Consuming Web Services from RPG

with HTTPAPI

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."

Our Agenda

1. Introduction
   - What's a web service?
   - Consuming vs. Providing
   - Types (REST/SOAP/XML/JSON)
2. Consuming a Web Service w/Utility
3. What is HTTPAPI? What are alternatives?
4. Example- Simple Web Service
5. Example- REST Web Service
6. Example- SOAP web service

What is a Web Service?

An API call using internet-type communications

- ”API” refers to a program that has no user interface and is meant to be called by other programs
- Input comes from “parameters”
- Output is returned in “parameters”
- They provide a “service” for their caller
- Can be called on the local machine, LAN, WAN or Internet (at provider’s discretion)

What is…. Example scenarios

- Web server on Linux needs data from IBM i program to determine when a work order will be complete. Calls RPG program, gets result. Shows result to end-user.
- Green-screen application needs to process credit cards. Calls bank’s computer, passes card info, gets back confirmation number.
- Application needs exchange rate to convert US dollars to Euros. Calls program on bank’s computer to get it.
- Track packages with UPS, DHL, USPS, FedEx, etc.
- Integrate CRM application on Windows Server in San Diego, CA with Billing Application on IBM i in Milwaukee, WI
- Mobile app sold in Google Play or Apple App Store needs access to data on IBM I
- Application has text in English, but needs it in Spanish. Calls Web Service, passes English text, gets back Spanish.

Consuming vs. Providing

In Web Services, these terms are important:

- Provider = program that provides a service (the “server” side of communications). This is the API.
- Consumer = program that makes the call (the “client” side of communications.). This calls the API.

This session focuses on consuming (not providing) web services.

Identify Consumer vs. Provider

- Web server on Linux needs data from IBM i program to determine when a work order will be complete. Calls RPG program, gets result. Shows result to end-user.
- Green-screen application needs to process credit cards. Calls bank’s computer, passes card info, gets back confirmation number.
- Application needs exchange rate to convert US dollars to Euros. Calls program on bank’s computer to get it.
- RPG Program tracks packages with UPS, DHL, USPS, FedEx, etc.
- Integrate CRM application on Windows Server in San Diego, CA with Billing Application on IBM i in Milwaukee, WI
- Mobile app sold in Google Play or Apple App Store needs access to data on IBM I
- Application has text in English, but needs it in Spanish. Calls Web Service, passes English text, gets back Spanish.

Consumers are in Red

Providers are In blue

Internet-type Communications

- I really meant "HTTP".
- That's really the *only* "web" part about "web services"
- Is *not* the same as a web page (does not have a UI)
- *A web browser is not used.*
- Can be consumed by a web page, but doesn't have to be!
- Can be a green-screen application, mobile application, Windows application, etc.
- *Always platform/language agnostic*. Can be called from anywhere.

Translation Example

We want to translate text from English to Spanish.

Remember: We're making a program call using HTTP

Input parameters:

```
model_id = 'en-es'; // translate English(en) to Spanish(es)
text = 'Hello';     // text to translate
```

Output parameter:

Translated text: `'Hola'`

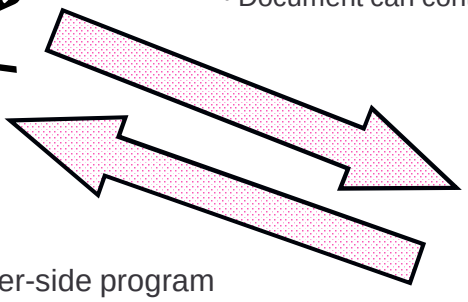
You can think of it like this:

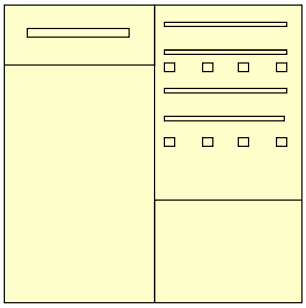
```
CALL PGM(TRANSLATE) PARM('en-es' 'Hello' &RESULT)
```

How Does It Really Work?

HTTP starts with a request for the server
- Can include a document (XML, JSON, etc)
- Document can contain "input parameters"

model_id=en-es
text=Hello





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.

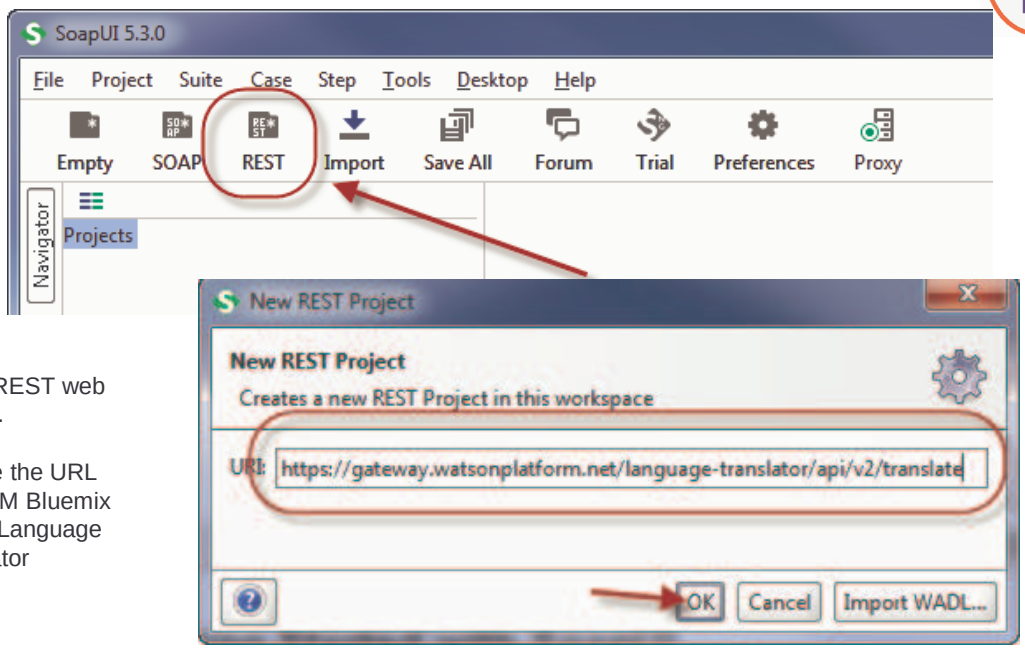
Result = hola

How Can We Try It Out?

