Web Services for RPGers

Presented by

Scott Klement

http://www.scottklement.com

"A computer once beat me at chess, but it was no match for me at kick boxing." — Emo Philips

Our Agenda

This workshop consists of three parts:

1. Introduction & Concepts
   - What's a web service?
   - Terminology and syntax
   - REST vs SOAP
   - XML vs JSON
2. Providing a Web Service with RPG
   - Using the IBM's Integrated Web Services
   - Manually with Apache
3. Consuming a Web Service with RPG
   - With SoapUI / HTTPAPI
   - With WSDL2RPG

I am a Web Service. What Am I?

A routine (program? Subprocedure?) that can be called over a TCP/IP network. (Your LAN? Intranet?

- A callable routine. (Program? Subprocedure?)
- Callable over a TCP/IP Network. (LAN? Intranet? Internet?)
- ….can also be called from the same computer.
- Using the HTTP (or HTTPS) network protocol

Despite the name, not necessarily "web"

- different from a "web site" or "web application"
- input and output are via "parameters" (of sorts) and are for programs to use. No user interface -- not even a browser.
- can be used *from* a web application (just as an API or program could) either from JavaScript in the browser, or from a server-side programming language like RPG, PHP, .NET or Java
- but is just as likely to be called from other environments… even 5250!

Write Once, Call From Anywhere

In other words… Services Oriented Architecture (SOA).

- Your business logic (business rules) are implemented as a set of "services" to any caller that needs them.
- Web services are only *one of many* ways to implement SOA. Don't believe the hype!

Callable from anywhere

- Any other program, written in (just about) any language.
- From the same computer, or from another one.
- From the same office (data center), or from another one.
- From folks in the same company, or (if desired) any of your business partners. Even the public, if you want!

RPG can function as either a *provider* (server) or a *consumer* (client)

Two Sides To Every Story

In Web Services there are always two sides.

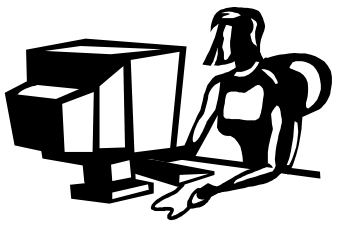
CONSUMER: The program "making the call".

- The program that "needs something"
- Usually is interfacing with the user
- The "client" program (vs. server program)
- Example: An order entry program might 'consume' a web service to look up shipping rates. This makes that program the 'consumer'.

PROVIDER: The program "providing the service".

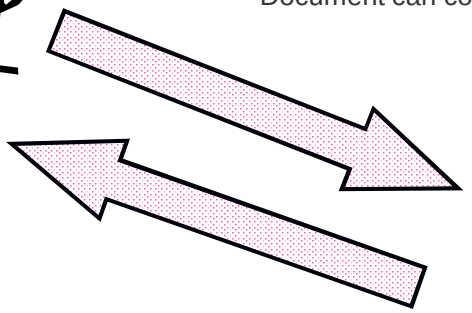
- Sits in the background waiting for requests from consumers.
- the "server" (vs. client) side of the conversation
- Example: A program on UPS's computer (or FedEx, DHL, etc) that accepts a weight, shipment type, and destination and calculates the shipping rate.

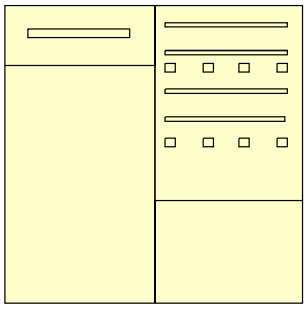
How Do They Work?



HTTP starts with a request for the server

- Can include a document (XML, JSON, etc)
- Document can contain "input parameters"





HTTP then runs server-side program

- input document is given to program
- HTTP waits til program completes.
- program outputs a new document (XML, JSON, etc)
- document contains "output parameters"
- document is returned to calling program.

Don't Confuse this With a Web Page

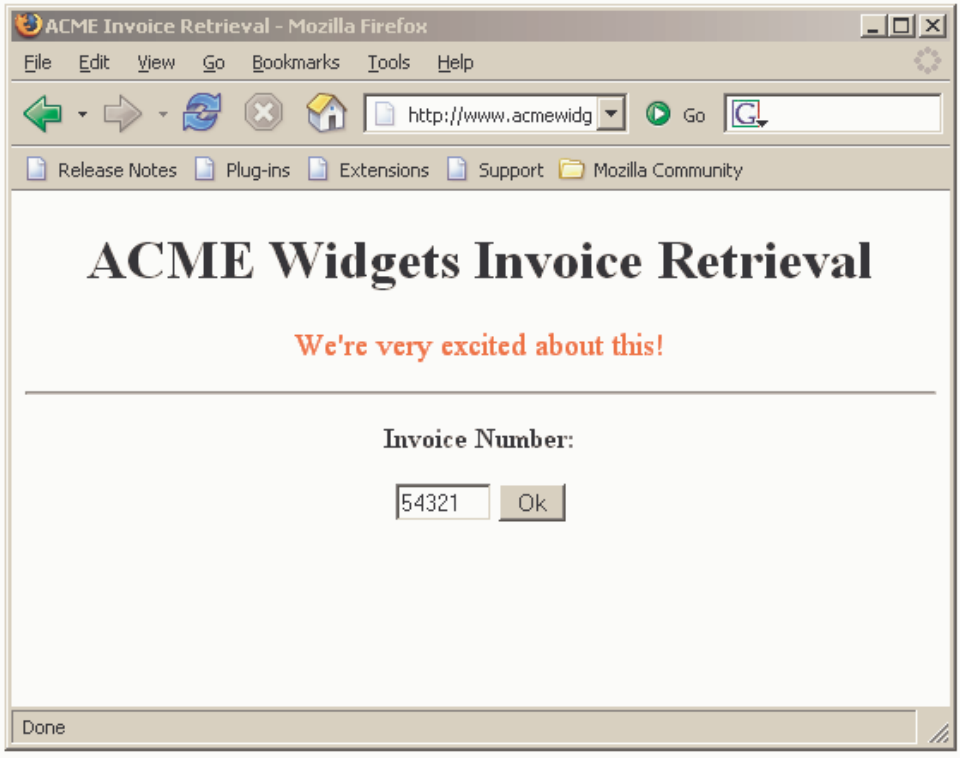
A web page is for displaying data to a user.

A web service is for program-to-program communication.

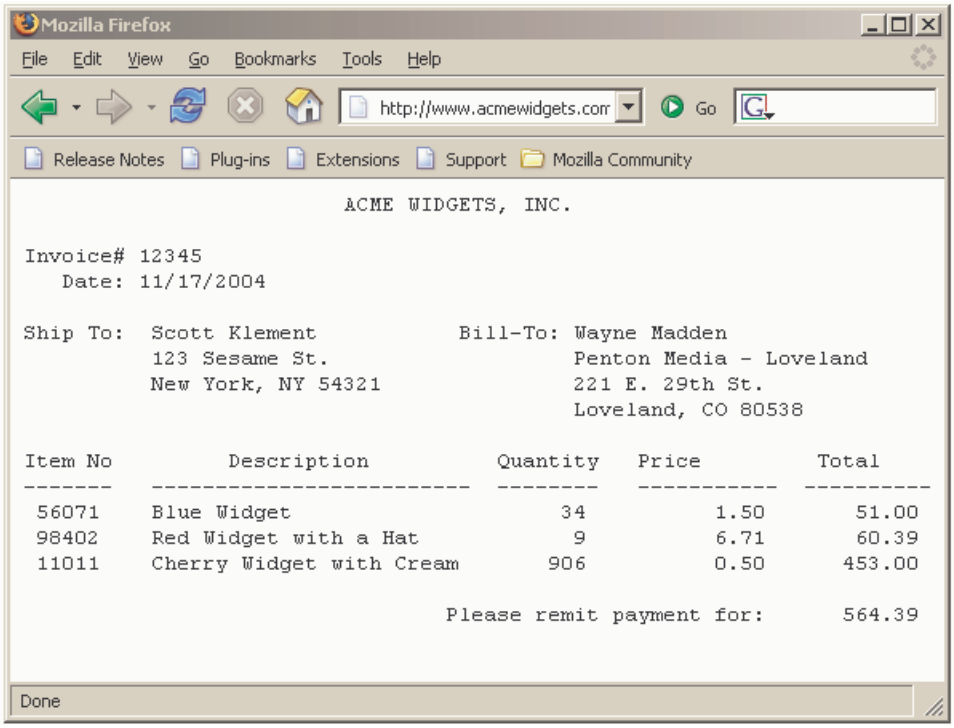
Though, it's possible for a web page to contain program code (JavaScript) that calls a web service, there is a significant difference between a web page (or "web application") and a web service.

Let's take a look at how they are different…

Web Page (Invoice)



Web Page (Invoice) Result



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"

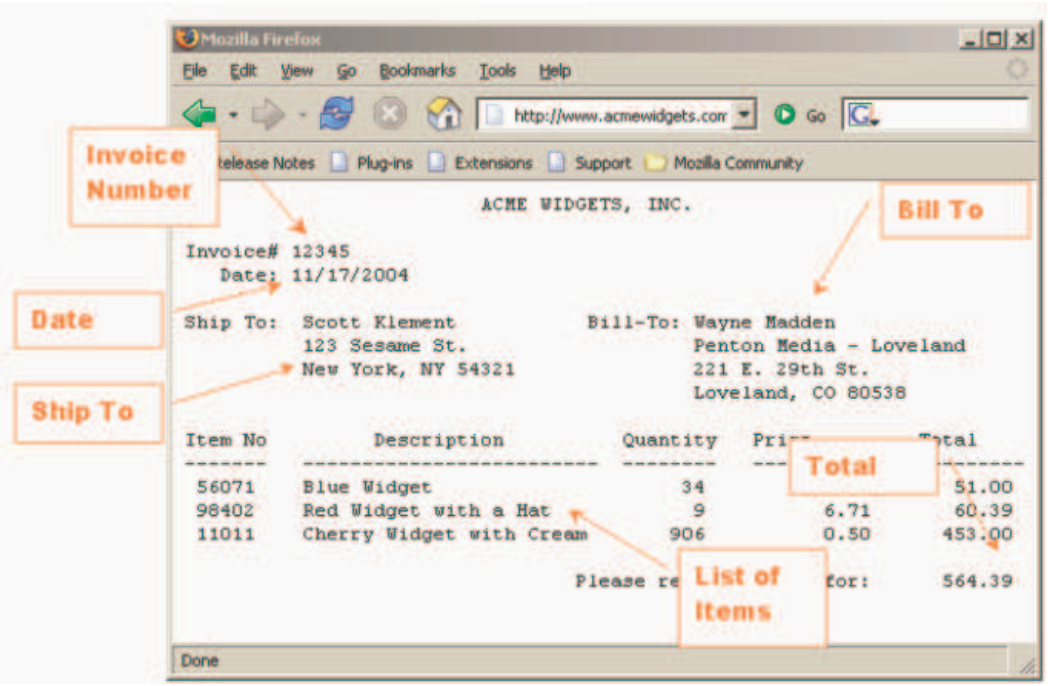
What you want to know is *what* things are, rather than:

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

The vendor needs to send data that's designed for a computer program to read.

Data should be "marked up."

"Marked Up" Data



One Way to "Mark Up" is XML

Quick XML Syntax Review

Elements

- An XML opening tag and closing tag.
- Optionally with character data in between.

<company> Acme Widgets, Inc </company>
(opening) char data (closing)

- Elements can be nested (see next slide)

Attributes

- Looks like a variable assignment
  <company name="Acme Widgets, Inc"> </company>
- Opening/Closing Can Be Combined (a "shortcut")
  <company name="Acme Widgets, Inc" />
- Possible to have multiple attributes and character data
  <company custno="1234" type="remit">Acme Widgets, Inc</company>

"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>
```


XML Is Only One Option

XML was the original option…

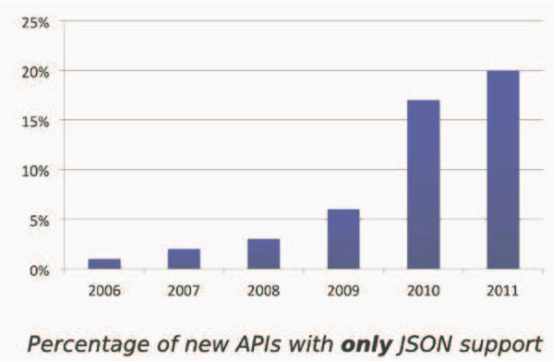
- As discussed, it identifies "what"
- Possible to add more info without breaking compatibility
- Readable from any modern programming language
- Self-describing (well, sort of.)

Not all web services use XML

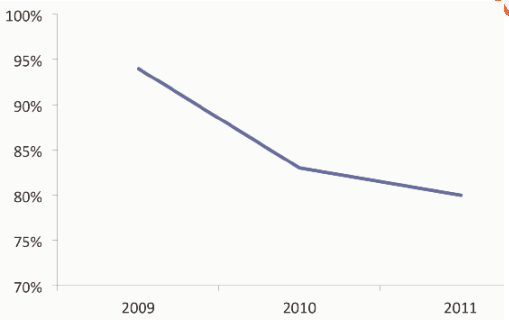
- Some do use it for both input and output
- Some use it only for output, and get input via URL
- Some use other formats (most commonly, JSON)

As time goes on, JSON has been overtaking XML.

JSON is Quickly Becoming Important



Percentage of new APIs with **only** JSON support



Percentage of APIs with XML support

Over 70% of all APIs in ProgrammableWeb's API directory are RESTful, increasingly at the expense of SOAP. More than 55% of those same APIs support JSON output, with 20% opting not to offer XML at all.

Source: 1 in 5 APIs Say "Bye XML", Adam DuVander, May 25, 2011

The JSON Concept

Originally JSON was the language used to describe "initializers" for JavaScript objects.

- Used to set the initial values of JavaScript Objects (data structures), and arrays. Even for arrays nested in data structures or vice-versa.
- Conceptually similar to "CTDATA" in RPG, except supports nested data as well.
- Unlike JavaScript, however, JSON does not support "methods" (executable routines in the object) so it's objects are equivalent to RPG data structures.

```
var DaysOfWeek = [ "Sunday",
                   "Monday",
                   "Tuesday",
                   "Wednesday",
                   "Thursday",
                   "Friday",
                   "Saturday" ];
```

JSON Syntax Summary

Arrays start/end with square brackets

```
[ "Monday", "Tuesday", "Wednesday", "Thursday", "Friday" ]
```

Objects (data structures in RPG) start/end with curly braces { x, x, x, x }

```
{  "first": "Scott", "last": "Klement", "sex": "male" }
```

Strings are in double-quotes. Quotes and control characters are escaped with backslashes. Numbers and true/false are not quoted.

```
{ "name": "Henry \"Hank\" Aaron",  "home_runs": 755, "retired": true }
```

Names are separated from values with a colon (as above)

Successive elements (array elements or fields in an object) are separated by commas. (as above)

Data can be nested (arrays inside objects and/or objects inside arrays).

Much Like XML

JSON is a format for encapsulating data as it's sent over networks

Much Like XML.

JSON is self-describing (field names are in the data itself) and human-readable.

Much Like XML

Very popular in Web Services and AJAX

Much Like XML

Can be used by all major programming languages

Much Like XML

So why is it better than XML…..?

Much Different Than XML

JSON is simpler:

- only supports UTF-8, whereas XML supports a variety of encodings.
- doesn't support schemas, transformations.
- doesn't support namespaces
- method of "escaping" data is much simpler.

JSON is faster

- more terse (less verbose). About 70% of XML's size on average
- simpler means faster to parse
- dead simple to use in JavaScript

JSON and XML to Represent a DS

```
D list             ds                    qualified
D                                        dim(2)
D   custno                          4p 0
D   name                           25a
```

For example, this is an array of a data structure in RPG.

```
[
  {
    "custno": 1000,
    "name": "ACME, Inc"
  },
  {
    "custno": 2000,
    "name": "Industrial Supply Limited"
  }
]
```

This is how the same array might be represented (with data inside) in a JSON document.

```
<list>
  <cust>
    <custno>1000</custno>
    <name>Acme, Inc</name>
  </cust>
  <cust>
    <custno>2000</custno>
    <name>Industrial Supply Limited</name>
  </cust>
</list>
```

And it's approximately the same as this XML document.

Types of Web Services

SOAP (Simple Object Access Protocol)

- Was the de-facto standard for several years
- Is standardized, but is sometimes a bit too complex
- Locked into XML as the only format. (Though other documents can be embedded inside XML.)

REST (REpresentational State Transfer)

- Has become the most popular type of web service
- Allows data in any format (usually XML or JSON)
- Simpler than SOAP, but less standardized

Others: POX, XML-RPC, etc are rarely used.

REST Web Services

```
http://www.scottklement.com/cust/495
```

- The URL is said to "represent" an object (or perhaps " a document") -- and sometimes also provides the input parameters
  - I like to think of it as "the noun"
- http ← the network protocol
- www.scottklement.com ← the server
- /cust/495 ← the thing you want to act upon (the "noun")
- The HTTP "method" (like an opcode) theoretically provides the "verb"
- Due to software limitations, sometimes part of the URL is used for the verb instead of the HTTP method.

Possible methods (and how they "change the state" of the object)

- GET (default) -- retrieve the customer -- same as typing URL into browser.
- POST -- create the customer (in which case you might upload a document)
- PUT -- modify the customer (also might upload a document)
- DELETE -- delete the customer

REST Noun Examples

```
http://www.scottklement.com/cust/OTHER-DATA-HERE
```

- `GET /cust` -- might return a list of all customers
- `GET /cust/495` – might return customer 495 details
- `POST /cust/496` – might create a new customer record for cust #496
- `PUT /cust/495` – might change the details of cust #495
- `DELETE /cust/496` – might remove the customer from the database

Of course, customer is just an example here. Could be anything:

- Creating an order? `POST /order`
- Retrieving an invoice listing? `GET /invoice/495/20100901/20100930`
- Check order status? `GET /order_status/12345`
- Add a new part to inventory? `PUT /warehouse/401/aisle6`

Also, just because you allow /cust/495 doesn't mean you HAVE to also allow /cust by itself to list all customers. Which options you provide are up to you.

Am I Being a Purist?

Technically, the URL for a REST web service should always identify the "noun" (the thing you're working with). This is considered the "true" meaning of REST.

But, many people will not follow that strictly. Often times, any service that puts input data in the URL will call itself "REST"… so don't be too much of a stickler.

Example:
http://www.scottklement.com/cust?custno=495&op=retrieve
or
http://www.scottklement.com/invlist?fromDate=20100901&toDate=20100930

A "purist" would say these are not truly REST web services, but there are many people out there that would call these REST.

RESTful Example

Easier way to think of REST

- all input is in URL
- output has no standard… can be anything (but usually is XML or JSON)

For example, you might have a web service that takes a customer number as input, and returns that customer's address.

Input

```
GET http://www.scottklement.com/cust/495
-or-
GET http://www.scottklement.com/cust/495?op=retrieve
```

Output

```
<result>
   <cust id="495">
      <name>ANCO FOODS</name>
      <street>1100 N.W. 33RD STREET</street>
      <city>POMPANO BEACH</city>
      <state>FL</state>
      <postal>33064-2121</postal>
   </cust>
</result>
```


REST With Multiple Parameters

- Although the previous slide had only one parameter, REST can have multiple parameters -- but they must all fit on the same URL.

