   value
D   depth                     10I 0 value
D   name                    1024A   varying const
D   path                   24576A   varying const
D   value                  32767A   varying const
D   attrs                       *   dim(32767)
D                                   const options(*varsize)
 /free

  select;
  when  path = '/TrackResponse/Shipment/Package/Activity';

      select;
      when name = 'Date';
        activity(act).Date = value;
      when name = 'Time';
        activity(act).Time = value;
      endsl;

  when  path = '/TrackResponse/Shipment/Package/Activity' +
               '/ActivityLocation';

      select;
      when name = 'Description';
        activity(act).Desc = value;
      when name = 'SignedForByName';
        activity(act).SignedBy = value;
      endsl;
```

This is called for each ending value. We use it to save the returned package information into an array.

Remember, this is called by `http_url_post_xml`, so it'll run before the code that loads this array into the subfile!

UPS Example (slide 11 of 11)

```
      when  path = '/TrackResponse/Shipment/Package/Activity' +
                   '/ActivityLocation/Address';

          select;
          when name = 'City';
            activity(act).City = value;
          when name = 'StateProvinceCode';
            activity(act).State = value;
          endsl;

      when  path = '/TrackResponse/Shipment/Package/Activity' +
                   '/Status/StatusType';

          if   name = 'Description';
              activity(act).Status = value;
          endif;

       endsl;

     /end-free
    P                 E
```

For More Information

You can download *HTTPAPI* from Scott's Web site:
http://www.scottklement.com/httpapi/

Most of the documentation for *HTTPAPI* is in the source code itself.
- **Read the comments in the HTTPAPI_H member**
- **Sample programs called EXAMPLE1 - EXAMPLE18**

The best place to get help for *HTTPAPI* is in the mailing list. There's a link to sign up for this list on Scott's site.

Info about Web Services:
- ***Web Services: The Next Big Thing* by Scott N. Gerard**
  http://www.iseriesnetwork.com/Article.cfm?ID=11607
- ***Will Web Services Serve You?* by Aaron Bartell**
  http://www.iseriesnetwork.com/Article.cfm?ID=19651
- ***W3 Consortium***
  http://www.w3.org and http://www.w3schools.com

For More Information

Web Service info, continued…

- ***RPG as a Web Service Consumer*** **by Scott Klement**
  http://www.iseriesnetwork.com/article.cfm?id=52099
- **XMethods.net**
  http://www.xmethods.net
- **UPS OnLine Tools**
  http://www.ups.com/content/us/en/bussol/offering/technology/automated_shipping/online_tools.html

XML Resources

- **Scott also gives a presentation on XML programming with Expat**
  - ✓ **On Scott's Web site: http://www.scottklement.com/expat/**
  - ✓ **In Club Tech iSeries Programming Tips newsletter:**
    http://www.iseriesnetwork.com/article.cfm?id=20046
    http://www.iseriesnetwork.com/article.cfm?id=20093
    http://www.iseriesnetwork.com/article.cfm?id=20100
    http://www.iseriesnetwork.com/article.cfm?id=50719

Conclusion

- Web services open a whole new world of business possibilities.
- Maybe some day, they'll replace EDI?
- It's possible to consume Web services with plain ol' RPG. No Java or WebSphere required.

Questions?

This Presentation

You can download a PDF copy of this presentation from:

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

Thank you!