- Web services are for program-to-program communication
- Normally, to use them, you must write a program!
- A web service testing tool allows testing without writing a program.
- Soap UI is a great (highly recommended) testing tool
  - Available in "Open Source" and "Professional" versions
  - Scott uses the open source (free) version.
  - Despite the name, can test REST as well as SOAP services

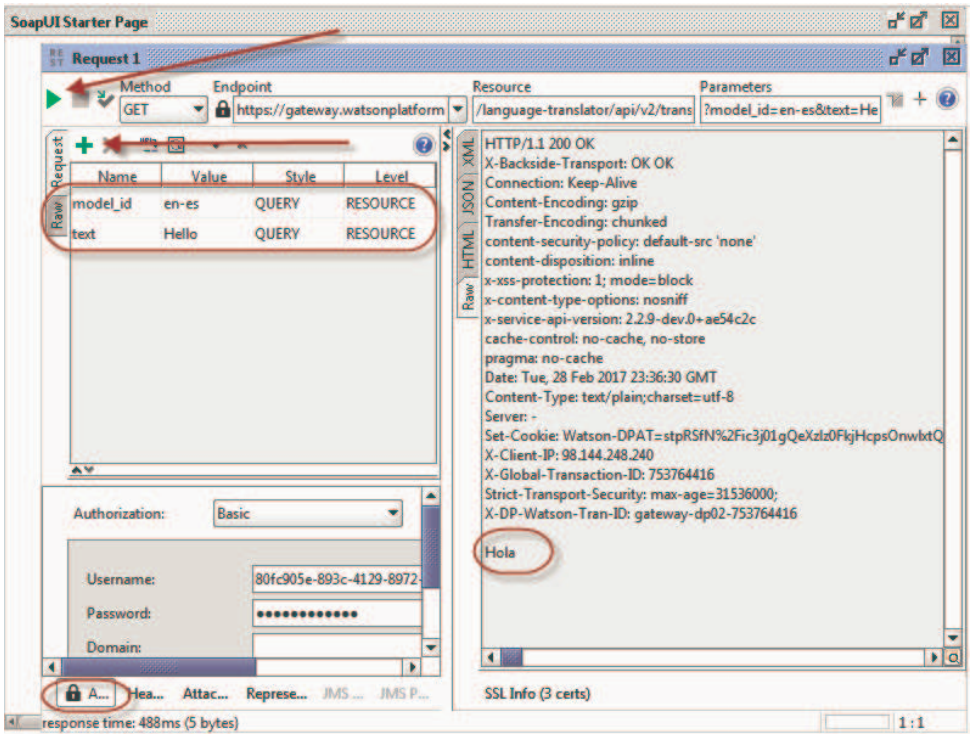
http://www.soapui.org

Setting It Up in SoapUI



- Use a REST web service.
- Provide the URL from IBM Bluemix for the Language Translator

Trying It Out in SoapUI



- Use the green PLUS to add the parameters
- Use the lock icon at the bottom to add the userid & password
- Use the green "play" button to run the web service
- The response is on the right.

What Just Happened?

- IBM Watson provides a language translation web service
- Soap UI is a testing tool that can consume web services
- We used the HTTP protocol
- Called IBM's "v2 translation" program
- Passed the languages to translate from/to.
- Passed the text to translate
- Got back the translated text

IBM Watson Language Translation

Not Really REST?

- Does not use the URL to identify a "resource"
- Does not use GET/PUT/POST/DELETE to determine what to do to the resource
- Purists would say it's not "REST", but a lot of people (most people?) now consider anything REST that is simple to use, like the Watson/Bluemix example.

Using It Yourself

- Fully supports commercial use
- Must sign up. (Free for 30 days.)
- First 250k of data translated for free
- After that, they charge per 1000 characters. Very inexpensive!

https://www.ibm.com/cloud-computing/bluemix/watson

HTTPAPI

Open Source (completely free tool)

- Created by Scott Klement, originally in 2001
- Written in native RPG code
- Enables HTTP communication from your ILE RPG programs
- http://www.scottklement.com/httpapi

2017 Updates

- Easier to use. Easier string support. Better HTTP method support.

Alternatives

- DB2 SQL HTTPGETCLOB, HTTPPOSTCLOB, etc functions
- IBM provides a SOAP (only) client in IWS
- 3rd party tools like GETURI

Language Translation in RPG

```
http_setAuth( HTTP_AUTH_BASIC
            : 'your-Watson-userid'
            : 'your-Watson-password' );

url = 'https://gateway.watsonplatform.net'
    + '/language-translator/api/v2/translate'
    + '?model_id=' + http_urlEncode(fromLang + '-' + toLang)
    + '&text=' + http_urlEncode(%trimr(fromText));

monitor;
   resp = http_string('GET': url);
on-error;
   httpcode = http_error();
endmon;
```

http_setAuth() – sets the userid/password used.

http_urlEncode() – encodes data so that it can safely be used in a URL

http_string() – sends an HTTP request, getting the input/output from strings

http_error() – returns an error message from HTTP communications

Running the Program

For example, the data from this screen can be fed into the code from the last slide.