```
http://www.scottklement.com/invoice/495/20100901/20100930
```

- This web service is designed to return a list of invoices for a given customer number, within a given date range.
- 495 = customer number
- 20100901 = start date (in year, month, date format)
- 20100930 = end date (in year, month, date format)

The web service would scan for the slashes, get the parameter info from the URL, and build an XML or JSON document that matches the criteria.

Hope you get the idea…

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.

"Simple" is a relative term!

- Not as simple as RPG parameter lists.
- Not as simple as REST!
- Simpler than CORBA.

- WSDL is always required (whereas it's optional with REST)
- SOAP is always XML (no other possibilities, unless they are embedded inside an XML document, which can be cumbersome.)

SOAP Action

in SOAP:
- The URL defines which service to call. (Think of it as a program name.)
- There's a special "soap action" keyword in the HTTP keywords, this provides a "verb"
- An XML message is uploaded, processed, then another is downloaded.
- This XML message is the "SOAP message" and is like parameters to a program.

An example of the HTTP transaction would be:

```
POST http://www.scottklement.com/SOAPSRV/CUSTPGM
Content-type: text/xml
SoapAction: "http://scottklement.com/retrieveCust"
```

An XML message in SOAP format would be:
- Sent from consumer to indicate which customer to retrieve
- Returned from provider to indicate the customer details.

SOAP Skeleton

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 parameter info. (Varies by application.)
       . . .
       <soap:Fault>
         (optional) error info.
       </soap:Fault>
       . . .
    </soap:Body>

</soap:Envelope>
```


How Would You Tell the World?

So… Web Services are Essentially Program Calls

- But different, because it's over "the web" (http)
- Every web service is different from the next.
- There are different methods of passing "parameters"
  - *and all of those methods are different from traditional RPG parameters!*

If you wrote a reusable program, and wanted everyone to use it, how would you explain it?

- Which server is it on?
- Which network protocol should you call it with?
- What parameters does it accept? (Sequence, data types, etc)

Would you use?

- Documentation in MS Word? Or PDF? Or a wiki somewhere?
- Maybe you'd teach other programmers in person? (like me!)
- Comments in the code?

SOAP requires the use of a WSDL for this.

WSDL Files

Web Services Description Language (WSDL)

- pronounced "WHIZ-dull"
- Standardized way of documenting a web service.
- A type (schema? flavor?) of XML
- Can be generated by a tool from your parameter list!
- Can be read by a computer program to make your service easy to call
- Almost always used with SOAP. Occasionally also used with POX or REST.

Describes the web service:

- What it does
- What routines it offers (like procedures in a service program)
- Where the service is located (domain name or IP address)
- Protocol to use
- Structure of input/output messages (parameters)

WSDL Skeleton

```
<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.)

Namespaces

SOAP always uses name spaces

- you combine your parameter data (user defined XML) with SOAP XML
- chance of conflicting names!
- name spaces keep them unique
- the URI of a name space isn't connected to over the network, it just guarantees uniqueness *(there's only one w3.org!)*

```
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
               xmlns:morgue="http://example.morgue.com/xml/cadavernet">
    <soap:Body>
        <morgue:CadaverArray>
           <morgue:Body>
              <morgue:lastName>Klement</morgue:lastName>
              <morgue:firstName>Scott</morgue:firstName>
           </morgue:Body>
           <morgue:Body>
              <morgue:lastName>Smith</morgue:lastName>
              <morgue:firstName>Paul</morgue:firstName>
           </morgue:Body>
        </morgue:CadaverArray>
     </soap:Body>
</soap:Envelope>
```


More Namespace Notes

- An xmlns without a prefix designates the "default" namespace.
- It's the URI, not the prefix that identifies the namespace.

  (in the example below, tns:Body and Body are interchangable)
- Until recently, XML-INTO had very poor support for name spaces. (A recent PTF added better namespace capability for 6.1+)

```
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
               xmlns:tns="http://example.morgue.com/xml/cadavernet"
               xmlns="http://example.morgue.com/xml/cadavernet" >
    <soap:Body>
        <CadaverArray>
           <tns:Body>
              <lastName>Klement</lastName>
              <firstName>Scott</firstName>
           </tns:Body>
           <Body>
              <lastName>Smith</lastName>
              <firstName>Paul</firstName>
           </Body>
        </CadaverArray>
     </soap:Body>
</soap:Envelope>
```

Currency Exchange Example

- Free "demo" web service from WebServiceX.net
- The most frequently used sample that's included with HTTPAPI

If you've never used it before, how would you find it?

- Browsing a site like **WebServiceX.net**
- Or **XMethods.net**
- Or **BindingPoint.com**
- Or **RemoteMethods.com**
- Or simply Google for "**(SUBJECT) WSDL**"
  - *such as "Currency Exchange WSDL"*

- Download the WSDL file to learn about the service.
- Almost everyone will use a tool (software) to understand WSDL
- I prefer an open source tool called SoapUI (which is available in both a "free" and "for money/supported" version.)

The WSDL will (of course) tell you what the SOAP messages would look like

Sample SOAP Documents

I've removed the namespace 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 (namespaces here)>
   <SOAP:Body>
      <ConversionRate>
         <FromCurrency>USD</FromCurrency>
         <ToCurrency>EUR</ToCurrency>
      </ConversionRate>
   </SOAP:Body>
</SOAP:Envelope>
```

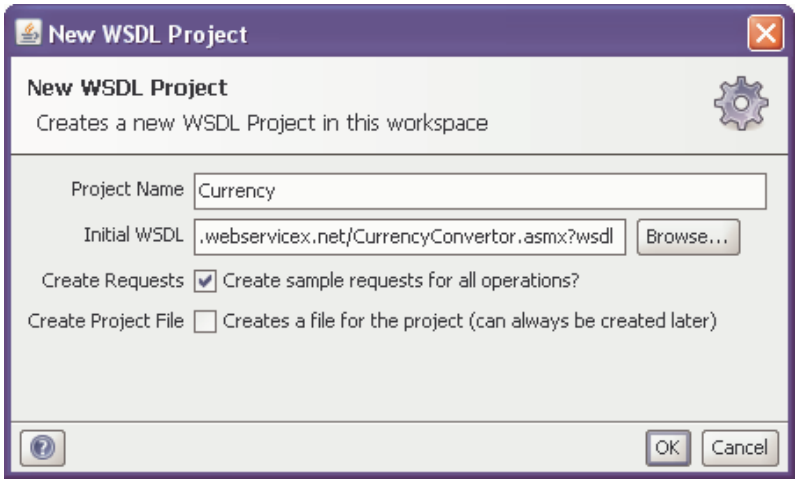
Output Message

```
<?xml version="1.0"?>
<SOAP:Envelope (namespaces here)>
   <SOAP:Body>
      <ConversionRateResponse>
         <ConversionRateResult>0.7207</ConversionRateResult>
      </ConversionRateResponse>
   </SOAP:Body>
</SOAP:Envelope>
```

SoapUI (1/2)

SoapUI is an open source (free of charge) program that you can use to get the SOAP messages you'll need from a WDSL document. http://www.soapui.org

Click ***File / New SOAP Project***



PROJECT NAME

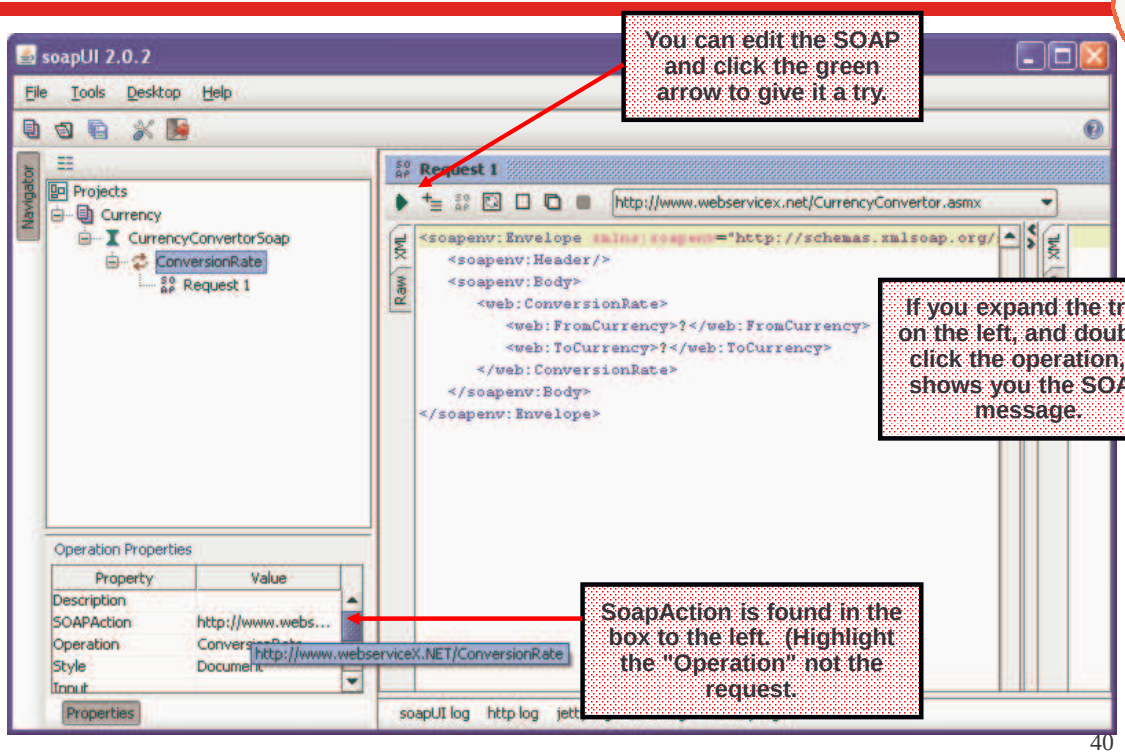
can be any name – use something you'll remember.

INITIAL WSDL

can be either a URL on the web, or a file on your hard drive.

You can use "Browse" to navigate via a standard Windows file dialog.

SoapUI (2/2)



You can edit the SOAP and click the green arrow to give it a try.

If you expand the tree on the left, and double-click the operation, it shows you the SOAP message.

SoapAction is found in the box to the left. (Highlight the "Operation" not the request.

RPG as a Web Service Provider

Presented by

Scott Klement

http://www.scottklement.com

"If you give someone a program, you will frustrate them for a day; if you teach them how to program, you will frustrate them for a lifetime."

Different Approaches to Providing

Included with the IBM i operating system (starting with V5R4) is an optionally-installed licensed program for providing/consuming web services. This is called "Integrated Web Services" (or IWS for short)

IWS is easy, but has limitations:

- Maximum of 7 parameters (but they can be data structures or arrays)
- Can't nest arrays inside arrays
- Supports XML or JSON, but you don't have much control over the format
- Very limited options for security
- Generates Java/C code under the covers, doesn't always perform well
- At this time, IWS does not support WSDL for REST web services.

You can also write your own web services from the ground up using the regular Apache HTTP server:

- Gives you complete control
- Performs great
- Requires more knowledge/work of web service technologies such as XML and JSON

Example of IWS Web Service

For this example, we'll write a SOAP web service that accepts a customer number as input, and returns a customer's address.

First we'll write an RPG program to do that.

- Parameters are used to get the input and send the output
- If there's an error, we send an error message to the caller (Using QMHSNDPM API – equivalent to the CL SNDPGMMSG command that you may be more familiar with.)
- The RPG compiler can generate an XML document called PCML that has information about the program's parameters.
- The IWS will then be used to provide our RPG as a web service.

Get Customer Example (1 of 2)

```
H DFTACTGRP(*NO) ACTGRP('SOAP') PGMINFO(*PCML: *MODULE)

FCUSTFILE  IF   E           K DISK    PREFIX('CUST.')

D CUST          E DS                  qualified
D                                     extname(CUSTFILE)

D GETCUST         PR                  ExtPgm('GETCUST')
D   CustNo                            like(Cust.Custno)
D   Name                              like(Cust.Name)
D   Street                            like(Cust.Street)
D   City                              like(Cust.City)
D   State                             like(Cust.State)
D   Postal                            like(Cust.Postal)
D GETCUST         PI
D   CustNo                            like(Cust.Custno)
D   Name                              like(Cust.Name)
D   Street                            like(Cust.Street)
D   City                              like(Cust.City)
D   State                             like(Cust.State)
D   Postal                            like(Cust.Postal)
```

PCML with parameter info will be embedded in the module and program objects.

This PREFIX causes the file to be read into the CUST data struct.

When there's no P-spec, the PR/PI acts the same as *ENTRY PLIST.

Get Customer Example (2 of 2)

```
/free
   chain CustNo CUSTFILE;
   if not %found;
      msgdta = 'Customer not found.';
      QMHSNDPM( 'CPF9897': 'QCPFMSG   *LIBL'
              : msgdta: %len(msgdta): '*ESCAPE'
              : '*PGMBDY': 1: MsgKey: err );
   else;
      Custno = Cust.Custno;
      Name   = Cust.name;
      Street = Cust.Street;
      City   = Cust.City;
      State  = Cust.State;
      Postal = Cust.Postal;
   endif;
   *inlr = *on;
/end-free
```

This API is equivalent to the CL SNDPGMMSG command, and causes my program to end with an exception ("halt")

When there are no errors, I simply return my output via the parameter list. IWS takes care of the XML for me!

Compiling with PCML Support

The IWS will generate XML or JSON automatically. For that to work, it needs to know about the parameters in the program. The RPG compiler can automatically generate parameter information in (yet another) XML format known as PCML.

On the compile command:

- `PGMINFO(*PCML:*MODULE)` will store the PCML inside the module/pgm/srvpgm object.
- `PGMINFO(*PCML) INFOSTMF('/ifs/path/here')` will store the PCML as an IFS file

For example:

```
CRTBNDRPG PGM(SKWEBSRV/GETCUST) PGMINFO(*PCML:*MODULE)
```

When you use *MODULE, you can also put it on the CTL-OPT or H-Spec. This is what I recommend:

```
     H PGMINFO(*PCML:*MODULE)      (fixed format)
or
     ctl-opt pgminfo(*pcml:*module);    (free format)
```

PCML Example

Here's an example of what the PCML looks like.

You don't have to ever see this or understand if you don't want to, but sometimes it helps to understand what's happening under the covers.

This is how the IWS will know what the program's parameters are:

```
<pcml version="4.0">
      <program name="GETCUST" path="/QSYS.LIB/SKWEBSRV.LIB/GETCUST.PGM">
      <data name="CUSTNO" type="zoned" length="5" precision="0"
            usage="inputoutput" />
      <data name="NAME" type="char" length="30" usage="inputoutput" />
      <data name="STREET" type="char" length="30" usage="inputoutput" />
      <data name="CITY" type="char" length="20" usage="inputoutput" />
      <data name="STATE" type="char" length="2" usage="inputoutput" />
      <data name="POSTAL" type="char" length="10" usage="inputoutput" />
   </program>
</pcml>
```


The Wizarding World

Once this program has been compiled and placed in a library, we can ask the IWS to provide it as a web service.

This is done by invoking a Wizard in the HTTP Server (Powered by Apache) *ADMIN server (in recent releases this is inside *IBM Navigator for i*)

- This is included in the operating system, but is an optional component
- If it's not already installed, instructions can be found here:

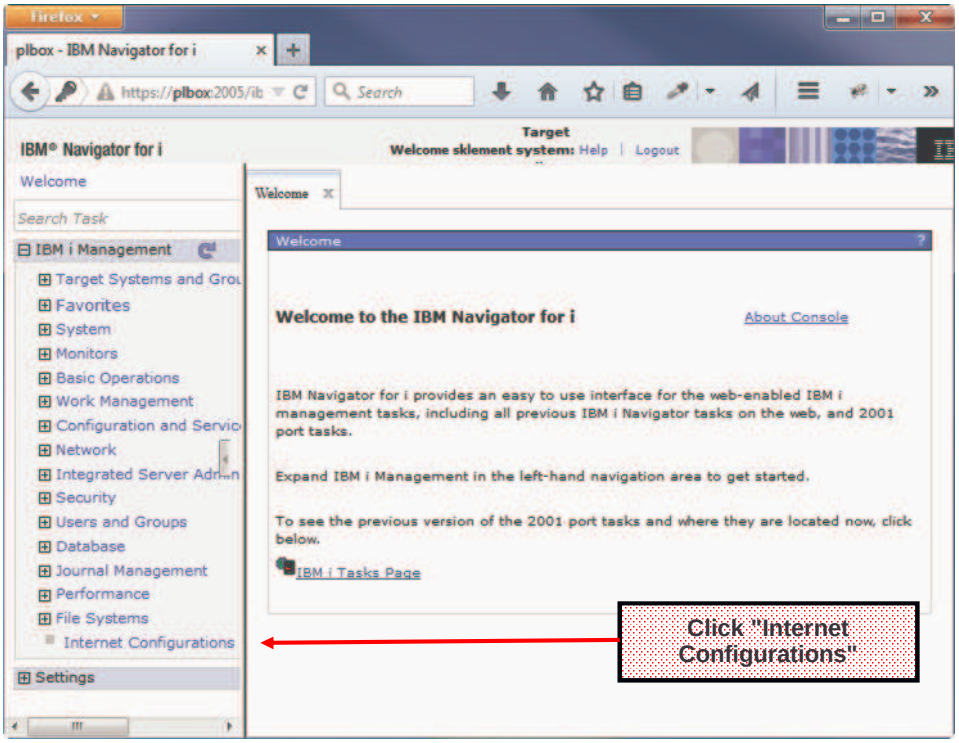
http://www-01.ibm.com/support/knowledgecenter/ssw_ibm_i_71/rzaie/rzaieinstalling.htm

- If installed but not running, you can start it with:

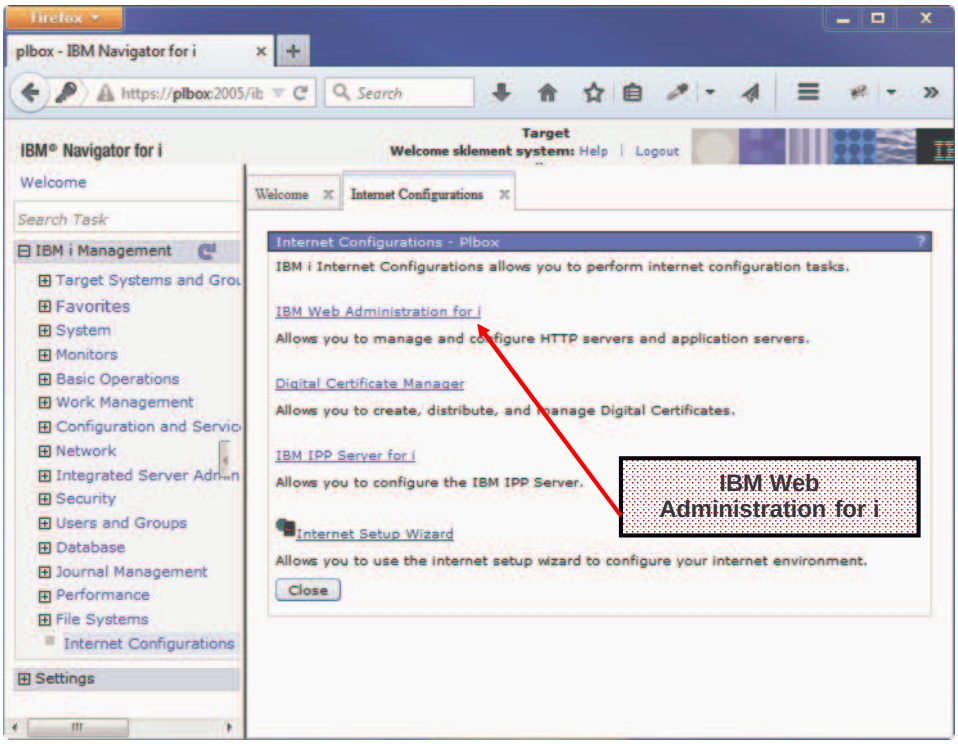
```
STRTCPSVR SERVER(*HTTP) HTTPSVR(*ADMIN)
```