The output of the last slide can be placed under "To Text".

```
                        Translate Text with IBM Watson

Languages: en to es    EN=English ES=Spanish FR=French IT=Italian PT=Portuguese

From Text:
Hello, my name is Scott
                                                                               
                                                                               
                                                                               
                                                                               
                                                                               


To Text:
Hola, mi nombre es Scott




HTTP Code:

F3=Exit
5250                                                                  024/006
```

Types of Web Services

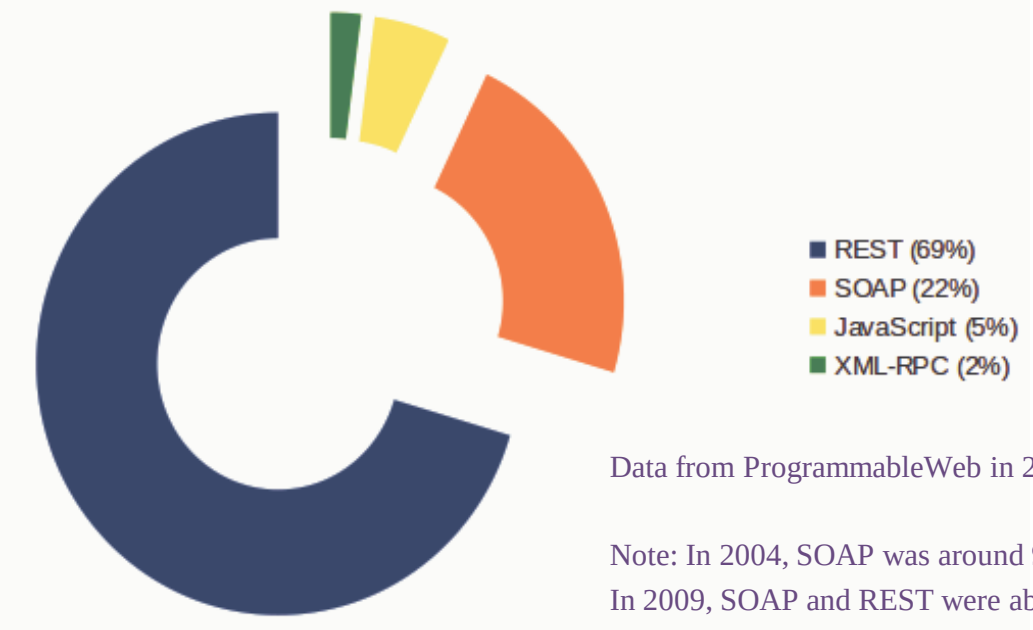
REST

- Most popular paradigm in the world (69% of all services and growing)
- URL represents a "resource"
- You can retrieve, create, modify or delete the resource
- Data can be in any format, but JSON is most popular, followed by XML
- *The term "REST" is often applied to any simple web service* (one that does not follow a complex standard like SOAP or XML-RPC)

SOAP

- Popularity peaked around 2004 (was 90%, now only 22% and shrinking)
- Highly standardized XML, requires more bytes, complexity
- Always uses the POST HTTP method
- Works well with tooling
- Too complicated to use without tooling

Types of Web Services



Data from ProgrammableWeb in 2014

Note: In 2004, SOAP was around 90%
In 2009, SOAP and REST were about even.

Data Formats (XML and JSON)

Both XML and JSON are widely used in web services:

- Self-describing.
- Can make changes without breaking compatibility
- Available for all popular languages / systems

XML:

- Has schemas, namespaces, transformations, etc.
- Has been around longer.
- Only format supported in SOAP

JSON:

- Natively supported by all web browsers
- Results in smaller documents (means faster network transfers)
- Parses faster.

JSON and XML to Represent a DS

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

Array of data structures in RPG...

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

Array of data structures in JSON

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

Array of data structures in XML

Without Adding Spacing for Humans

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

87 bytes

```
<list><cust><custno>1000</custno><name>ACME, Inc</name
></cust><cust><custno>2000</custno><name>Industrial S
upply Limited</name></cust></list>
```

142 bytes

In this simple "textbook" example, that's a 35% size reduction.

55 bytes doesn't matter, but sometimes these documents can be megabytes long – so a 35% reduction can be important.

…and programs process JSON faster, too!

Customer Maintenance Example

- The Watson example wasn't "real" REST, it was just a simple web service that played by it's own rules.
- For a "real" REST example, I have a Customer Maintenance web service on my IBM i. It is a demo web service that I wrote. You can download the full source code for both the provider and green-screen consumer from my web site.
- The purpose is to let sales people view, add and change customer information.
- Used a web service (instead of database directly) so we can have mobile, web and green-screen applications that all share the same back-end program.
- This web service happens to support both xml and json.
- The business logic is VERY simple, but it provides a good demonstration of REST web service mechanics.

What Is Meant By "Real" REST?

A REST "purist" would tell you that REST is where the URL specifies a "resource" (something to operate on) an the HTTP method specifies what to do.

```
http://my-server/webservices/cust/1234
```

- GET = Retrieve the resource (get customer 1234)
- PUT = Make idempotent changes (update customer 1234)
- POST = Make non-idempotent changes (write customer 1234)
- DELETE = Removes the resource (delete customer 1234)

Idempotent means that multiple calls will result in the same thing. For example, if you set a customers "balance" to 10, it does not matter if you do it once or 100 times, the end result will be a balance of 10. However, if you add 10 to their balance, it is not idempotent. If you do it once, it's 10 higher, do it 100 times, it's 1000 higher.

REST/CRUD analogy

An easy way to understand "real" REST is to think of it like Create, Retrieve, Update, Delete (CRUD) database operations.

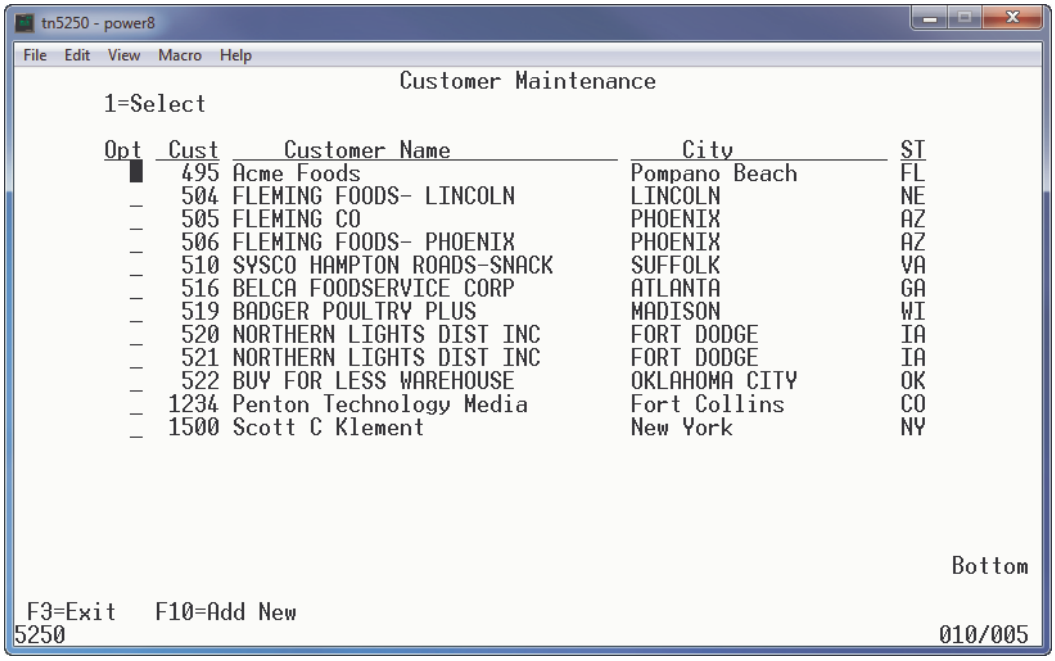
```
http://my-server/webservices/xml/cust/1234
```

- URL = an identifier, like a "unique key" (identity value) that identifies a record. (But also identifies what type of record, in this case, a customer.)
- GET = Retrieves – much like RPG's READ opcode reads a record
- PUT = Modifies – much like RPG's UPDATE opcode
- POST = Creates – much like RPG's WRITE opcode (or SQL INSERT)
- DELETE = Removes – like RPG's DELETE

Consider the difference between writing a record and updating it. If you update it 100 times, you still have the one record. If you write (insert) 100 times, you have 100 records. That is idempotent vs. non-idempotent.

Cust Maint – Start Screen

The customer maintenance program starts by letting the user select a customer.



Retrieving All Customers

That list did not come (directly) from a database – it came from consuming the web service!

This web service returns a list of all customers when you do a GET request and do not provide a customer number.

```
GET http://my-server/webservices/xml/cust
```

- HTTP GET is the REST for "retrieve"
- The "resource" in this is "customers in general" since no specific number was given.

```
dcl-c BASEURL 'http://localhost:8500/webservices/xml/cust';
dcl-s xmlData varchar(100000);

xmlData = http_string( 'GET' : BASEURL );
```

- http_string() routine receives data into a string (vs. a file).
- The first parameter is the HTTP method (GET)
- xmlData will be the XML document (all customers) as a string.

Reading the Returned XML

The XML returned from the service looks like this. RPG's built-in XML-INTO opcode can easily put this data into a data structure.

```
<cust success="true" errorMsg="">
  <data custno="495">
    <name>Acme Foods</name>
    <address>
      <street>1100 NW 33rd Street</street>
      <city>Pompano Beach</city>
      <state>FL</state>
      <postal>33064-2121</postal>
    </address>
  </data>
  <data custno="XXXXX">
  ... repeats for each customer ...
  </data>
</cust>
```

Data Structure Definition

The data structure needs to have the same definition as the XML document.

```
dcl-ds address_t qualified template;
   street varchar(30) inz('');
   city   varchar(20) inz('');
   state  char(2)     inz('  ');
   postal varchar(10) inz('');
end-ds;

dcl-ds data_t qualified template;
   custno packed(5: 0) inz(0);
   name varchar(30) inz('');
   address likeds(address_t) inz(*likeds);
end-ds;

dcl-ds cust_t qualified template;
   success varchar(5) inz('true');
   errorMsg varchar(500) inz('');
   data likeds(data_t) inz(*likeds) dim(999);
   cnt_data int(10);
end-ds;

dcl-ds cust likeds(cust_t) inz(*likeds);

xml-into cust %xml(xmlData:'case=any path=cust countprefix=cnt_');
```

"cust" contains "data" inside it (like the XML)

Likewise, "data" will contain "address"

cnt_data will be populated with the number of data elements in the XML using RPG's "countprefix" option.

Loading the List Into the Subfile

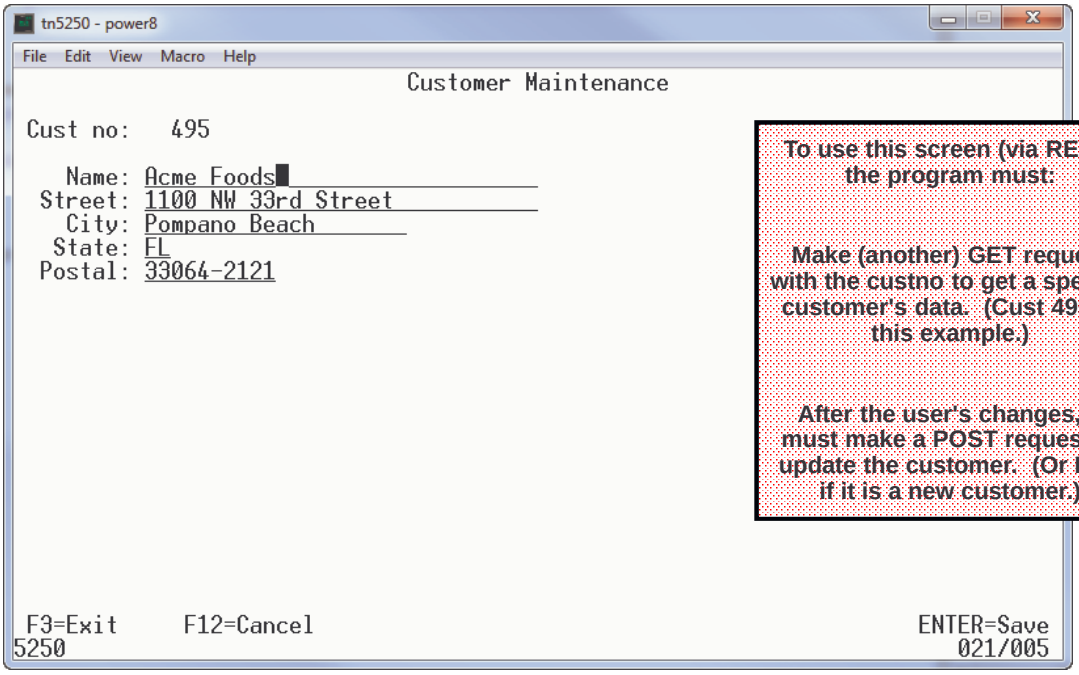
```
dcl-ds cust likeds(cust_t) inz(*likeds);

xml-into cust %xml(xmlData:'case=any path=cust countprefix=cnt_');

clearSfl();

for i = 1 to cust.cnt_data;

   custno = cust.data(i).custno;
   name   = cust.data(i).name;
   street = cust.data(i).address.street;
   city   = cust.data(i).address.city;
   state  = cust.data(i).address.state;
   postal = cust.data(i).address.postal;
   opt    = *blanks;

   RRN += 1;
   recsLoaded = RRN;
   write SFL;
   dspf.sfldsp = *on;

endfor;
```