- To use it, connect your browser to http://your-server:2001

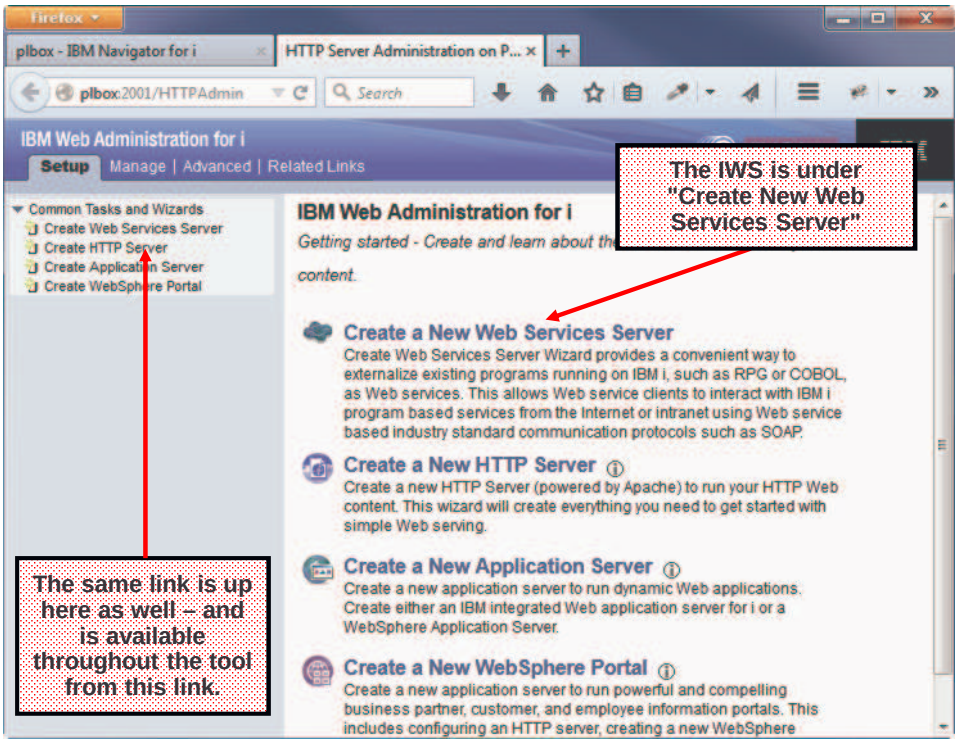
IBM Navigator for i



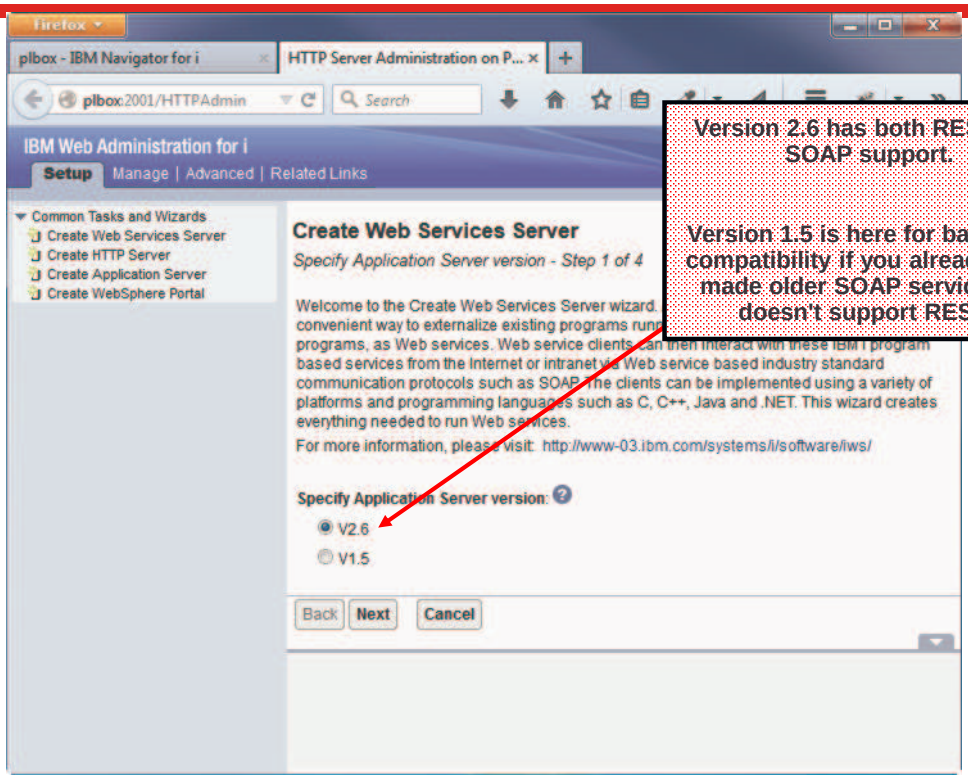
Internet Configurations



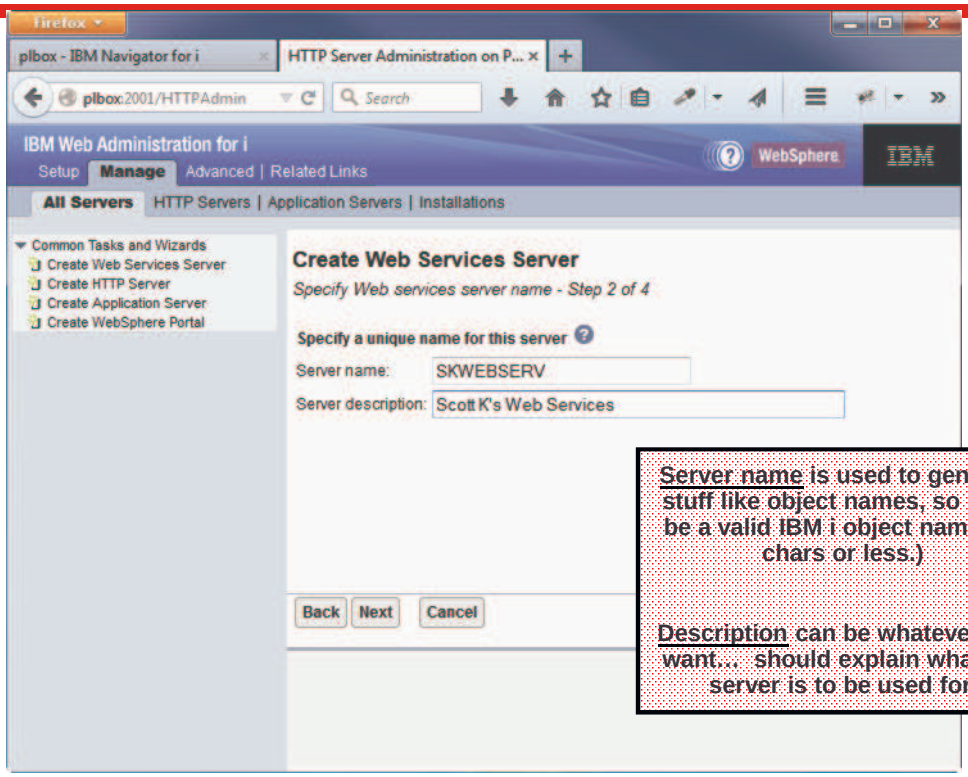
Web Administration for i



Create IWS Server (1 of 4)



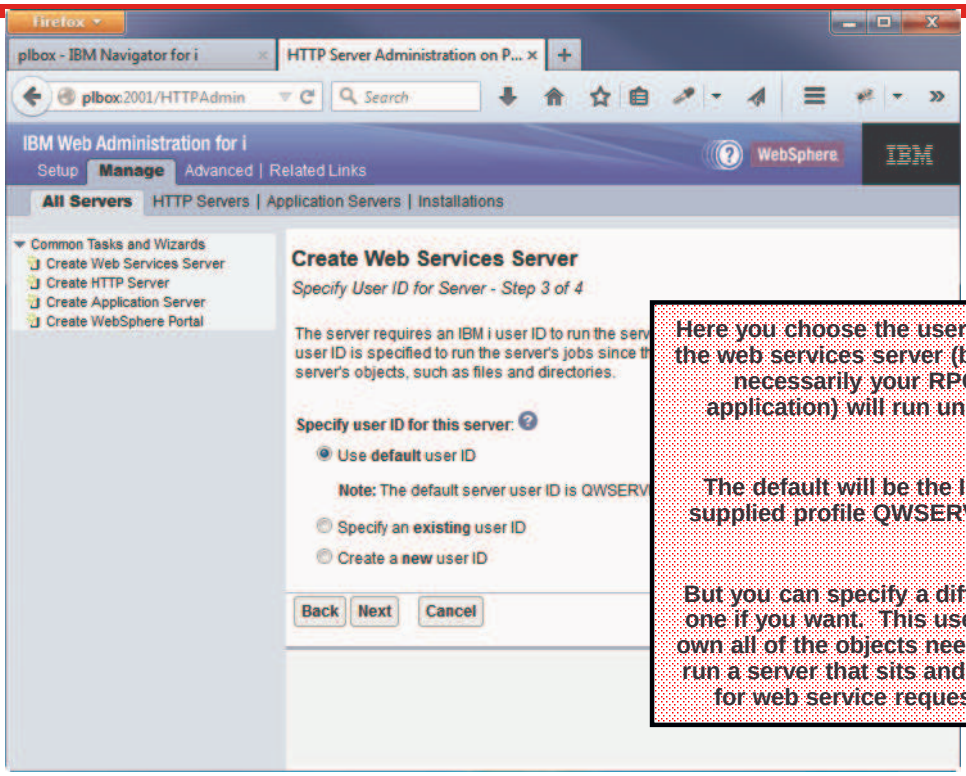
Create IWS Server (2 of 4)



Server name is used to generate stuff like object names, so must be a valid IBM i object name (10 chars or less.)

Description can be whatever you want... should explain what the server is to be used for.

Create IWS Server (3 of 4)

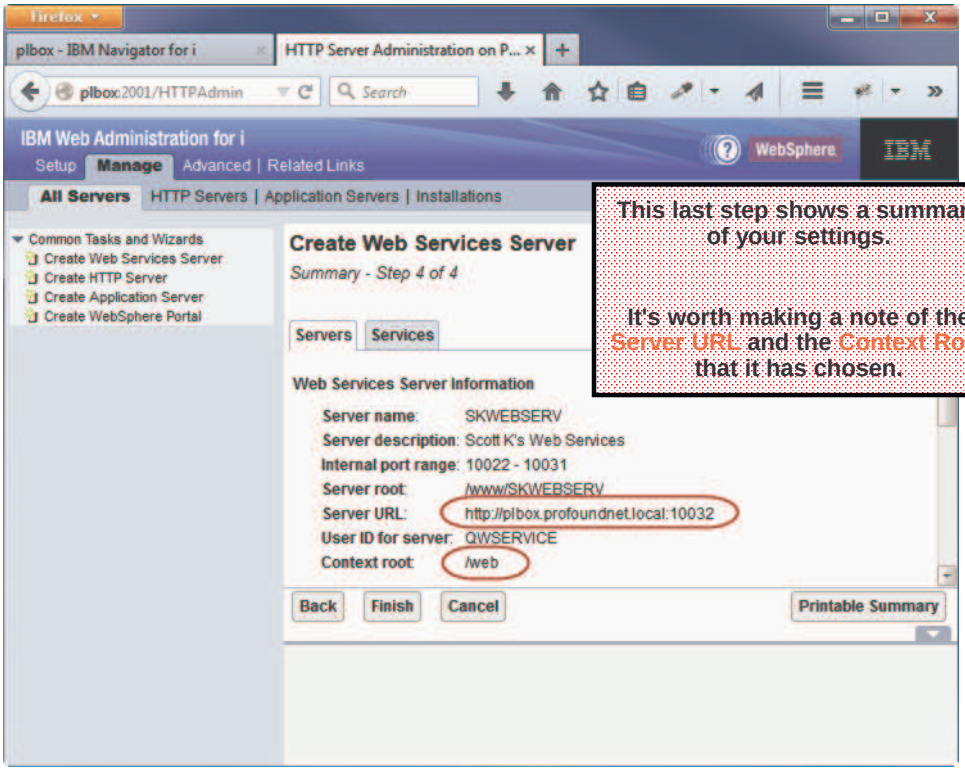


Here you choose the userid that the web services server (but not necessarily your RPG application) will run under.

The default will be the IBM-supplied profile QWSERVICE.

But you can specify a different one if you want. This user will own all of the objects needed to run a server that sits and waits for web service requests.

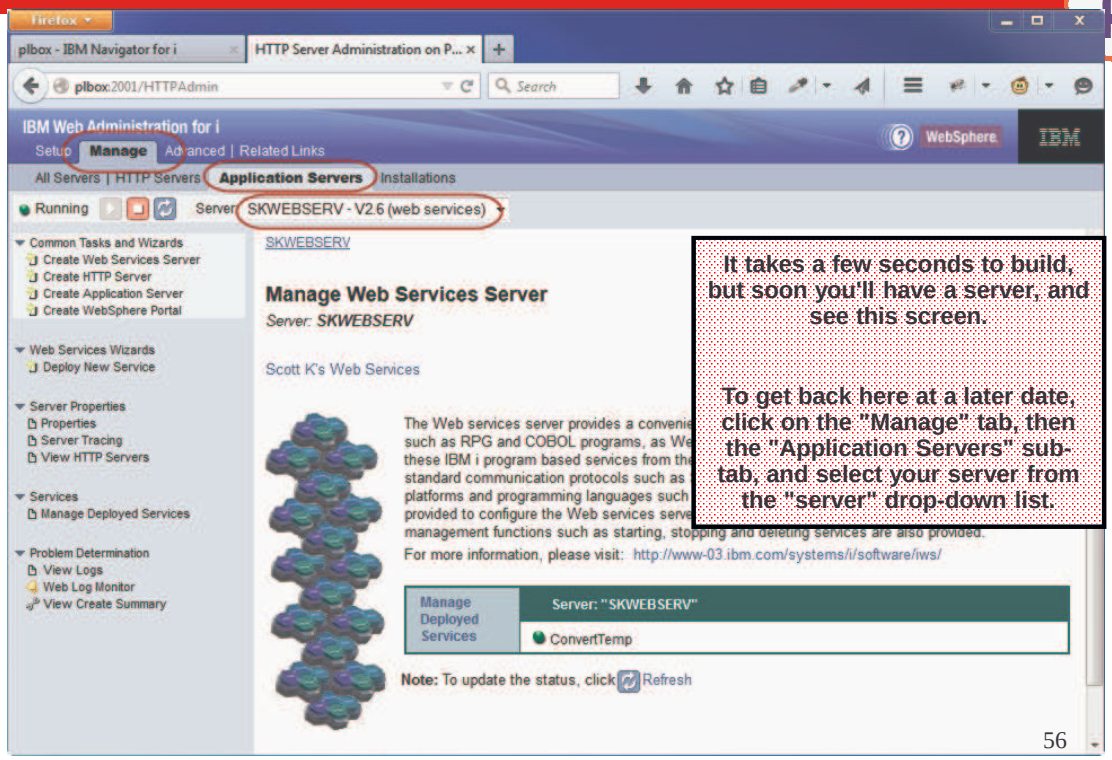
Create IWS Server (4 of 4)



This last step shows a summary of your settings.

It's worth making a note of the Server URL and the Context Root that it has chosen.

We Now Have a Server!



It takes a few seconds to build, but soon you'll have a server, and see this screen.

To get back here at a later date, click on the "Manage" tab, then the "Application Servers" sub-tab, and select your server from the "server" drop-down list.

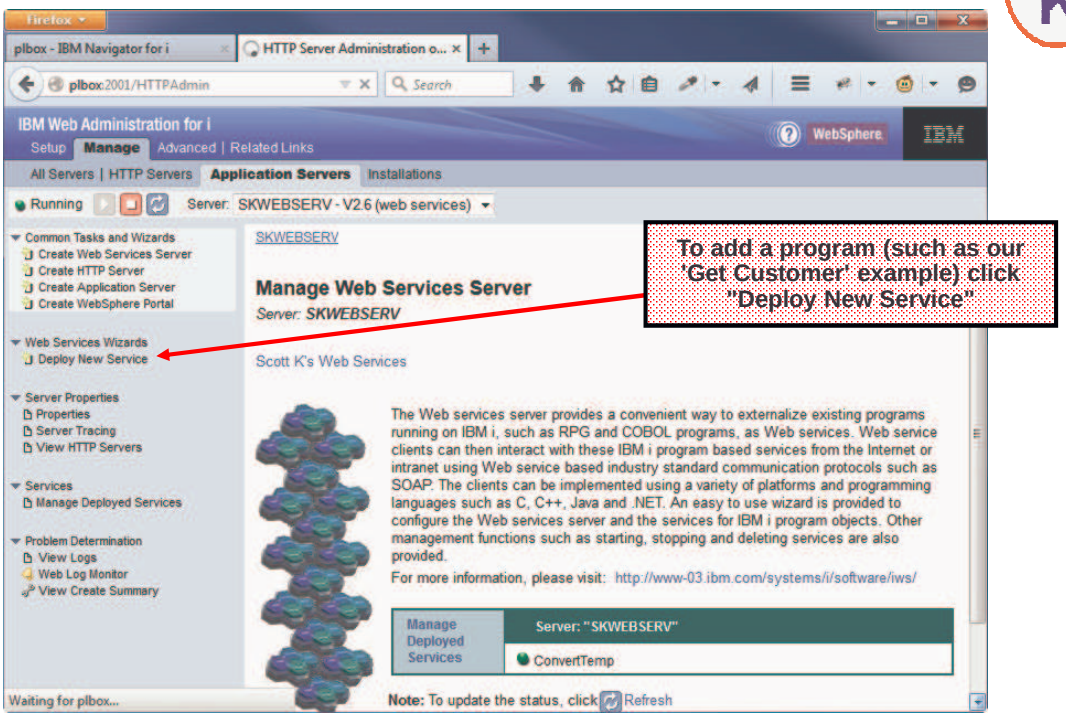
Now What?

Now that we have a web services server, we can add (or "deploy" is the official term) web services… i.e. programs/subprocedures that can be called as web services.