The XML-INTO simply puts the XML data into the matching data structure.

To load the subfile, I can just loop through the array of "data" elements and load it.

Maintenance Screen

When you select a customer, it displays this screen



To use this screen (via REST) the program must:

Make (another) GET request with the custno to get a specific customer's data. (Cust 495 in this example.)

After the user's changes, it must make a POST request to update the customer. (Or PUT if it is a new customer.)

Retrieve Specific Customer

To retrieve information about a customer (name, address, etc)

```
GET http://my-server/webservices/xml/cust/XXXXX
```

In RPG (with HTTPAPI) the code looks like this:

```
dcl-c BASEURL 'http://localhost:8500/webservices/xml/cust';
dcl-s xmlData varchar(5000);

xmlData = http_string( 'GET' : BASEURL + '/' + %char(custno));
```

This is exactly like the previous example, except:

- A slash and customer number are added to the URL.
- xmlData can be smaller because only one record will be returned.

Populating the Maintenance Screen

```
dcl-ds orig likeds(data_t) inz(*likeds);

xml-into cust %xml(xmlData:'case=any path=cust countprefix=cnt_');

// If there was an error, put it on the screen
if cust.success <> 'true';
   msg = cust.errorMsg;
endif;

// If no error, put the cust data on the screen.
if cust.success = 'true';
   custno = cust.data(1).custno;
   name   = cust.data(1).name;
   street = cust.data(1).address.street;
   city   = cust.data(1).address.city;
   state  = cust.data(1).address.state;
   postal = cust.data(1).address.postal;
   eval-corr orig = cust.data(1);
endif;
```

The exact same data structure is used for XML-INTO

The only difference is that there will be only 1 <data> element (only one customer)

Just copy element 1 to the screen fields...

Also save an "orig" copy of the data so we can tell what was changed.

NOT SHOWN: If the user hit F10 = new customer, we skip this and just blank out the screen fields and "orig"

We Send XML For Updates

When doing a POST/PUT to save the changes, we send an XML document.

The format the XML is the same, except:

- It is generated by the consumer (we have to create it)
- Only the fields that were changed are sent.
- In this example, the street address, city and state were changed:

```
1  <cust success="true" errorMsg="">
2    <data>
3      <address>
4        <street>123 Sesame Street</street>
5        <city>Houston</city>
6        <state>TX</state>
7      </address>
8    </data>
9  </cust>
```

Creating XML in RPG

```
dcl-s data varchar(5000);

data = '<?xml version="1.0"?>+
        <cust success="true"><data>';

if name <> orig.name;
   data += '<name>' + xml(name) + '</name>';
endif;

data += '<address>';

if street <> orig.address.street;
  data += '<street>' + xml(street) + '</street>';
endif;

... above repeated for each field ...

 data += '</address>';
 data += '</data></cust>';
```

RPG does not have an opcode to create XML.

...but, it is not hard to create XML with string concatenation!

The only tricky part is what about special characters in the data, like <, > or &?

For that, I wrote the xml() subprocedure (next slide)

When "New Customer" was selected, orig is blank.

Escaping Special Characters

```
dcl-proc xml;
  dcl-pi *n varchar(5000);
     inp varchar(5000) const options(*varsize);
  end-pi;
  dcl-s x int(10);
  dcl-s result varchar(5000);

  for x = 1 to %len(inp);
     select;
     when %subst(inp:x:1) = '&';
        result += '&amp;';
     when %subst(inp:x:1) = '<';
        result += '&lt;';
     when %subst(inp:x:1) = '>';
        result += '&gt;';
     when %subst(inp:x:1) = '"';
        result += '&quot;';
     when %subst(inp:x:1) = '''';
        result += '&apos;';
     other;
        result += %subst(inp:x:1);
     endsl;
  endfor;

  return %trim(result);
end-proc;
```

For example, input like "Gravity < Zero"

Will be escaped like "Gravity < Zero"

Sending Changes To Provider

```
if isNew;
   method = 'POST';
   url = BASEURL;
else;
   method = 'PUT';
   url = BASEURL + '/' + %char(custno);
endif;

monitor;
   http_string( method: url: data: 'text/xml' );
on-error;
   msg = http_error();
endmon;
```

F10=New Customer sets "isNew" indicator.

In that case, no customer number is given, since it will be generated.

Remember: PUT is for update, POST is for writing new customer.

http_string() has optional parameters when a document needs to be sent.

- 3rd parameter is the data to send
- 4th parameter identifies the type of the data sent.
- The server will return the updated customer record (this consumer doesn't use it, however.)

Working With JSON Data

The Customer Maintenance Web Service also supports JSON instead of XML. It works exactly the same, except:

- data is json instead of xml (of course)
- URL is `http://your-server/webservices/json/cust`
- Type is sent as `'application/json'`

```
{
    "success": true,
    "errorMsg": "",
    "data": [
        {
            "custno": 495,
            "name": "Acme Foods",
            "address": {
                "street": "1100 NW 33rd Street",
                "city": "Pompano Beach",
                "state": "FL",
                "postal": "33064-2121"
            }
        },
        { repeated for each customer }
    ]
}
```

RPG Does Not Have JSON Opcodes

While this service provides both XML and JSON, some REQUIRE JSON … and RPG doesn't have built-in support.

- Try the YAJL open-source project
- or… DB2 has a new JSON_TABLE function you can use via SQL
- I prefer YAJL, it runs much faster and is more versatile.

Unfortunately, I don't have time to teach both YAJL and HTTPAPI in one presentation!

But, here is a quick overview:

- yajl_buf_load_tree() loads JSON data from a string into a "json tree" in memory.
- yajl_object_find() can find one sub-field in a JSON object
- yajl_object_loop() can loop through all sub-fields in a JSON object
- yajl_get_string(), yajl_get_number() and yajl_is_true() will retrieve values of different types of subfields.

Retrieving/Processing JSON

The Customer Maintenance Web Service also supports JSON instead of XML. It works exactly the same, except:

- data is json instead of xml (of course)
- URL is `http://your-server/webservices/json/cust`

```
dcl-c BASEURL 'http://localhost:8500/webservices/json/cust';
jsonData = http_string( 'GET': BASEURL + '/' + %char(custno));

docNode = yajl_buf_load_tree( %addr(jsonData:*data)
                            : %len(jsonData)
                            : ErrMsg );
if errMsg <> '';
   msg = errMsg;
   return *off;
endif;

node = yajl_object_find( docNode: 'success' );
if yajl_is_false(node);
   node = yajl_object_find( docNode: 'errorMsg');
   errMsg = yajl_get_string(node);
endif;
```

http_string() works the same as before. The data is JSON because that's what the server returns for this URL.

yajl_buf_load_tree loads the JSON into memory

yajl_object_find here finds the'success' and 'errorMsg' fields in the JSON document.

Processing JSON Objects (1/2)

```
dataNode = yajl_object_find( docNode: 'data' );

i = 0;
dow yajl_object_loop(dataNode: i: field: custNode);

   select;
   when field = 'name';

      exsr load_field;

   when field = 'address';

      j = 0;
      dow yajl_object_loop(custNode: j: field: addrNode);
         exsr load_field;
      enddo;

   endsl;

enddo;

yajl_tree_free(docNode);
```

Things are set up so that yajl_object_loop loops through all fhe fields needed on the screen.

Then for each field, calls load_field

yajl_tree_free frees up the memory reserved by yajl_buf_load_tree

Processing JSON Objects (2/2)

```
begsr load_field;

   select;
   when field = 'custno';
         custno = yajl_get_number(custNode);
   when field = 'name';
      name = yajl_get_string(custNode);
   when field = 'street';
      street = yajl_get_string(addrNode);
   when field = 'city';
      city = yajl_get_string(addrNode);
   when field = 'state';
      state = yajl_get_string(addrNode);
   when field = 'postal';
      postal = yajl_get_string(addrNode);
   endsl;

endsr;
```

load_field gets the values of each json field from it's node.

It does this by calling yajl_get_number() or yajl_get_string() as appropriate for the data type.

Generating JSON Data

```
yajl_genOpen(*on);
yajl_beginObj();                              //  {

yajl_addBool('success': *on);                 //    "success": true,
yajl_beginObj('data');                        //    "data": {

if orig.name <> name;
   yajl_addChar('name': %trim(name));         //       "name": "xxxx",
endif;

yajl_beginObj('address');                     //       "address": {

if orig.street <> street;
   yajl_addChar('street': %trim(street));     //          "street": "XXX",
endif;

... code above repeats for all other fields ...

yajl_endObj();                                //       }
yajl_endObj();                                //    }
yajl_endObj();                                //  }
```


Sending the JSON

```
%len(jsonData) = %len(jsonData:*MAX);
yajl_copyBuf( JOB_CCSID
            : %addr(jsonData:*data)
            : %len(jsonData)
            : newLen );
%len(jsonData) = newLen;

yajl_genClose();

if isNew;
   method = 'POST';
   url = BASEURL;
else;
   method = 'PUT';
   url = BASEURL + '/' + %char(custno);
endif;

monitor;
   http_string( method: url: jsonData
              : 'application/json' );
on-error;
   msg = http_error();
   return *off;
endmon;
```

yajl_copyBuf() copies the resulting JSON data into our variable.

The %len and %addr logic lets a VARCHAR field be loaded from a pointer to a memory buffer.

yajl_genClose() frees up the memory that YAJL was using for our generated document.

The logic to send the data is the same as the XML example, except the type is now 'application/json'

For More About YAJL

As mentioned earlier, I don't have enough time to explain all of the details of YAJL in this talk. However, I do have another talk that focuses entirely on YAJL.

Working with JSON in RPG Using Open Source Tools

The handout for that talk can be found on my web site:
http://www.scottklement.com/presentations/

You can also download YAJL from my web site:
http://www.scottklement.com/yajl/

SOAP Web Services

SOAP web services are XML based, and very highly standardized. They require two different types of XML document.

SOAP = Simple Object Access Protocol

SOAP is not only the name of the standard, but also the name of one of the XML formats that is used. This is the format that is sent/received with the parameter data.

These are sent over the network with the POST http method

WSDL = Web Services Description Language

Pronounced "whiz-dull".

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. A WSDL document contains everything your program needs to know in order to call a SOAP-based web service.

Generally, a SOAP web service provider will give you the link to his WSDL. You can then use a tool (like SoapUI) to get all of the needed information about the web service.