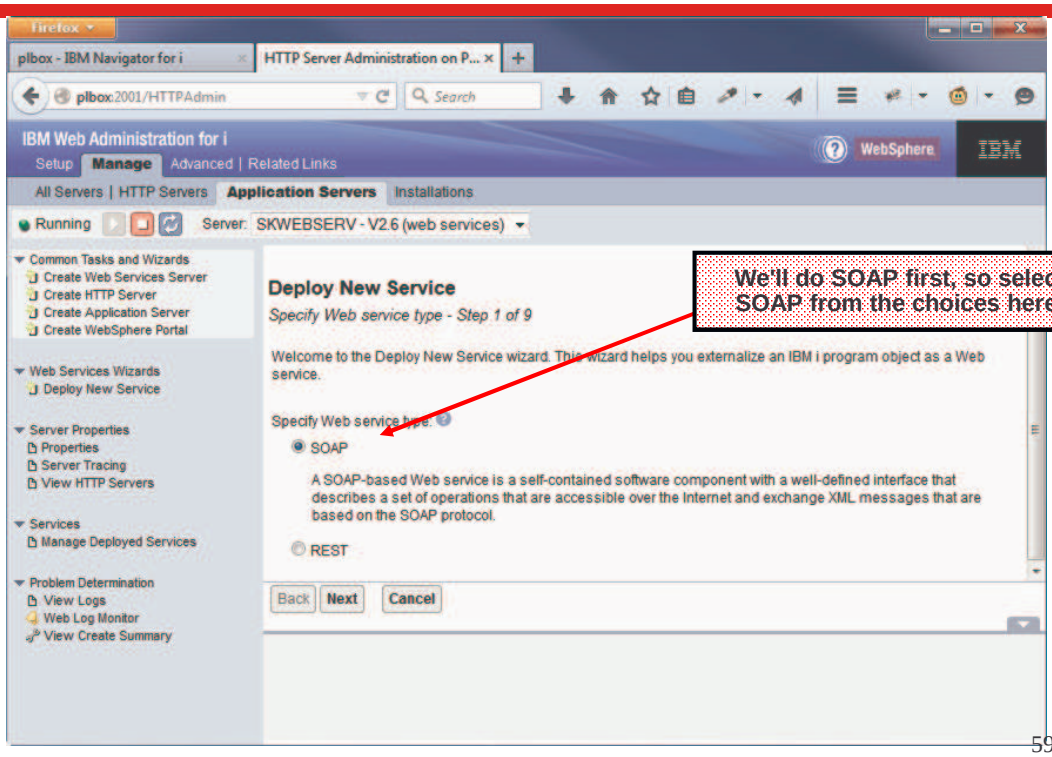
- One server can handle many services (programs/procedures)
- The same server can handle both REST and SOAP services (version 2.6+)
- IBM provides a "ConvertTemp" service as an example.

The "manage deployed services" button can be used to stop/start individual services as well as add/remove them.

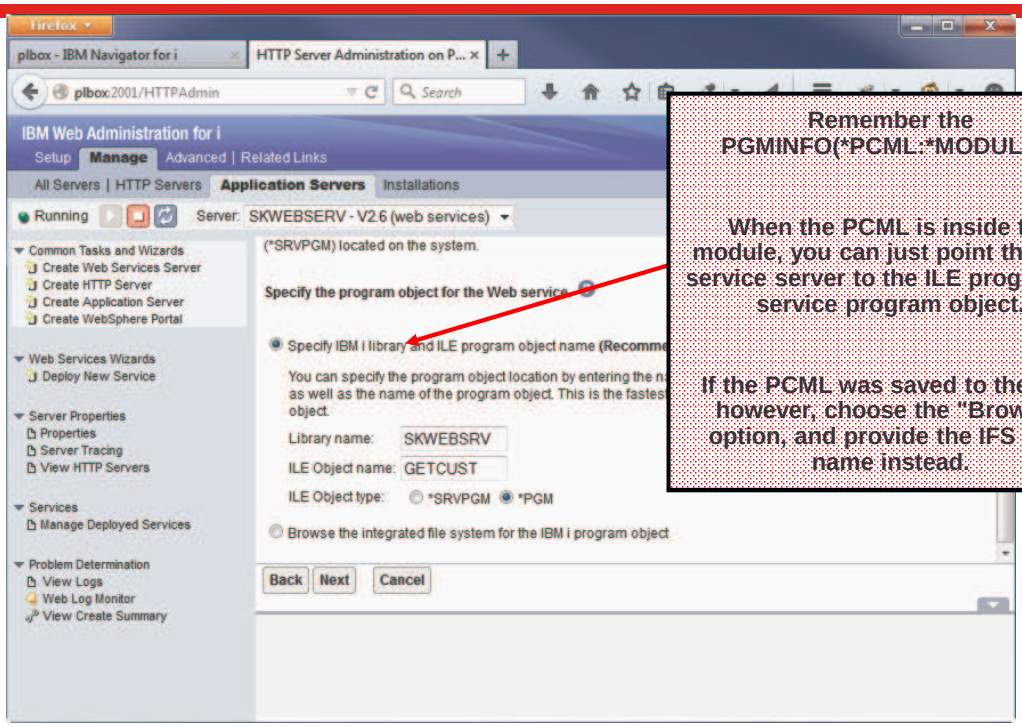
GETCUST as SOAP Service



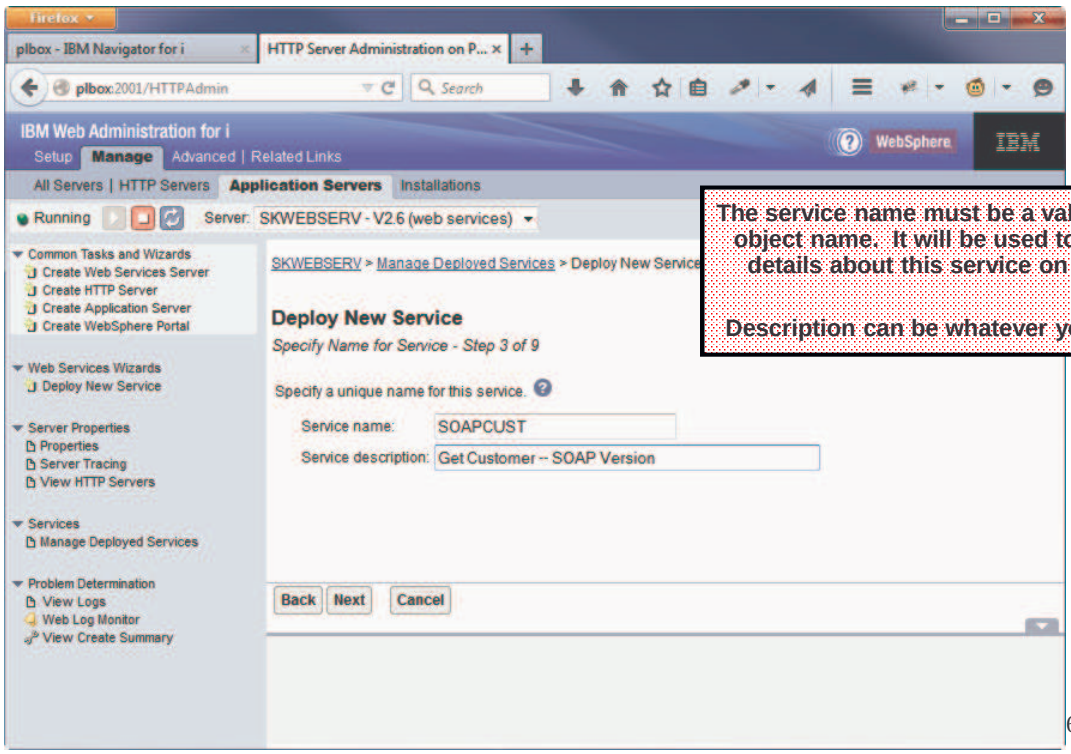
SOAP Example (1 of 9)



SOAP Example (2 of 9)



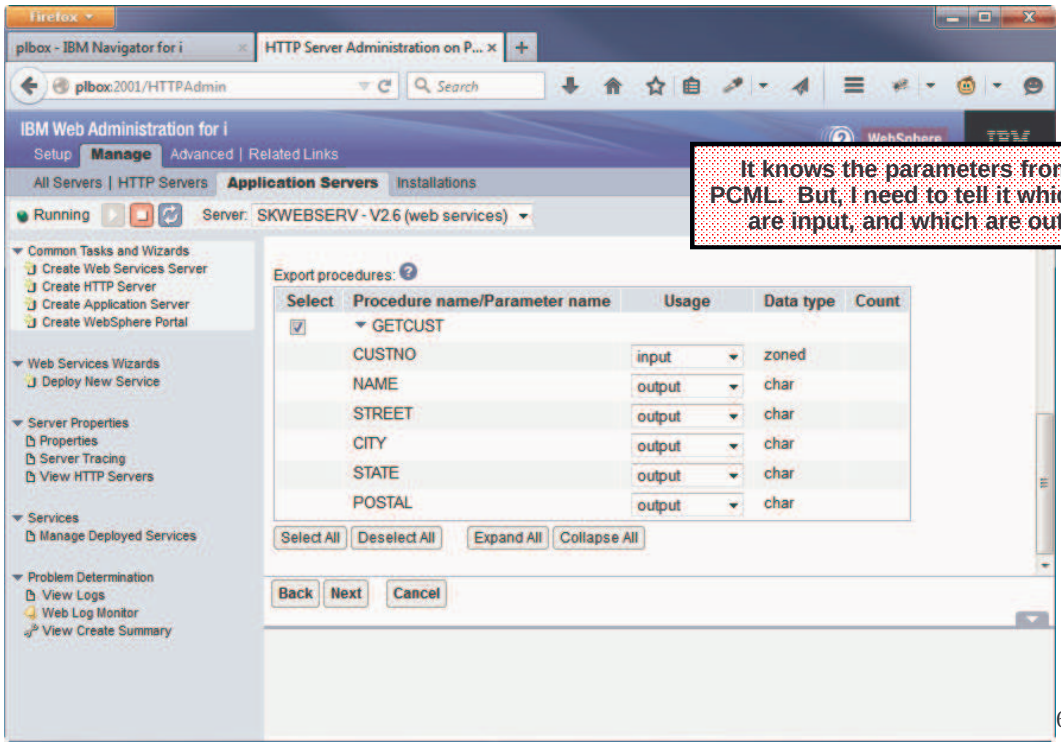
SOAP Example (3 of 9)



The service name must be a valid IBM i object name. It will be used to store details about this service on disk.

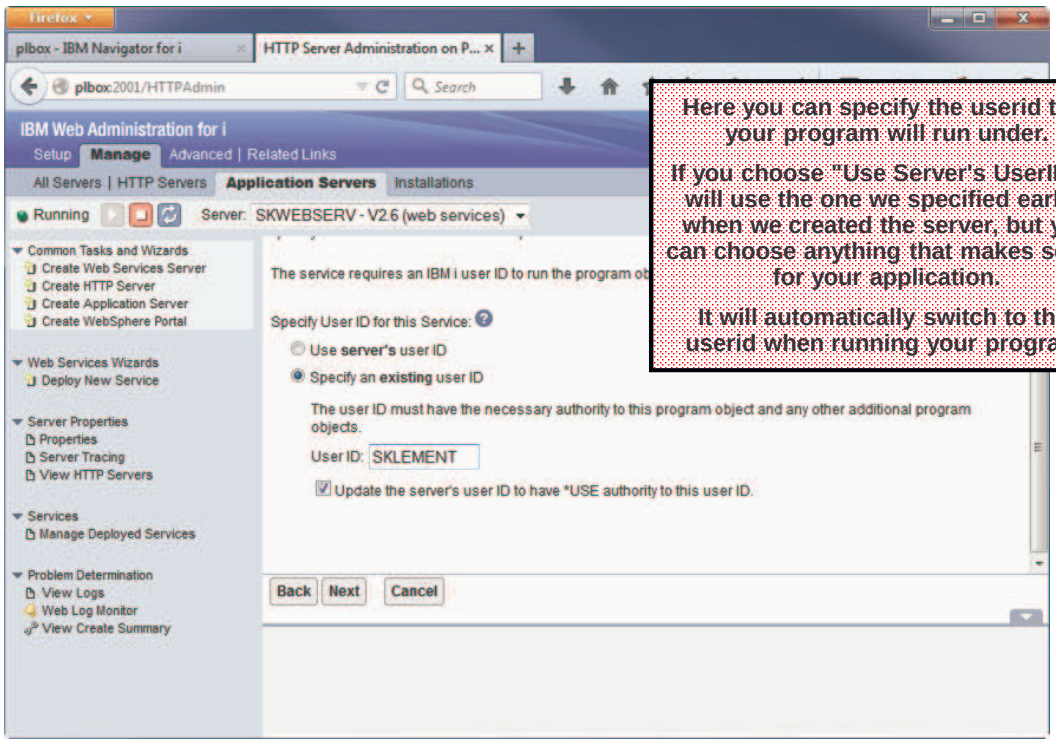
Description can be whatever you like.

SOAP Example (4 of 9)



It knows the parameters from the PCML. But, I need to tell it which ones are input, and which are output.

SOAP Example (5 of 9)

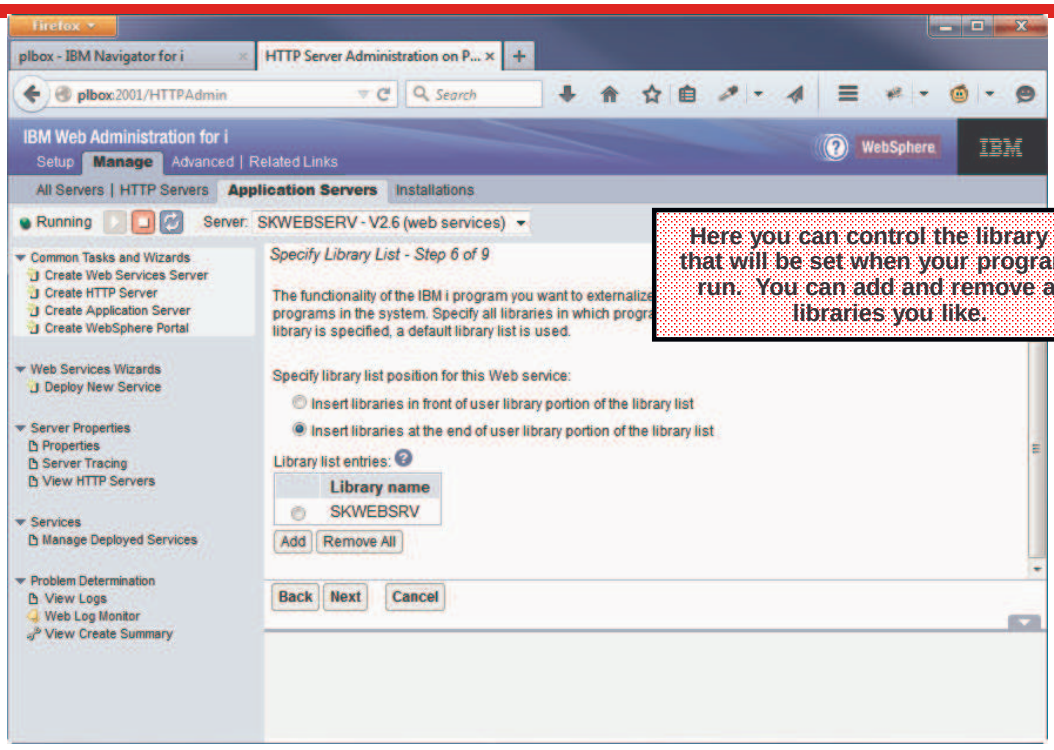


Here you can specify the userid that your program will run under.

If you choose "Use Server's UserID" it will use the one we specified earlier when we created the server, but you can choose anything that makes sense for your application.

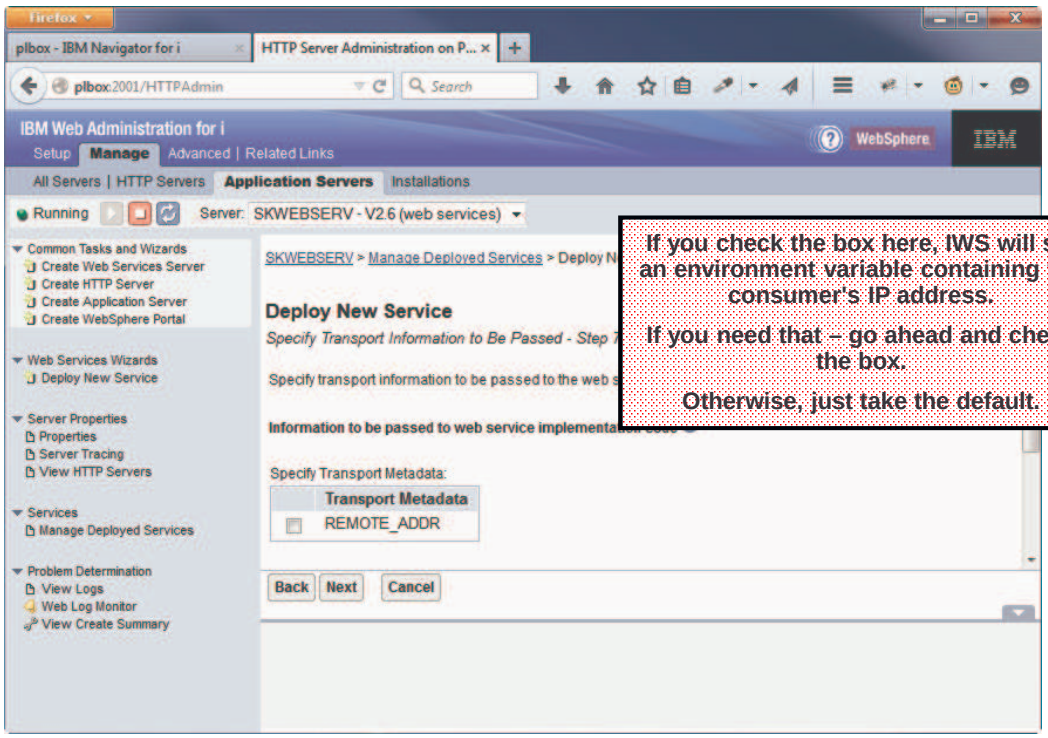
It will automatically switch to this userid when running your program.

SOAP Example (6 of 9)



Here you can control the library list that will be set when your program is run. You can add and remove any libraries you like.

SOAP Example (7 of 9)

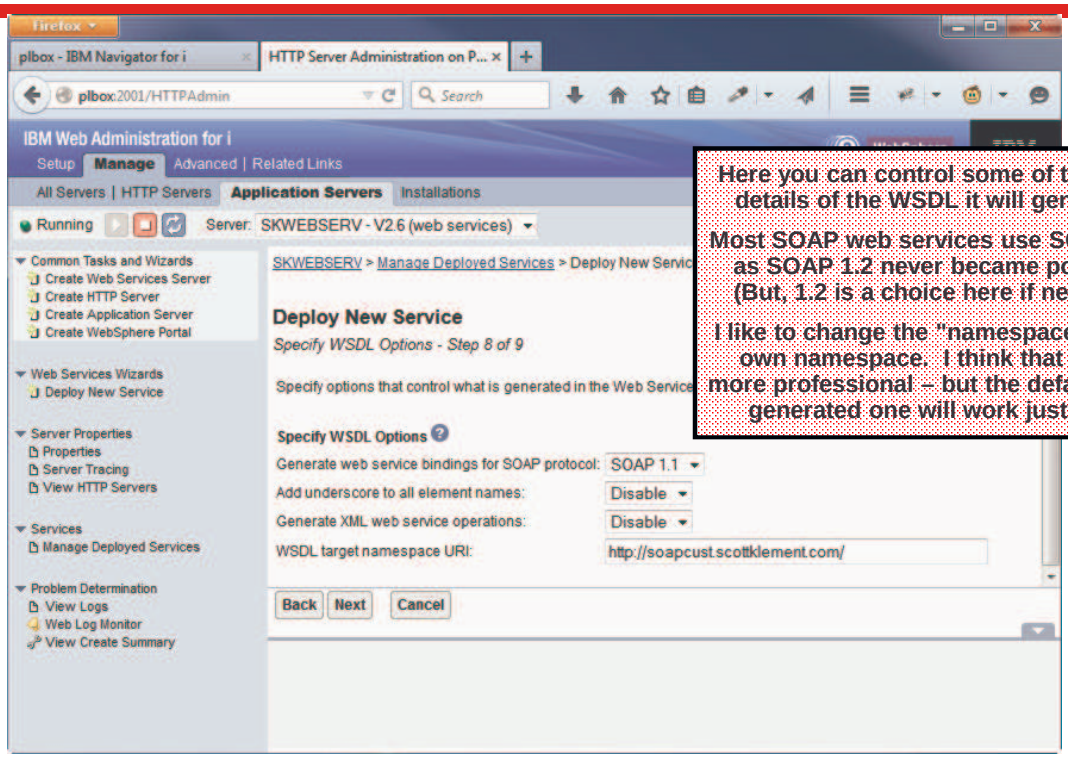


If you check the box here, IWS will set an environment variable containing the consumer's IP address.