Currency Exchange Example

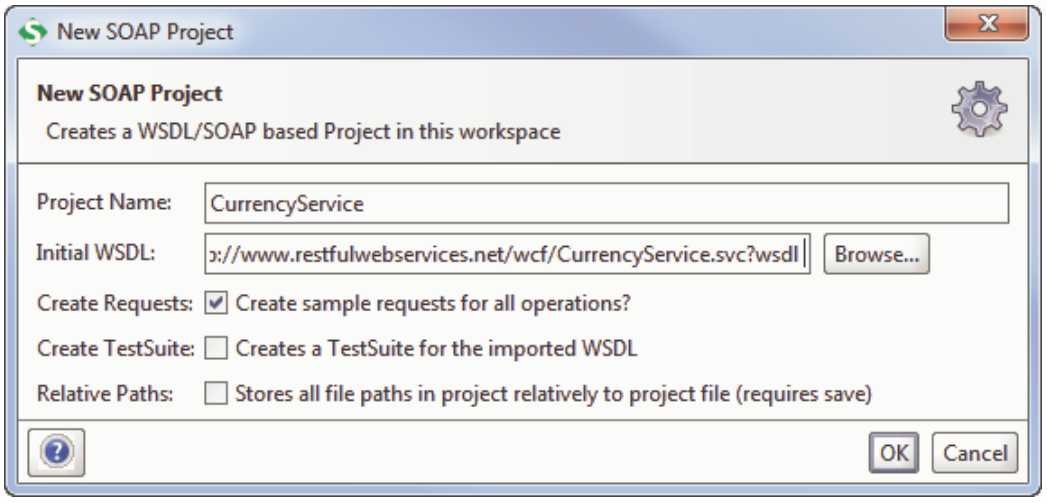
This example will use a publicly available SOAP web service called "Currency Convertor" found at http://www.restfulwebservices.net

Despite the name of the site, this web service is available in SOAP format. (REST is available as well, though.)

To try it:

- Navigate to http://www.restfulwebservices.net
- Click "Currency Convertor"
- Scroll down to the "SOAP" section
- Copy the WSDL URL into your clipboard
- In SoapUI click "SOAP"
- Paste the WSDL URL into the 'Initial WSDL' field
- Click "OK"

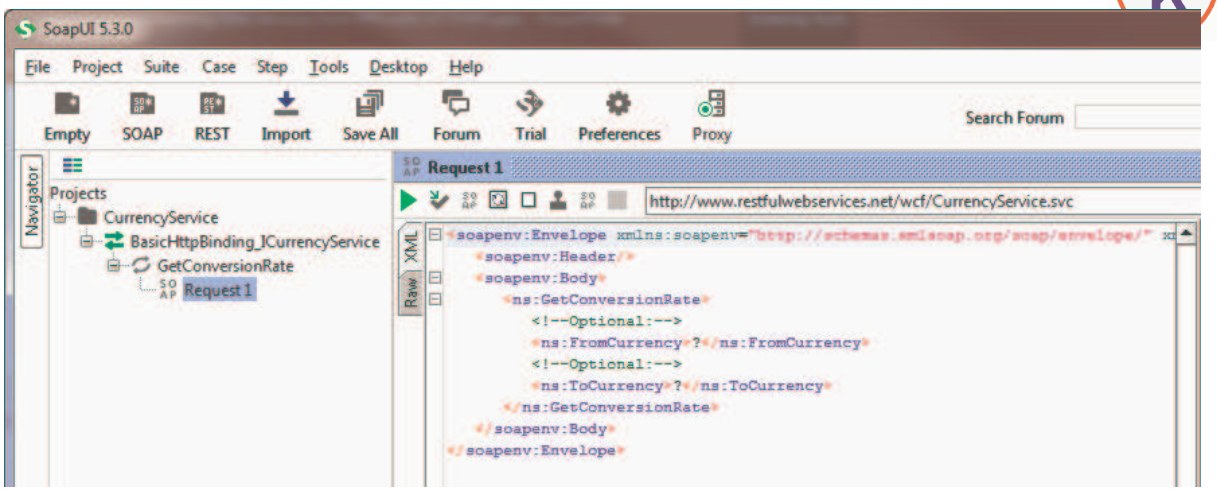
Currency Exchange Example



The WSDL is pasted into the "Initial WSDL" field (second field, above)

Click "OK" to see the result.

Currency Exchange Example



- Expand the tree on the left to get to "Request 1".
- Double-click "Request 1" to see the XML
- Fill in the "from" and "to" currencies, and click the green "play" button to see the response.

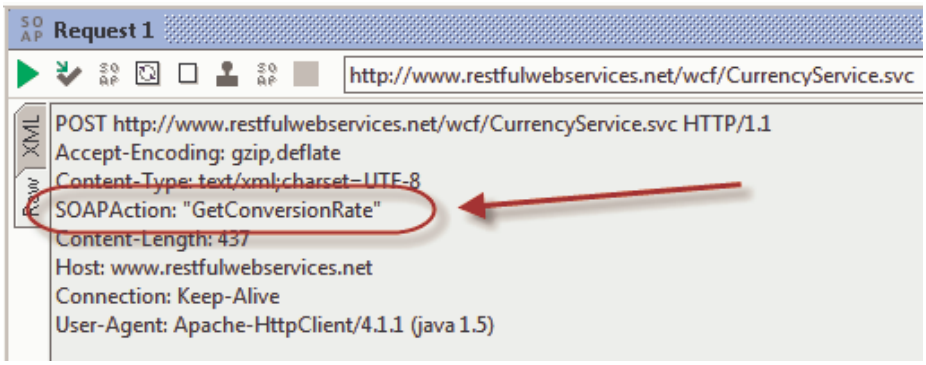
SoapAction

SOAP messages also require a "SOAPAction"

- Tells SOAP what to do with your input
- Similar to how GET/POST/PUT/DELETE is used in REST
- Can be any string the provider wants to use.