If you need that – go ahead and check the box.

Otherwise, just take the default.

SOAP Example (8 of 9)

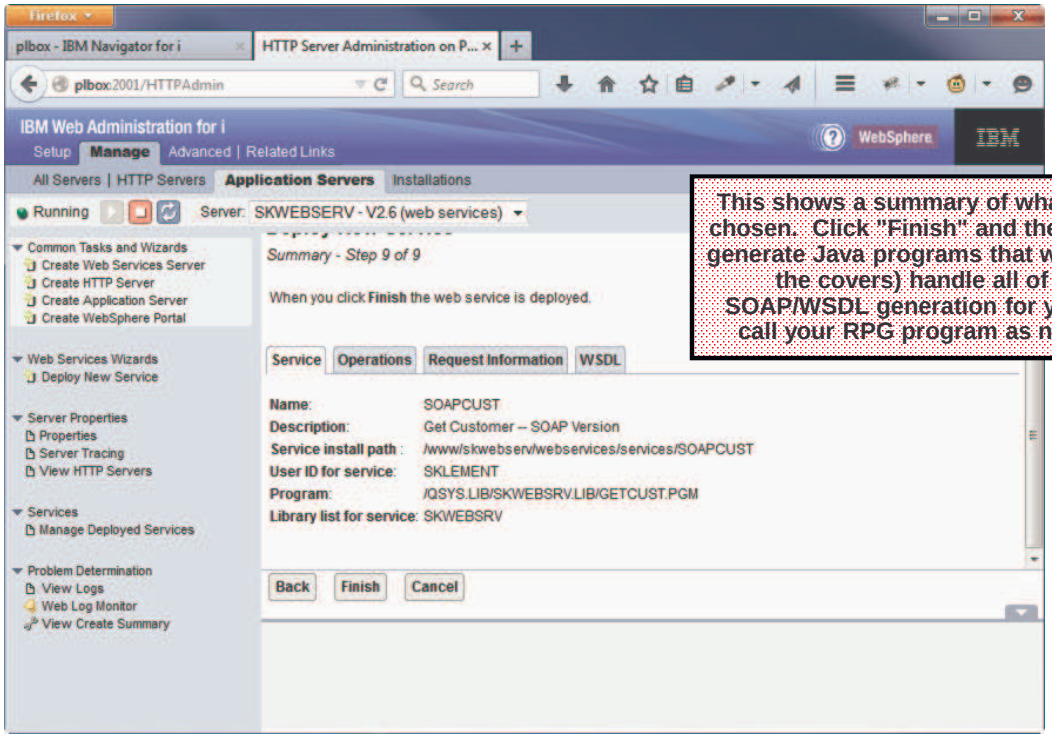


Here you can control some of the finer details of the WSDL it will generate.

Most SOAP web services use SOAP 1.1, as SOAP 1.2 never became popular. (But, 1.2 is a choice here if needed.)

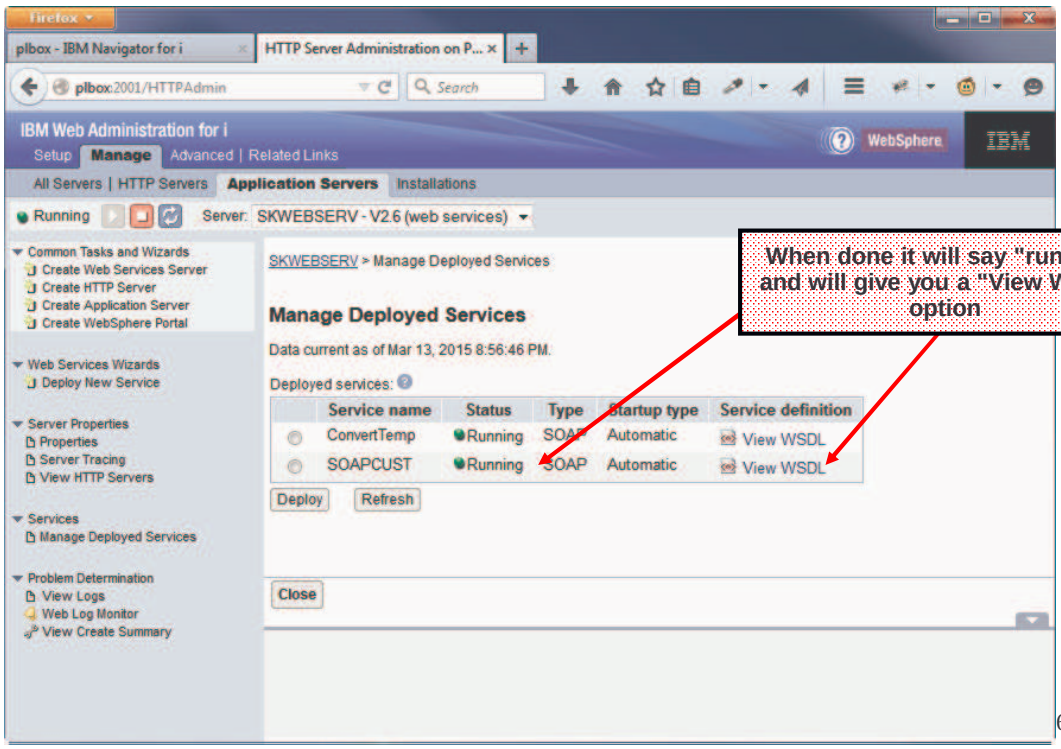
I like to change the "namespace" to my own namespace. I think that looks more professional – but the default IBM-generated one will work just fine.

SOAP Example (9 of 9)



This shows a summary of what you've chosen. Click "Finish" and the IWS will generate Java programs that will (under the covers) handle all of the SOAP/WSDL generation for you, and call your RPG program as needed.

Manage Screen



When done it will say "running" and will give you a "View WSDL" option

Expected SOAP Input

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:a="http://soapcust.scottklement.com/">
   <soap:Body>
      <a:getcust>
         <arg0>
            <CUSTNO>495</CUSTNO>
         </arg0>
      </a:getcust>
   </soap:Body>
</soap:Envelope>
```

This is the input parameter list sent from the consumer to the provider.

- Notice that it matches the skeleton from earlier, but with the details filled in
- Is generated from the (more complex) WSDL document.
- 'arg0' is like a data structure with one subfield, 'CUSTNO'
- The term for putting params into a DS like this are referred to as 'wrapped' (because an XML tag wraps the whole parameter list)
- Originally, there were other styles, but today almost all SOAP services use the 'wrapped' style.

Expected SOAP Output

```
<soap:Envelope
   xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Body>
      <a:getcustResponse
         xmlns:a="http://soapcust.scottklement.com/">
         <return>
            <STATE>FL</STATE>
            <STREET>1100 N.W. 53RD STREET</STREET>
            <POSTAL>33064-2121</POSTAL>
            <CITY>POMPANO BEACH</CITY>
            <NAME>ACME FOODS</NAME>
         </return>
      </a:getcustResponse>
   </soap:Body>
</soap:Envelope>
```

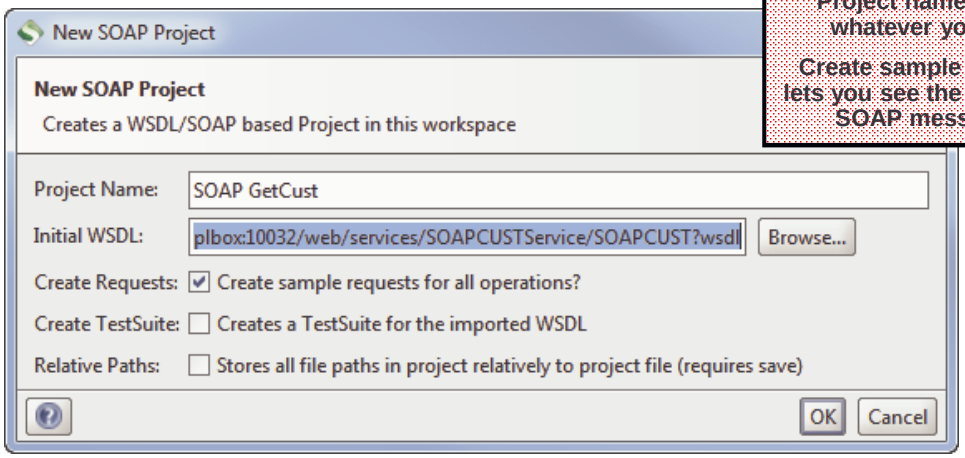
This is the output parameter sent back from the provider to the consumer.

- Returns the details about the customer.
- Notice that the output is also in 'wrapped' style.
- Also generated from the WSDL document

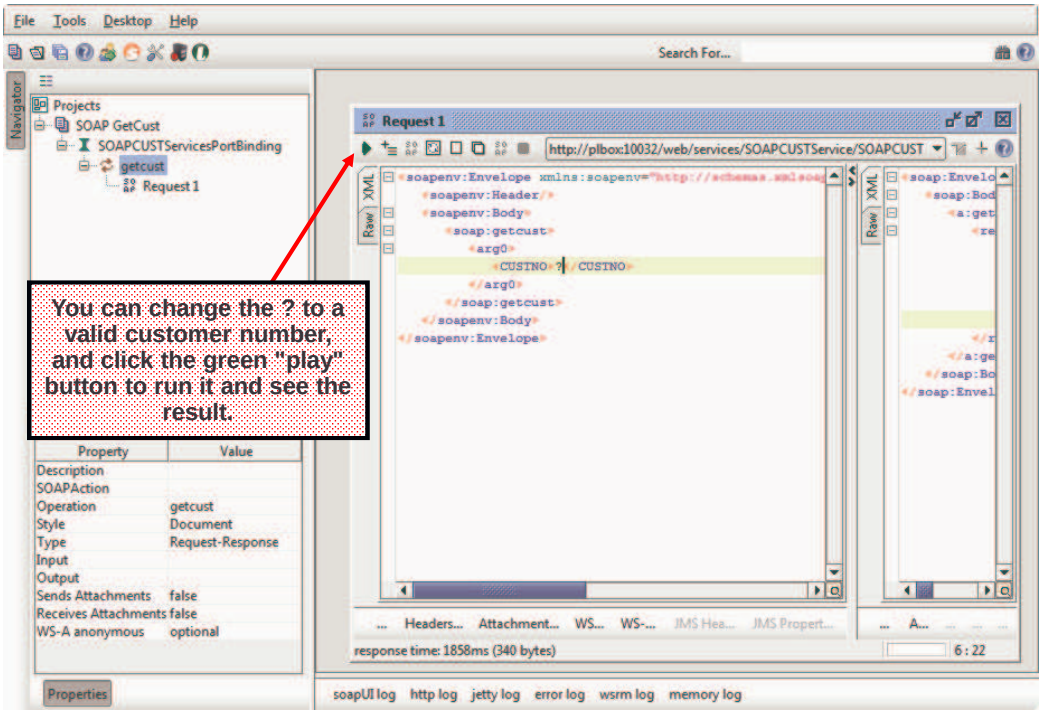
Testing it Out with SOAPUI

Although the IWS has it's own SOAP testing tool included, I prefer a different tool called SOAPUI. It's available in both commercial and free versions, and is much more powerful than the built-in tool. (Link at the end of the handout.)

To try it in SOAPUI, click File / New SOAP project, and copy/paste the WSDL link into the "initial WSDL" field.



SOAPUI Results



After SOAP, I Need a REST

Remember that REST (sometimes called 'RESTful') web services differ from SOAP in that:

- the URL points to a "noun" (or "resource")
- the HTTP method specifies a "verb" like GET, POST, PUT or DELETE. (Similar to a database Create, Read, Update, Delete…)
- REST sounds nicer than CRUD, haha.

IWS structures the URL like this:

```
http://address:port/context-root/root-resource/path-template
```

- context-root = Distinguishes from other servers. The default context-root is /web/services, but you can change this in the server properties.
- root-resource = identifies the type of resource (or "noun") we're working with. In our example, we'll use "/cust" to identify a customer. The IWS will also use this to determine which program to run.
- path-template = identifies the variables/parameters that distinguish this noun from others. In our example, it'll be the customer number.

After SOAP, I Need a REST

For our example, we will use this URL:

```
http://address:port/web/services/cust/495
```

Our URL will represent a customer record. Then we can:

- GET <url> the customer to see the address.
- potentially POST <url> the customer to create a new customer record
- potentially PUT <url> the customer to update an existing customer record
- potentially DELETE <url> to remove the customer record.

Though, in this particular example, our requirements are only to retrieve customer details, so we won't do all four possible verbs, we'll only do GET.

That means in IWS terminology:

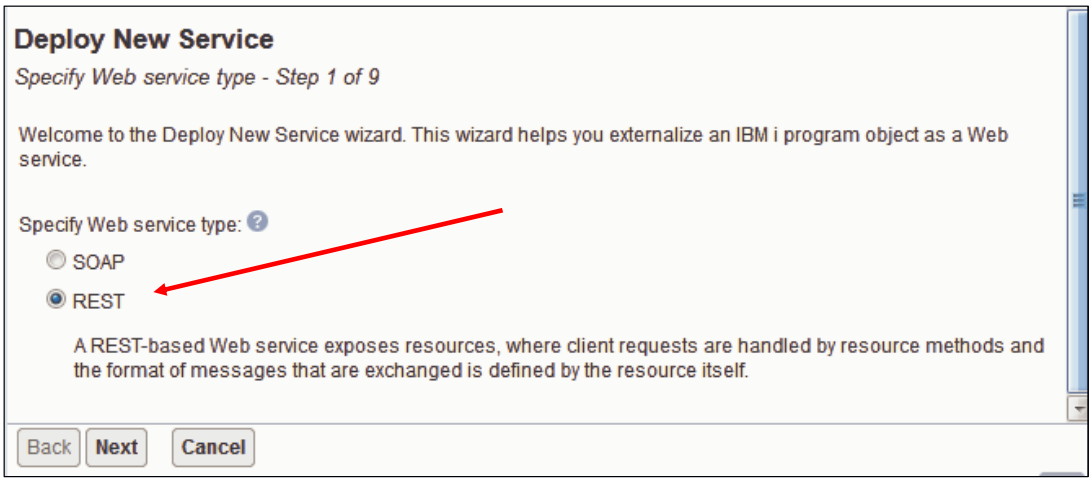
- /web/services is the context root.
- /cust is the root resource (and will point to our GETCUST program)
- /495 (or any other customer number) is the path template.

With that in mind, we're off to see the wizard… the wonderful wizard of REST.

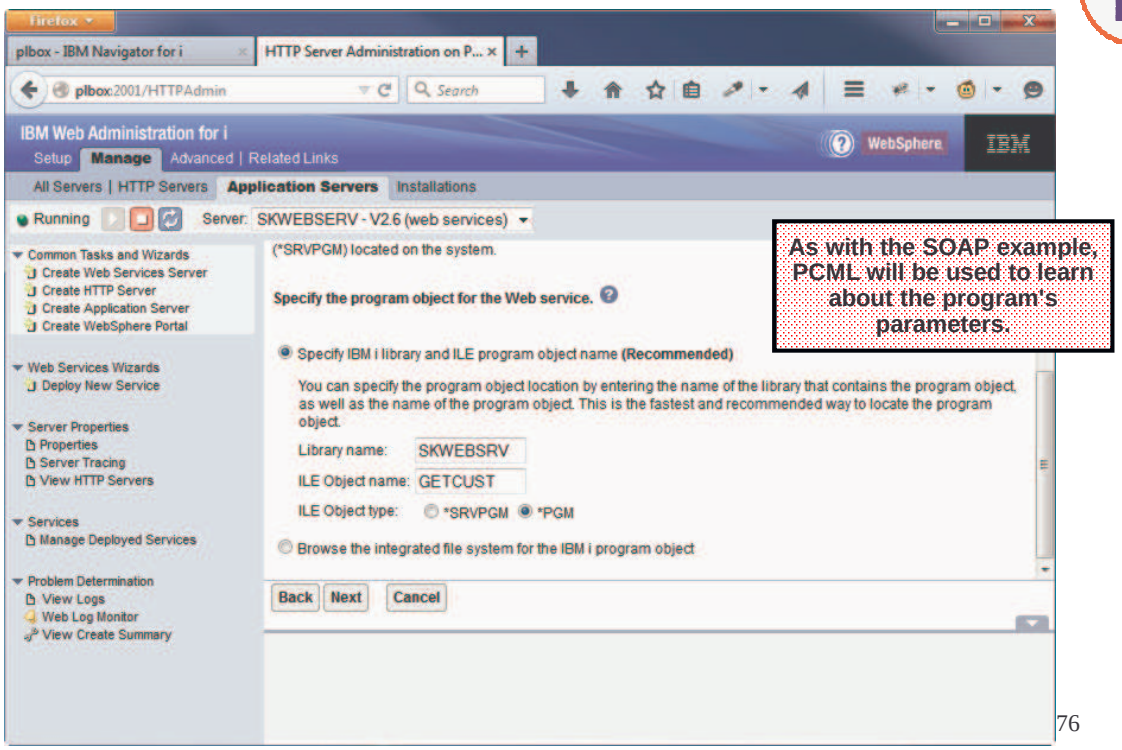
REST Wizard (1 of 9)

Now I'd like to do the same web service as REST instead of SOAP. (The IWS also supports REST in the latest versions.)

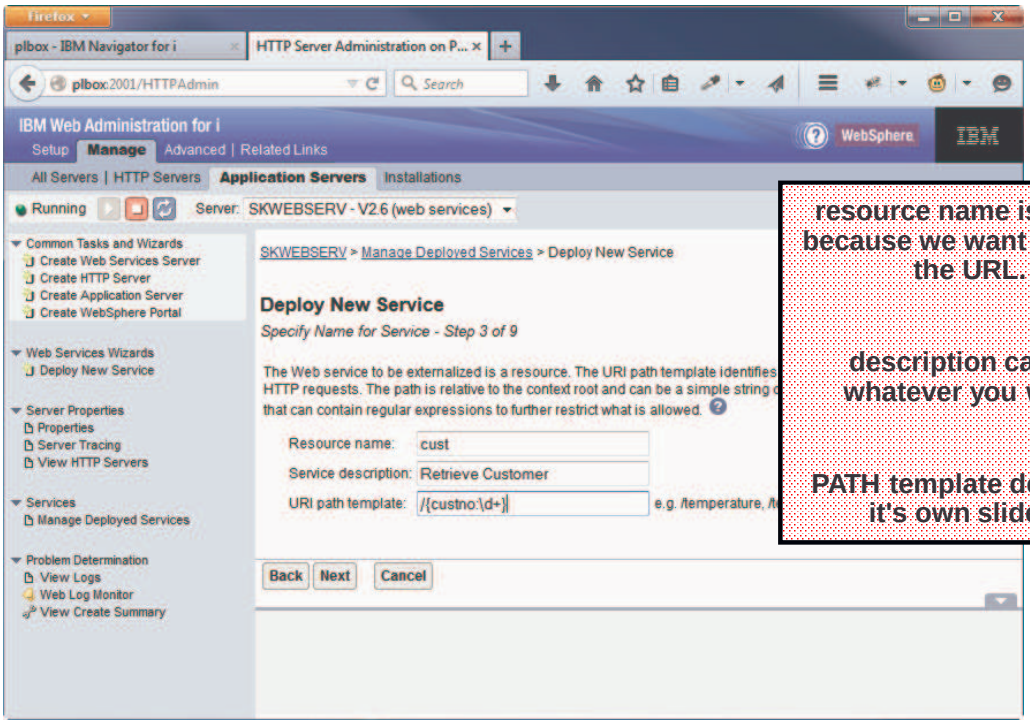
To do that, I'll click 'Deploy New Service' again, this time choosing REST.



REST Wizard (2 of 9)



REST Wizard (3 of 9)



resource name is 'cust', because we want /cust/ in the URL.

description can be whatever you want.

PATH template deserves it's own slide ☺

Path Templates

You can make your URL as sophisticated as you like with a REST service. For example:

- Maybe there are multiple path variables separated by slashes
- Maybe they allow only numeric values
- Maybe they allow only letters, or only uppercase letters, or only lowercase, or both letters and numbers
- maybe they have to have certain punctuation, like slashes in a date, or dashes in a phone number.

Path templates are how you configure all of that. They have a syntax like:

```
{ identifier : regular expression }
```

- The identifier will be used later to map the variable into a program's parameter.
- The regular expression is used to tell IWS what is allowed in the parameter

Path Template Examples

For our example, we want /495 (or any other customer number) in the URL, so we do:

/{custno:\d+} identifier=custno, and regular expression \d+ means
\d = any digit, + = one or more

As a more sophisticated example, consider a web service that returns inventory in a particular warehouse location. The path template might identify a warehouse location in this syntax

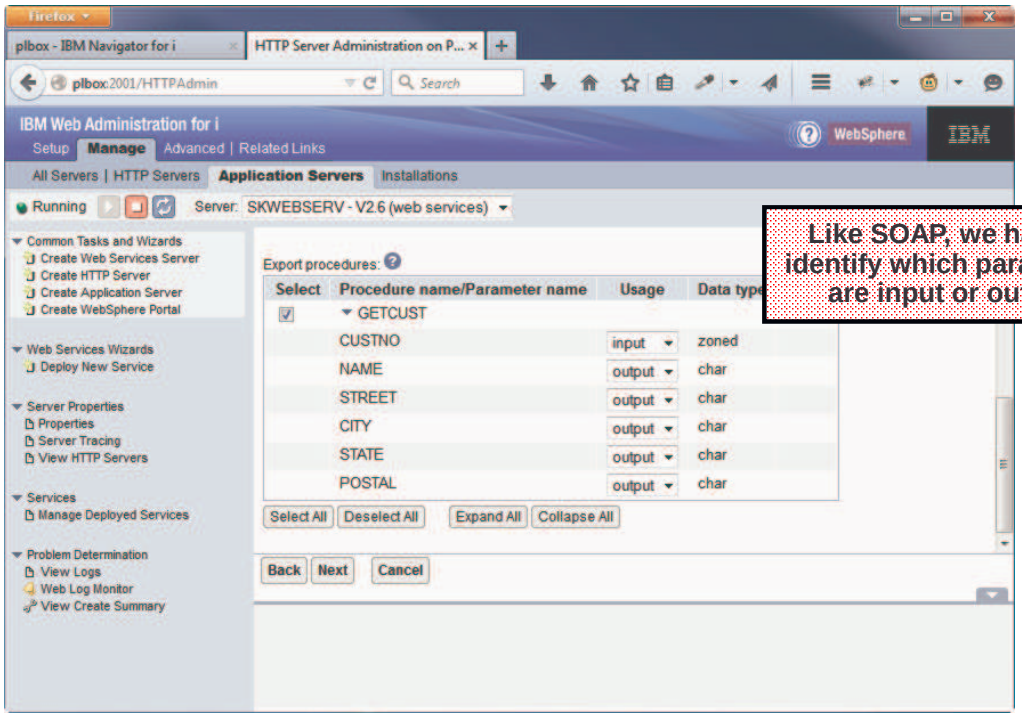
```
/Milwaukee/202/Freezer1/B/12/C
```

These identify City, Building, Room, Aisle, Slot and Shelf. The path template might be
/{city:\w+}/{bldg:\d+}/{room:\w+}/{aisle:[A-Z]}/{slot:\d\d}/{shelf:[A-E]}

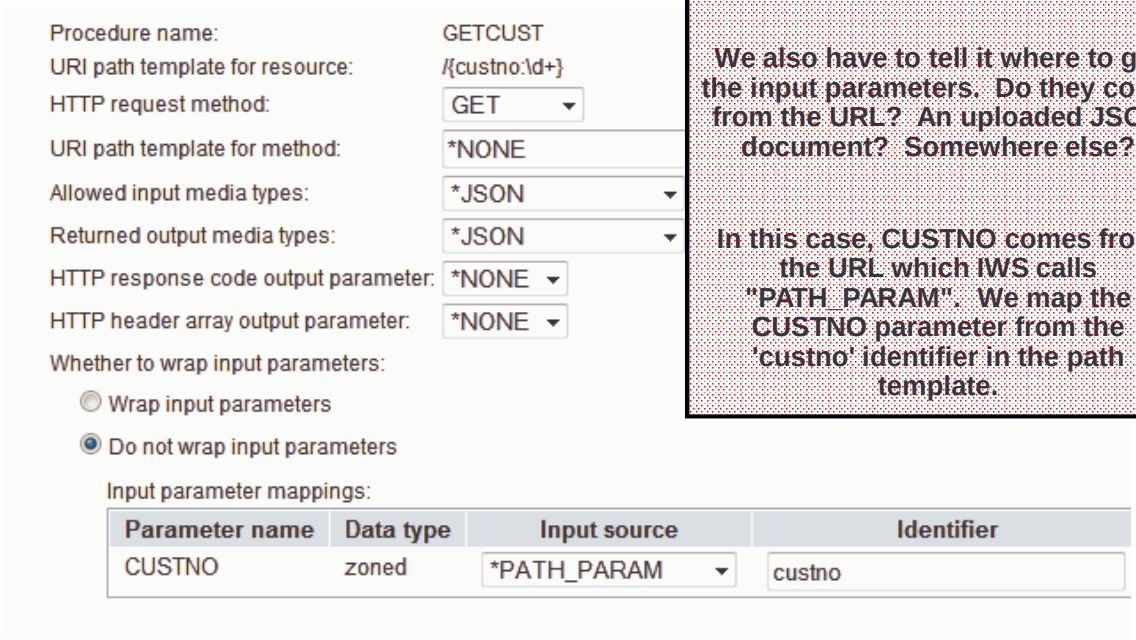
\w+ = one or more of A-Z, a-z or 0-9 characters.
Aisle is only one letter, but can be A-Z (capital)
slot is always a two-digit number, from 00-99, \d\d means two numeric digits
Shelf is always capital letters A,B,C,D or E.

IWS uses Java regular expression syntax. A tutorial can be found here:
https://docs.oracle.com/javase/tutorial/essential/regex/

REST Wizard (4 of 9)



REST Wizard (5 of 9)



Here we tell it we want to use GET, and JSON as the data format.

We also have to tell it where to get the input parameters. Do they come from the URL? An uploaded JSON document? Somewhere else?

In this case, CUSTNO comes from the URL which IWS calls "PATH_PARAM". We map the CUSTNO parameter from the 'custno' identifier in the path template.

REST Wizard (steps 6 to 9)

These steps are the same as the SOAP version

STEP 6 = UserID to run the program under

STEP 7 = Library List to run under

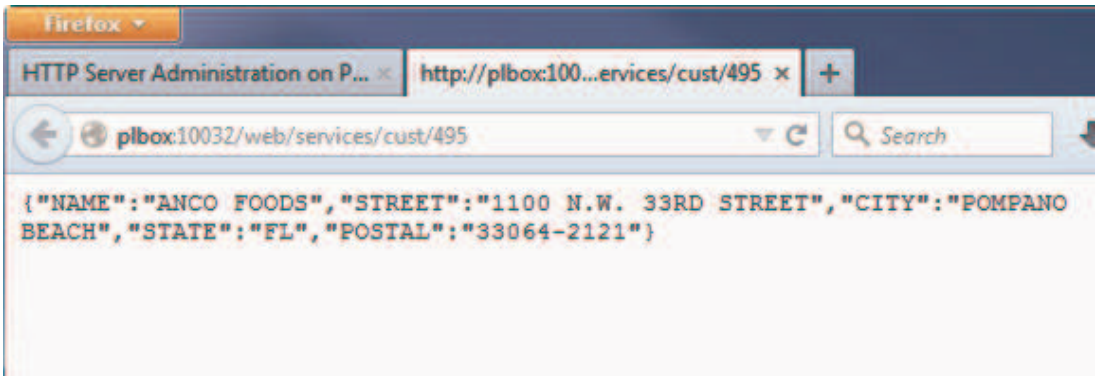
STEP 8 = consumer's IP address or any other HTTP meta data

STEP 9 = Summary screen where you click "Finish" to create the service.

Test REST By Doing a REST Test

When you put a URL into the "location" box in your web browser, the browser does a GET HTTP request. Therefore, a web browser is an easy way to test REST web services that use the GET method.

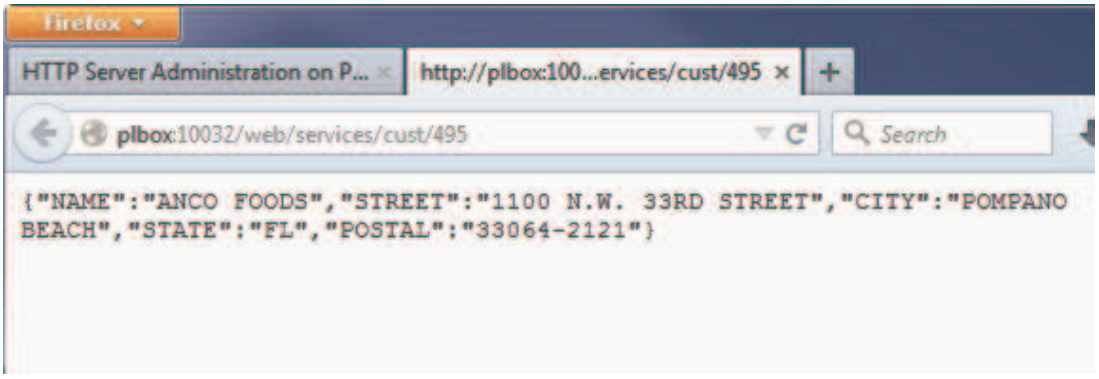
That way, you can make sure your service works before opening it up to other people who may be using a web service consumer.



Test REST By Doing a REST Test

When you put a URL into the "location" box in your web browser, the browser does a GET HTTP request. Therefore, a web browser is an easy way to test REST web services that use the GET method.

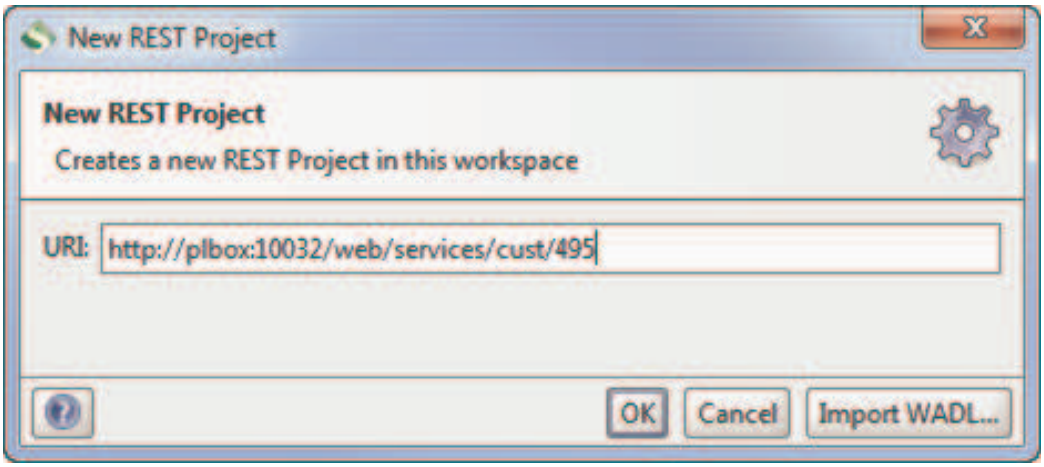
That way, you can make sure your service works before opening it up to other people who may be using a web service consumer.



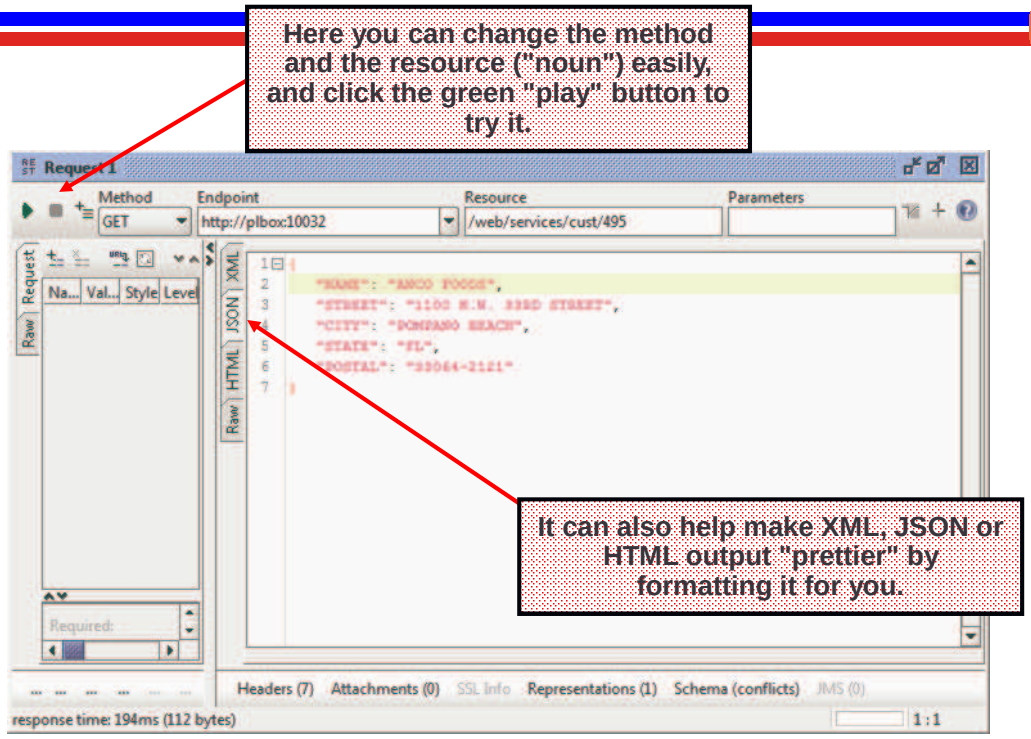
SOAPUI REST Testing (1 of 2)

Since it's hard to test other methods (besides GET) in a browser, it's good to have other alternatives. Recent versions of SoapUI have nice tools for testing REST services as well.

Choose File / New REST Project, and type the URL, then click OK



SOAPUI REST Testing (2 of 2)



Do It Yourself

IWS is a neat tool, but:

- Maximum of 7 params
- Can't nest arrays inside arrays
- Supports only XML or JSON
- Very limited options for security
- doesn't always perform well

Writing your own:

- Gives you complete control
- Performs as fast as your RPG code can go.
- Requires more knowledge/work of web service technologies such as XML and JSON
- You can accept/return data in any format you like. (CSV? PDF? Excel? No problem.)
- Write your own security. UserId/Password? Crypto? do whatever you want.
- The only limitation is your imagination.

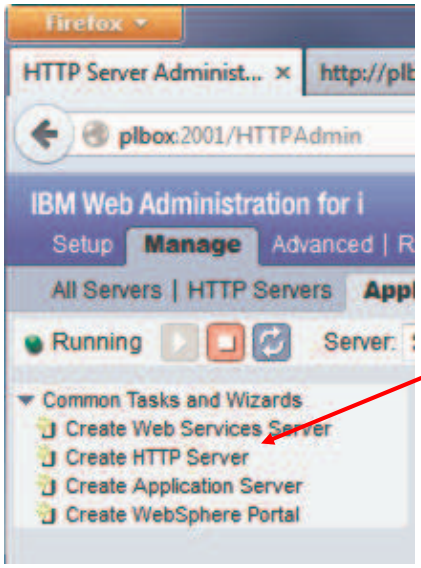
How To Write Your Own

When I write my own web services without IBM's tools, I typically:

- Use the standard IBM HTTP server (powered by Apache)
- Configure Apache to call my RPG program
- Use the "Read Standard Input" (QtmhRdStin) API to get any uploaded documents (such as XML or JSON input parameters)
- Use the "Write Standard Output" (QtmhWrStout) API to send back results.
- Retrieve the URL (to get REST resource) from the REQUEST_URI environment variable.
- If needed, the open source YAJL tool can generate/parse JSON
- RPG has built-in support for parsing XML
- If needed, the CGIDEV2 tool can make it easy to output either JSON or XML.

Set Up an Apache Server

In the same HTTP Administration Server (*ADMIN server) we've been using to create web service servers, there's an option to create a standard HTTP server.



Click the "Create HTTP Server" link.

This creates a standard IBM HTTP Server (powered by Apache)

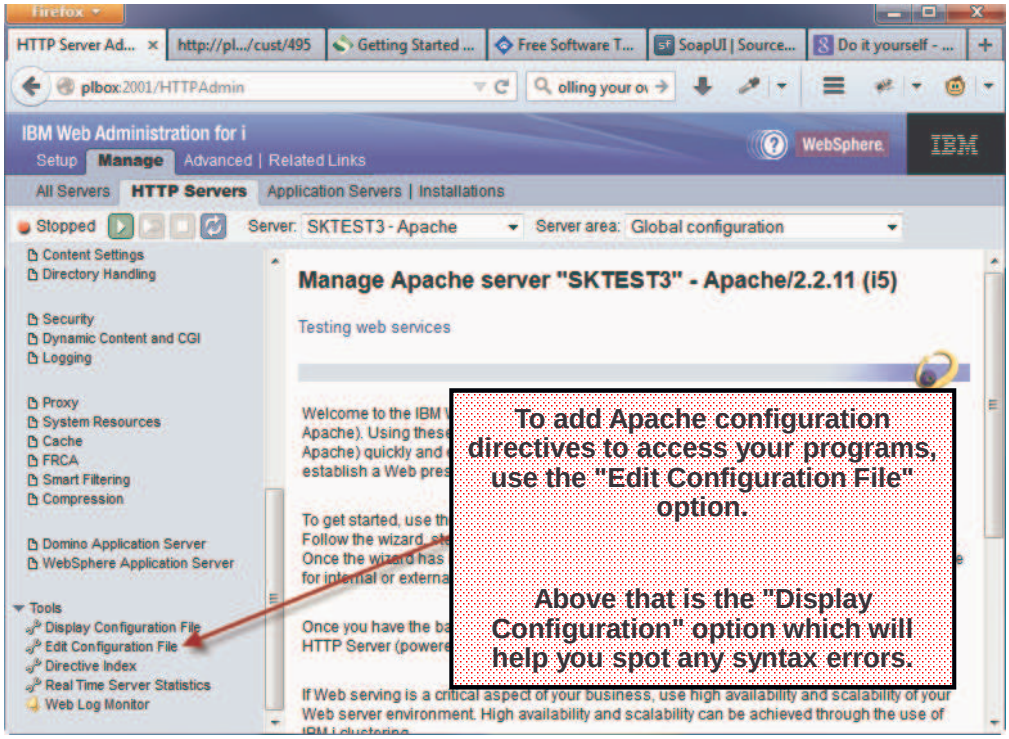
For brevity, I will refer to this as "Apache" going forward.

Set Up an Apache Server

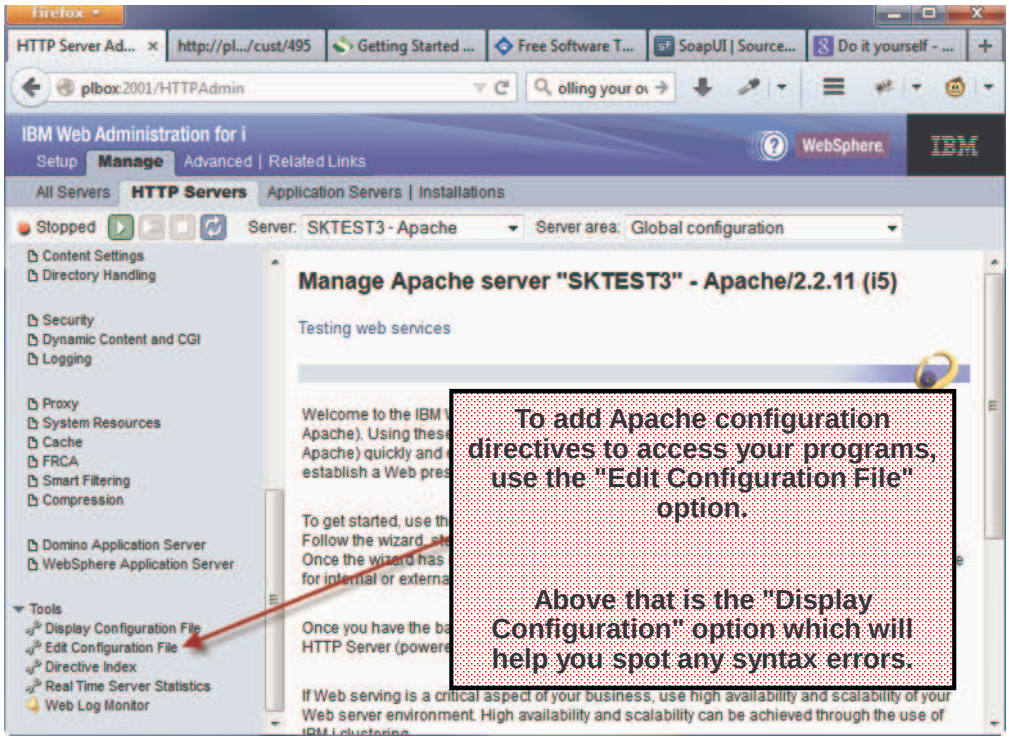
Setting up the server is done with a wizard, similar to what we've done for web servers. The prompts will be:

- Server Name = used to generate disk objects. Must be a valid IBM i object name.
- Server description = can be whatever you like
- Server root = IFS directory for server config files (just take the default.)
- Document Root = IFS directory for downloadable data (just take the default)
- Port number = pick one that's not used for another application. The 8000-10000 range is often used for HTTP servers. I will use 8500 for examples.
- Access log = Will keep a log of every access made to the server. Turn this on in test environments, off in production (unless you need to audit it.)
- Time to keep the logs = Allows Apache to purge old log files. I usually take the default of purge after 7 days.
- Summary page = shows the options you selected. Click "Finish" to create your server.

Tell Apache About Your RPG Programs



Tell Apache About Your RPG Programs



Apache directives (1 of 2)

Add one of these to the bottom of the config file:

```
ScriptAlias /cust /qsys.lib/skwebsrv.lib/custinfo.pgm
-or-
ScriptAliasMatch /rest/([a-z0-9]+)/.* /qsys.lib/skwebsrv.lib/$1.pgm
```

- A "Script Alias" tells Apache to call a disk object (instead of downloading it.)
- The first parameter is what Apache looks for in the URL.
- The second parameter is the IFS path name to your object.
- In the first example, a URL starting with /cust will do a CALL to program CUSTINFO in library SKWEBSRV.
- If using ScriptAliasMatch you can use regular expressions to allow generic names.
- In the second example, a URL that starts with /rest/ followed by one or more of characters a-z or digits 0-9, followed by another slash will be mapped to a program in the SKWEBSRV library. This way, I can make all programs in a library available with one directive.

Apache directives (2 of 2)

Beneath the ScriptAlias, enable access to the library:

```
<Directory /qsys.lib/skwebsrv.lib>
  Order Allow,Deny
  Allow From All
</Directory>
```

- Order Allow,Deny means to evaluate the Allow directives first, if no match, deny access.
- The Allow directive allows from all.
- Now people will be able to access the SKWEBSRV library.
- It's possible to restrict by IP address, too
- Or to require a userid/password, etc.

Once you've made your changes, use the 'Apply' button to save them, then click the "Play" button at the top to start your server.

DIY Customer Example

For this example, we will use the "ScriptAlias /cust" option given above.

We will use it to handle a URL like this one:

```
http://example.com:8500/cust/495
```

Like previous examples, our goal will be to return customer information. This time, though, we will write our own custom XML format like this, just to provide a simple example:

```
<result>
   <cust id="495">
      <name>ANCO FOODS</name>
      <street>1100 N.W. 33RD STREET</street>
      <city>POMPANO BEACH</city>
      <state>FL</state>
      <postal>33064-2121</postal>
   </cust>
</result>
```


This is CGI -- But It's Not HTML

Web servers (HTTP servers) have a standard way of calling a program on the local system. It's know as Common Gateway Interface (CGI)

- The URL you were called from is available via the REQUEST_URI env. var
- If any data is uploaded to your program (not usually done with REST) you can retrieve it from "standard input".
- To write data back from your program to Apache (and ultimately the web service consumer) you write your data to "standard output"

To accomplish this, I'm going to use 3 different APIs (all provided by IBM)

- **QtmhRdStin** ← reads standard input
- **getenv** ← retrieves an environment variable.
- **QtmhWrStout** ← writes data to standard output.

Example REST Provider (1 of 3)

```
     FCUSTFILE  IF   E           K DISK

     D getenv          PR                *   extproc('getenv')
     D   var                             *   value options(*string)

     D QtmhWrStout     PR                    extproc('QtmhWrStout')
     D   DtaVar                      65535a  options(*varsize)
     D   DtaVarLen                     10I 0 const
     D   ErrorCode                   8000A   options(*varsize)

     D err             ds                    qualified
     D   bytesProv                     10i 0 inz(0)
     D   bytesAvail                    10i 0 inz(0)

     D xml             pr              5000a   varying
     D   inp                           5000a   varying const

     D CRLF            C                     x'0d25'
     D pos             s                10i 0
     D uri             s               5000a   varying
     D data            s               5000a
```


Example REST Provider (2 of 3)

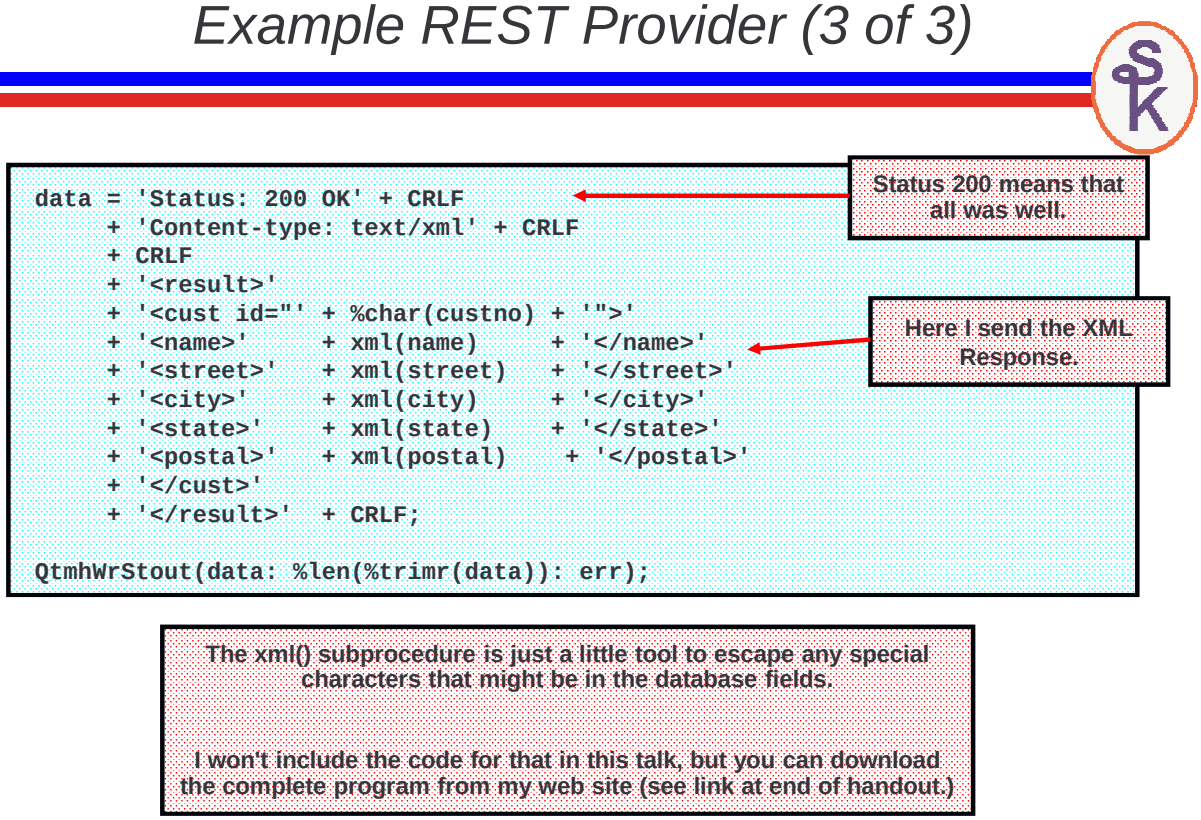
```
/free
  uri = %str( getenv('REQUEST_URI') );

  monitor;
     pos = %scan('/cust/': uri) + %len('/cust/');
     custno = %int(%subst(uri:pos));
  on-error;
     data = 'Status: 500 Invalid URI' + CRLF
          + 'Content-type: text/xml' + CRLF
          + CRLF
          + '<error>Invalid URI</error>' + CRLF;
     QtmhWrStout(data: %len(%trimr(data)): err);
     return;
  endmon;

  chain custno CUSTFILE;
  if not %found;
     data = 'Status: 500 Unknown Customer' + CRLF
          + 'Content-type: text/xml' + CRLF
          + CRLF
          + '<error>Unknown Customer Number</error>' + CRLF;
     QtmhWrStout(data: %len(%trimr(data)): err);
     return;
  endif;
```

REQUEST_URI will contain http://x.com/cust/495

Custno is everything after /cust/ in the URL

If an error occurs, I set the status to 500, so the consumer knows there was an error. We also provide a message in XML, in case the consumer wants to show the user.

Example REST Provider (3 of 3)



```
data = 'Status: 200 OK' + CRLF
     + 'Content-type: text/xml' + CRLF
     + CRLF
     + '<result>'
     + '<cust id="' + %char(custno) + '">'
     + '<name>'     + xml(name)     + '</name>'
     + '<street>'   + xml(street)   + '</street>'
     + '<city>'     + xml(city)     + '</city>'
     + '<state>'    + xml(state)    + '</state>'
     + '<postal>'   + xml(postal)    + '</postal>'
     + '</cust>'
     + '</result>'  + CRLF;

QtmhWrStout(data: %len(%trimr(data)): err);
```


The xml() subprocedure is just a little tool to escape any special characters that might be in the database fields.

I won't include the code for that in this talk, but you can download the complete program from my web site (see link at end of handout.)

Test It Like Any other REST Service

Final Thoughts on Providing

In this section we discussed providing using either the IBM tool or by rolling your own.

- The examples were simple, meant to give you the idea without spending a lot of time studying complex code.
- But more complex cases are very possible!
- You can use any RPG code you like in your programs, so the sky is the limit.
- In the IWS examples, you can use parameters that are arrays or data structures to handle more complex circumstances.
- In the DIY examples, you can receive/return gigabytes of data if needed, and use any file format that suits you.

RPG as a Web Service Consumer

Presented by

Scott Klement

http://www.scottklement.com

"I would love to change the world, but they won't give me the source code"

Approaches

There are many approaches available to consume web services from RPG:

- IBM's IWS has a client-side tool you can learn about here:
  http://www.ibm.com/systems/power/software/i/iws/
  - generates complex C code in IFS to be called from RPG
  - works well, but is limited in the formats it supports.
- In a recent technology refresh, HTTP support was added to SQL.
  - But, I find the syntax of retrieving HTTP and parsing XML in SQL to be extremely cumbersome. I prefer to save SQL for database access.
- Several commercial alternatives.
- Open Source HTTPAPI, created by Scott Klement (used in this talk)

HTTPAPI

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/`

Consume REST (1 of 3)

This is the DIY REST example from the last section -- but now I'll consume it!

```
     H DFTACTGRP(*NO) ACTGRP('KLEMENT') BNDDIR('HTTPAPI')

      /copy HTTPAPI_H
      /copy IFSIO_H

     D url             s           1000a   varying
     D stmf            s           1000a   varying
     D rc              s             10i 0
     D errMsg          s             52a   varying

     D custInfo        ds                    qualified
     D   id                           4s 0
     D   name                        25a
     D   street                      25a
     D   city                        15a
     D   state                        2a
     D   postal                      10a

     C     *ENTRY        PLIST
     C                   PARM                    InputCust       15 5
```


Consume REST (2 of 3)

```
/free
       stmf = '/tmp/getcust.xml';
       url = 'http://example.com:8500/cust/'
           + %char(%int(InputCust));

       rc = http_get(url: stmf);
       if (rc<>1 and rc<>500);
          http_crash();
       endif;

       if rc=500;
          xml-into errMsg %xml(stmf: 'path=error doc=file');
          dsply errMsg;
       else;
          xml-into custInfo %xml(stmf: 'path=result/cust doc=file');
          dsply custInfo.name;
          dsply custInfo.street;
          dsply ( custInfo.city + ' '
                + custInfo.state + ' '
                + custInfo.postal );
       endif;

       unlink(stmf);
       *inlr = *on;
     /end-free
```

Consume REST (3 of 3)

When I run it like this:

```
CALL MYCUST PARM(495)
```

It responds with:

```
DSPLY  ANCO FOODS
DSPLY  1100 N.W. 33RD STREET
DSPLY  POMPANO BEACH   FL 33064-2121
```

When I run it like this:

```
CALL MYCUST PARM(123)
```

It responds with:

```
DSPLY  Unknown Customer Number
```


Talking To Outside Web Services

Although you can consume your own RPG web services from RPG consumer programs (as I did in the last example) it doesn't offer many benefits vs. just calling those routines directly.

However, there is a lot of benefit available for using RPG consumers to integrate with other business partners web services!

For example:

- Get shipping information or track packages with UPS, Fedex, DHL, US Postal Service
- Process credit cards by communicating with a bank or other credit card provider
- Integrate your programs with 3rd party canned software that offers web services\
- Interact with cloud servers or SAS packages

The ability to call/interact with programs all over the world opens up huge new capabilities to your RPG programs.

Currency Exchange Example

In the first section of this seminar, we talked about currency exchange, and I showed you what the SOAP messages for WebServiceX.net's currency exchange looked like.

Now it's time to try calling that web service from an RPG program!

Steps to writing a SOAP web service consumer with HTTPAPI:

- Get the WSDL
- Try the WSDL with SoapUI so you know what it looks like.
- Copy/paste the XML for the SOAP message into an RPG program.
  - Convert to a big EVAL statement
  - Insert any variable data at the right places
  - Create one big string variable with XML data.
- Pass the SOAP message to HTTPAPI's **`http_post_xml()`** routine.
- Parse the XML you receive as a response.

SOAP Consumer (1/4)

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

D EXCHRATE        PR                  ExtPgm('EXCHRATE')
D   Country1                    3A   const
D   Country2                    3A   const
D   Amount                     15P 5 const
D EXCHRATE        PI
D   Country1                    3A   const
D   Country2                    3A   const
D   Amount                     15P 5 const

 /copy 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."

SOAP Consumer (2/4)

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

```
/free
  SOAP =
   '<?xml version="1.0" encoding="iso-8859-1" standalone="no"?>'
  +'<SOAP:Envelope'
  +'    xmlns:SOAP="http://schemas.xmlsoap.org/soap/envelope/"'
  +'    xmlns:tns="http://www.webserviceX.NET/">'
  +'<SOAP:Body>'
  +'  <tns:ConversionRate>'
  +'      <tns:FromCurrency>'+ %trim(Country1) +'</tns:FromCurrency>'
  +'      <tns:ToCurrency>'+ %trim(Country2) + '</tns:ToCurrency>'
  +'  </tns:ConversionRate>'
  +'</SOAP:Body>'
  +'</SOAP:Envelope>';

  rc = http_post_xml(
            'http://www.webservicex.net/CurrencyConvertor.asmx'
                  : %addr(SOAP) + 2
                  : %len(SOAP)
                  : *NULL
                  : %paddr(Incoming)
                  : %addr(rate)
                  : HTTP_TIMEOUT
                  : HTTP_USERAGENT
                  : 'text/xml'
                  : 'http://www.webserviceX.NET/ConversionRate');
```

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.

SOAP 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 = 'ConversionRateResult');
        rate = %float(value);
     endif;
 /end-free
P                 E
```

Display the error or result on the screen.

This is called for every XML element in the response.

When the element is a "Conversion Rate Result" element, save the value, since it's the exchange rate we're looking for!

SOAP 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('USD' 'EUR' 185.50)
      DSPLY  Result = 133.69









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_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 **`<ConversionRateResult>`** element, it saves the element's value to the "rate" variable.
- When **`http_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_post()`** API instead of **`http_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://www.webservicex.net/CurrencyConvertor.asmx'
                  : %addr(SOAP) + 2
                  : %len(SOAP)
                  : '/tmp/CurrencyExchangeResult.soap'
                  : HTTP_TIMEOUT
                  : HTTP_USERAGENT
                  : 'text/xml'
                  : 'http://www.webserviceX.NET/ConversionRate' );
. . .
```

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. (XML-SAX op-code)

Handling Errors with HTTPAPI

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 wi
      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)

WSDL2RPG

Instead of SoapUI, you might consider using WSDL2RPG – another open source project, this one from Thomas Raddatz. You give WSDL2RPG the URL or IFS path of a WSDL file, and it generates the RPG code to call HTTPAPI.