To get it from SoapUI:

- Run the web service in SoapUI (using the "play" button)
- Click "Raw" on the box that showed the input SOAP message



Sample SOAP Documents

Here are example SOAP messages that I discovered by running the WSDL through SoapUI. Now that I know what these look like, I can do the same thing from RPG…

Input Message

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:ns="http://www.restfulwebservices.net/ServiceContracts/2008/01">
   <soapenv:Body>
      <ns:GetConversionRate>
         <ns:FromCurrency>USD</ns:FromCurrency>
         <ns:ToCurrency>EUR</ns:ToCurrency>
      </ns:GetConversionRate>
   </soapenv:Body>
</soapenv:Envelope>
```

Output Message

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
   <s:Body>
      <GetConversionRateResponse xmlns="http://www.restfulwebservices.net/ServiceContracts/2008/01">
         <GetConversionRateResult xmlns:a="http://www.restfulwebservices.net/DataContracts/2008/01"
                                  xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
            <a:Rate>0.9446</a:Rate>
         </GetConversionRateResult>
      </GetConversionRateResponse>
   </s:Body>
</s:Envelope>
```


Currency Exchange from RPG

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

D EXAMPLE16       PR                  ExtPgm('EXAMPLE16')
D   Country1                     3A   const
D   Country2                     3A   const
D   Amount                     15P 5  const
D EXAMPLE16       PI
D   Country1                     3A   const
D   Country2                     3A   const
D   Amount                     15P 5  const

 /copy httpapi_h

D URL             s            100a   varying
D SOAP            s           1000A   varying
D response        s           1000a   varying
D rate            s              9p 4
D Result          s             12P 2

 /free

  URL = 'http://www.restfulwebservices.net/wcf/CurrencyService.svc';

  http_setOption('SoapAction': '"GetConversionRate"');
```

Currency Exchange from RPG

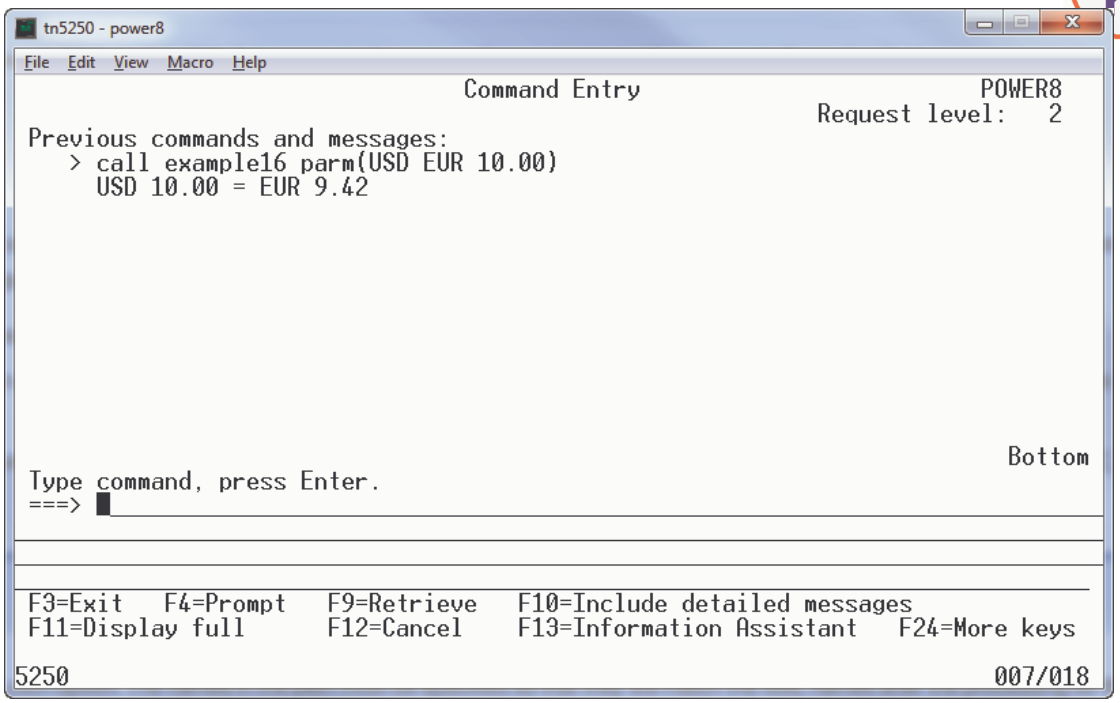
```
SOAP =
'<soapenv:Envelope +
   xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" +
   xmlns:ns="http://www.restfulwebservices.net/+
   ServiceContracts/2008/01">+
 <soapenv:Header/>+
 <soapenv:Body>+
    <ns:GetConversionRate>+
       <ns:FromCurrency>'+%trim(Country1)+'</ns:FromCurrency>+
       <ns:ToCurrency>'+%trim(Country2)+'</ns:ToCurrency>+
    </ns:GetConversionRate>+
 </soapenv:Body>+
 </soapenv:Envelope>';

 response = http_string( 'POST': URL: SOAP: 'text/xml');

 xml-into rate %xml(response: 'case=any ns=remove +
   path=Envelope/Body/GetConversionRateResponse+
        /GetConversionRateResult/Rate');

 Result = Amount * Rate;

 http_comp( %trim(Country1) + ' ' + %char(Amount)
          + ' = '
          + %trim(Country2) + ' ' + %char(Result));
```

The ns=remove option strips the namespaces.

The path= option lets us extract one piece of the document.

http_comp sends a "completion" message

Currency Exchange from RPG



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

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, EXAMPLE2, EXAMPLE3, etc..

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/

More Information / Resources

w3schools.com **-- free (and great!) site for learning web technology**

XML:	**http://www.w3schools.com/xml/default.asp**
Web Services:	**http://www.w3schools.com/webservices/default.asp**
WSDL:	**http://www.w3schools.com/wsdl/default.asp**
SOAP:	**http://www.w3schools.com/soap/default.asp**

IBM's web site for the Integrated Web Services (IWS) tool:

http://www.ibm.com/systems/i/software/iws/
http://www.ibm.com/systems/i/software/iws/quickstart_server.html

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

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

This Presentation

You can download a PDF copy of this presentation, as well as other related materials from:

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

Thank you!