```
WSDL2RPG URL('/home/klemscot/CurrencyConvertor.wsdl')
         SRCFILE(LIBSCK/QRPGLESRC)
         SRCMBR(CURRCONV)
```

Then compile CURRCONV as a module, and call it with the appropriate parameters.

- Code is still beta, needs more work.
- The RPG it generates often needs to be tweaked before it'll compile.
- The code it generates is much more complex than what you'd use if you generated it yourself, or used SoapUI
- Can only do SOAP (not POX or REST)

But don't be afraid to help with the project! It'll be really nice when it's perfected!
http://www.tools400.de/English/Freeware/WSDL2RPG/wsdl2rpg.html

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, 57xx-SS1)
- TCP/IP Connectivity Utilities for iSeries (57xx-TC1)
- IBM HTTP Server for iSeries (57xx-DG1)
- IBM Developer Kit for Java (57xx-JV1)
- IBM Cryptographic Access Provider (5722-AC3) *(pre-V5R4 only)*

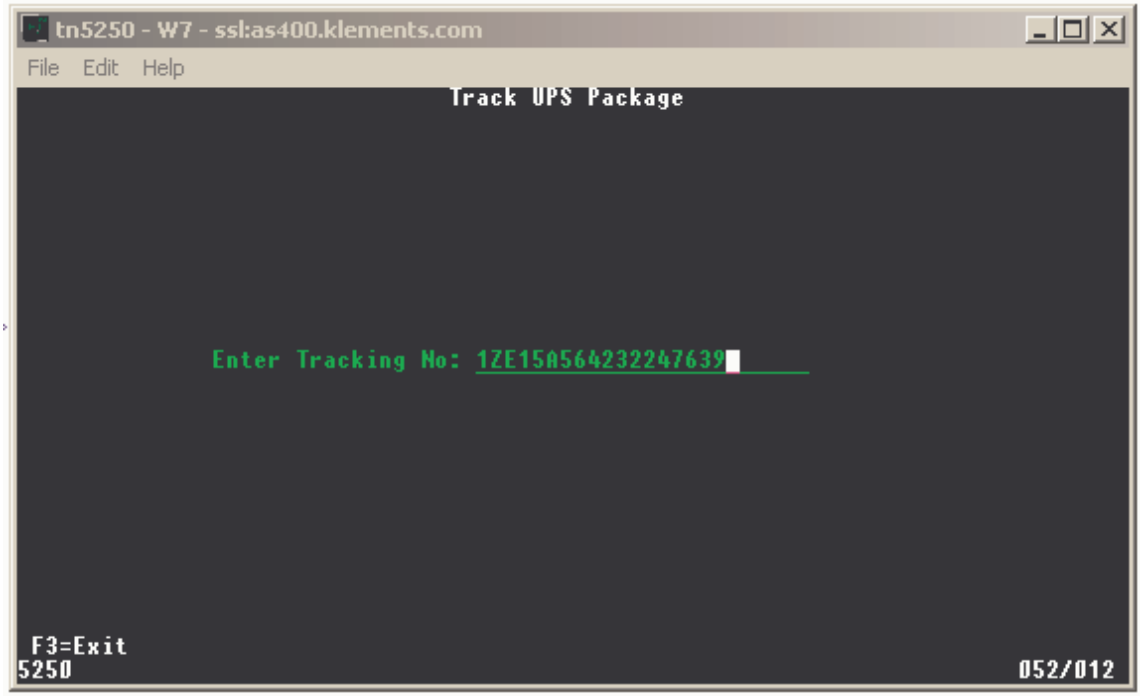
Because of (historical) import/export laws, 5722-AC3 is not shipped with OS/400. However, it's a no-charge item. You just have to order it separately from your business partner. It is included automatically in V5R4 and later as 57xx-NAE

UPS Example (slide 1 of 11)

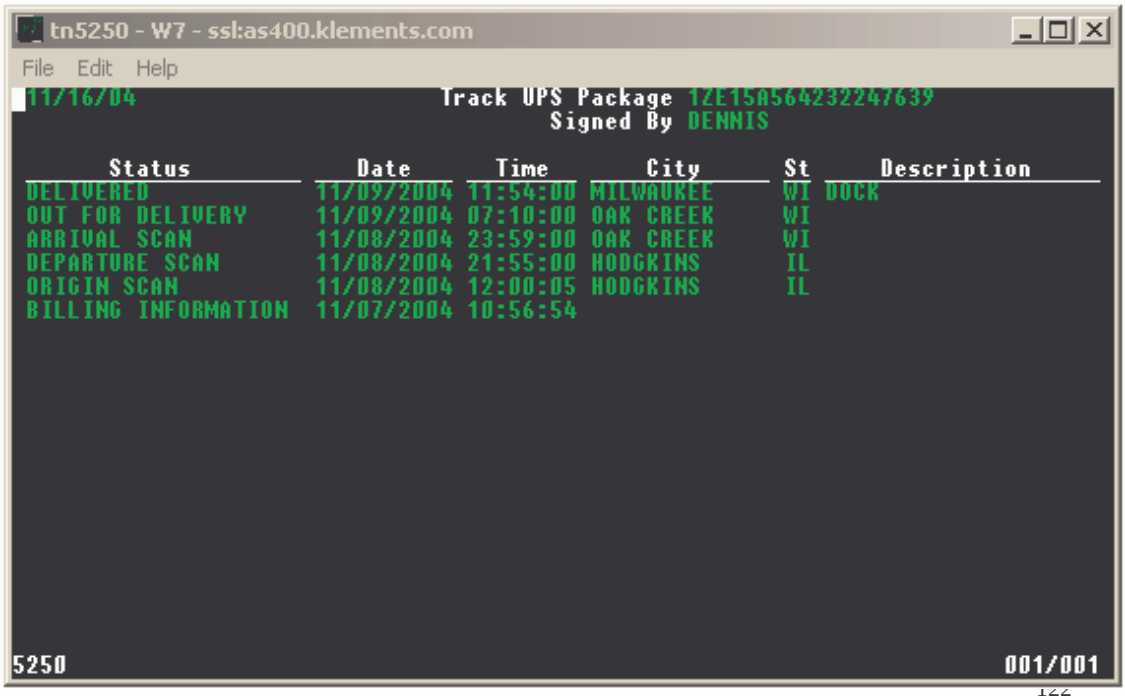
This demonstrates the "UPS Tracking Tool" that's part of UPS OnLine Tools. There are a few differences between this and the previous example:

- **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 WSDL for their Web services – but does use XML. Some folks call this "Plain Old XML" (POX).**
- **Instead of WSDL, 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.**

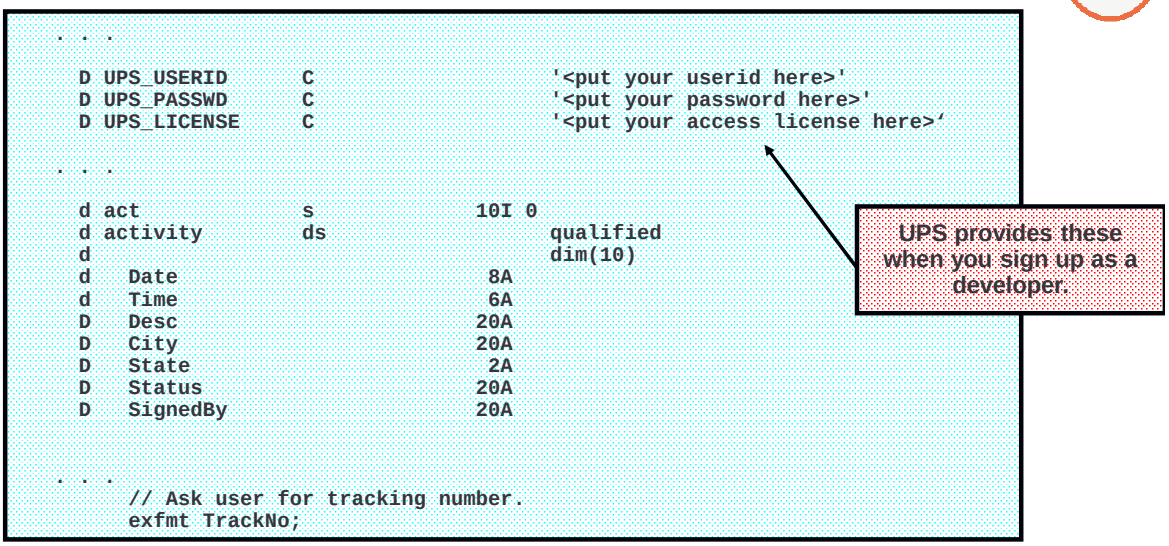
UPS Example (slide 2 of 11)



UPS Example (slide 3 of 11)



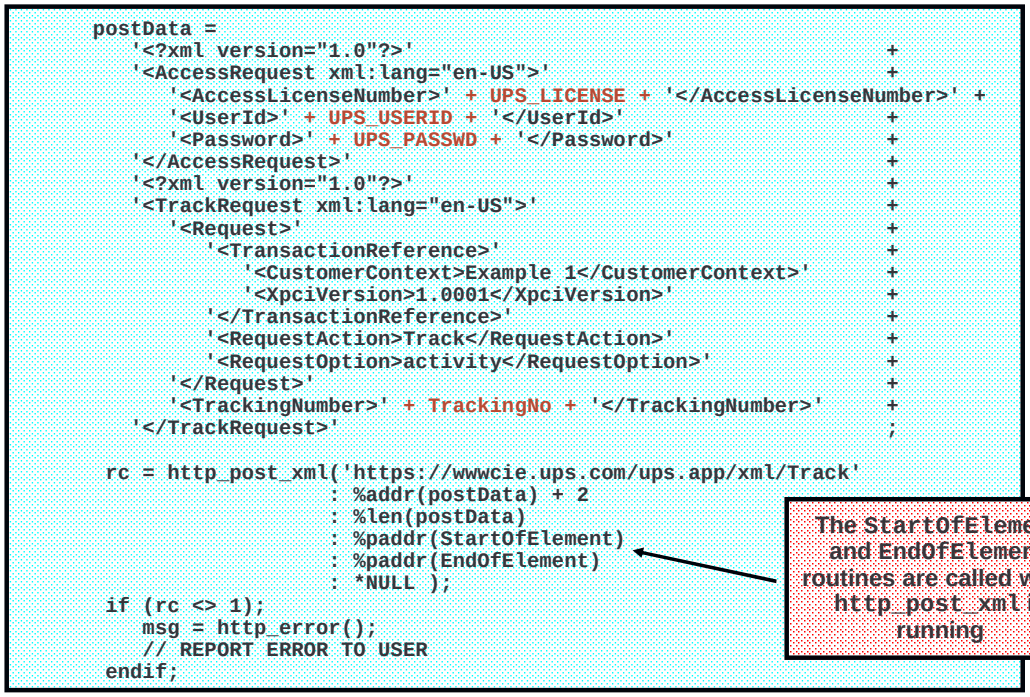
UPS Example (slide 4 of 11)



```
. . .

  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.

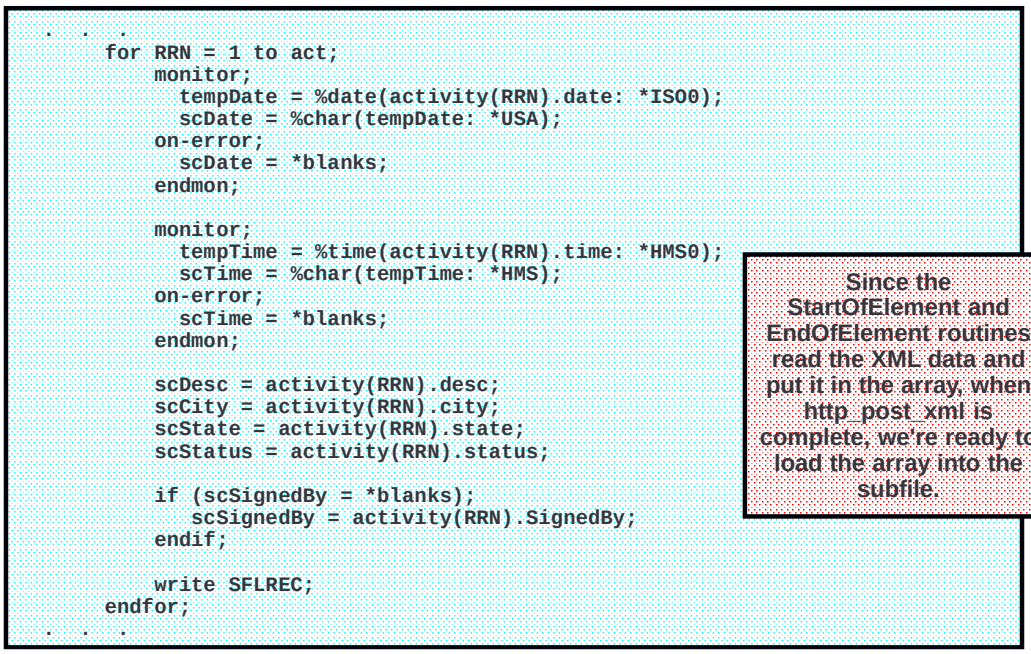
UPS Example (slide 5 of 11)



```
   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_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_post_xml is running

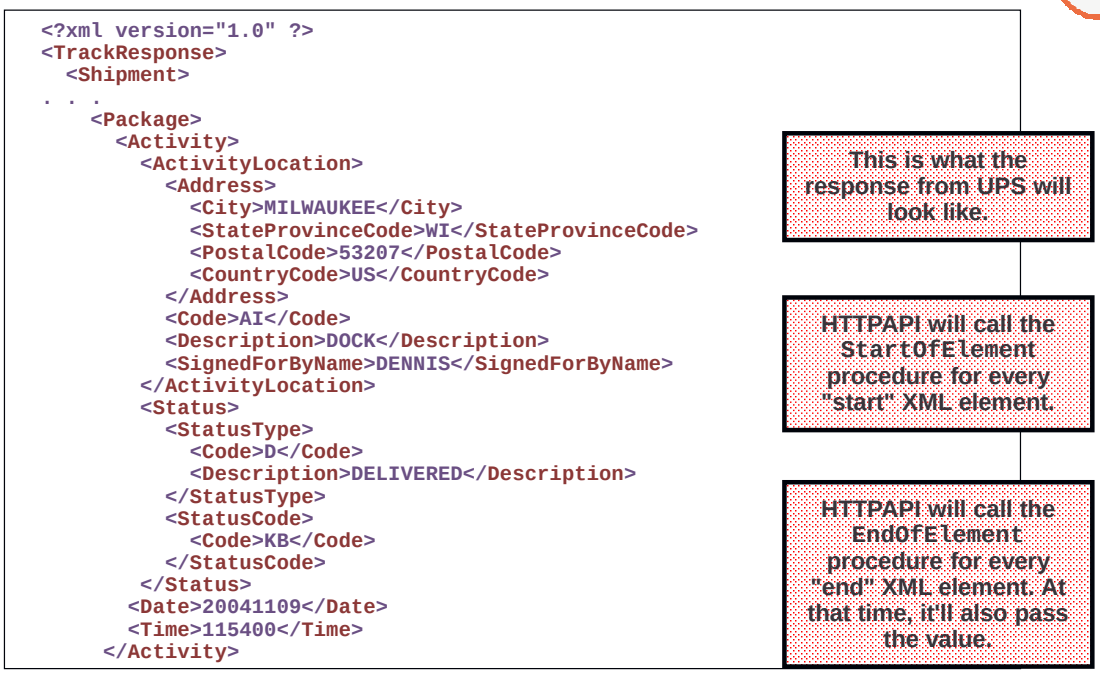
UPS Example (slide 6 of 11)



```
. . .
    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_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_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_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
```

HTTPAPI 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 - EXAMPLE20

The best places to get help for *HTTPAPI* are:

- the FTPAPI/HTTPAPI mailing list
  Signup: http://www.scottklement.com/mailman/listinfo/ftpapi
  Archives: http://www.scottklement.com/archives/ftpapi/
- Code/400 forums
  http://www.code400.com
- the iPro Developer Forums
  http://www.iprodeveloper.com/forums

More Information / Resources

Gaining a basic understanding of HTTP:

What Is HTTP, Really? (Scott Klement)
http://systeminetwork.com/article/what-http-really

What's the Difference Between a URI, URL, and Domain Name? (Scott Klement)
http://www.systeminetwork.com/article/application-development/whats-the-difference-between-a-uri-url-and-domain-name-65224

Gaining a basic understanding of Web Services & Terminology:

Web Services: The Next Big Thing (Scott N. Gerard)
http://www.systeminetwork.com/article/other-languages/web-services-the-next-big-thing-13626

SOAP, WDSL, HTTP, XSD? What? (Aaron Bartell)
http://systeminetwork.com/article/soap-wdsl-http-xsd-what

More Information / Resources

IBM's web site for the Integrated Web Services (IWS) tool:
http://www.ibm.com/systems/i/software/iws/

List of updates made to IWS, and which PTF level you need for each:
http://www.ibm.com/developerworks/ibmi/techupdates/iws

SoapUI home page
http://www.soapui.org

WSDL2RPG Home Page
http://www.tools400.de/English/Freeware/WSDL2RPG/wsdl2rpg.html

Call a Web Service with WSDL2RPG (Thomas Raddatz)
http://iprodeveloper.com/rpg-programming/call-web-service-wdsl2rpg

More Information / Resources

How-To Articles About Consuming/Providing Web Services:

RPG Consumes the REST (Scott Klement)
http://systeminetwork.com/article/rpg-consumes-rest

RPG Consuming Web Services with HTTPAPI and SoapUI (Scott Klement)
http://systeminetwork.com/article/rpg-consuming-web-services-httpapi-and-soapui

IBM's Integrated Web Services (Scott Klement)
http://systeminetwork.com/article/ibms-integrated-web-services

Consume Web Services with IBM's IWS (Scott Klement)
http://www.systeminetwork.com/article/rpg-programming/consume-web-services-with-ibms-iws-66209

UPS OnLine Tools
http://www.ups.com/e_comm_access/gettools_index

More Information / Resources

Sites that offer web service directories

- **WebServiceX.net**
- **XMethods.net**
- **BindingPoint.com**
- **RemoteMethods.com**

RPG's XML Opcodes & BIFs:

"Real World" Example of XML-INTO (Scott Klement)
http://systeminetwork.com/article/real-world-example-xml

RPG's XML-SAX Opcode
http://systeminetwork.com/article/rpgs-xml-sax-opcode

PTFs for Version 6.1 Enhance RPG's XML-INTO
http://systeminetwork.com/article/ptfs-version-61-enhance-rpgs-xml

XML-INTO: Maximum Length
http://systeminetwork.com/article/xml-maximum-length

XML-INTO: Read XML Data Larger Than 65535
http://systeminetwork.com/article/xml-read-xml-data-larger-65535

XML-INTO: Output to Array Larger than 16 MB
http://systeminetwork.com/article/xml-output-array-larger-16-mb

This Presentation

You can download a PDF copy of this presentation from:

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

The Sample Web Service Providers/Consumers in this article are also available at the preceding link.

Thank you!