Consuming Web Services from RPG

with HTTPAPI

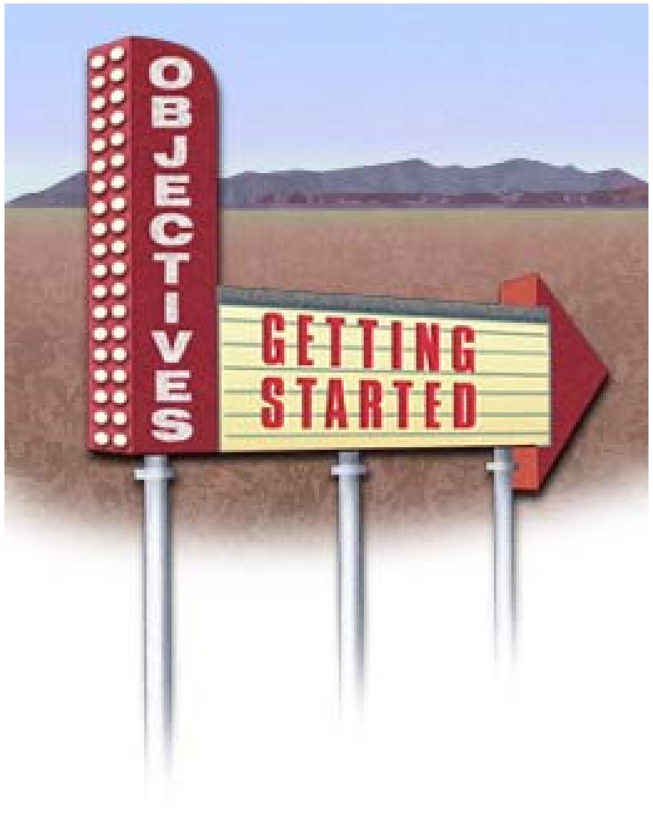
Presented by

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
```

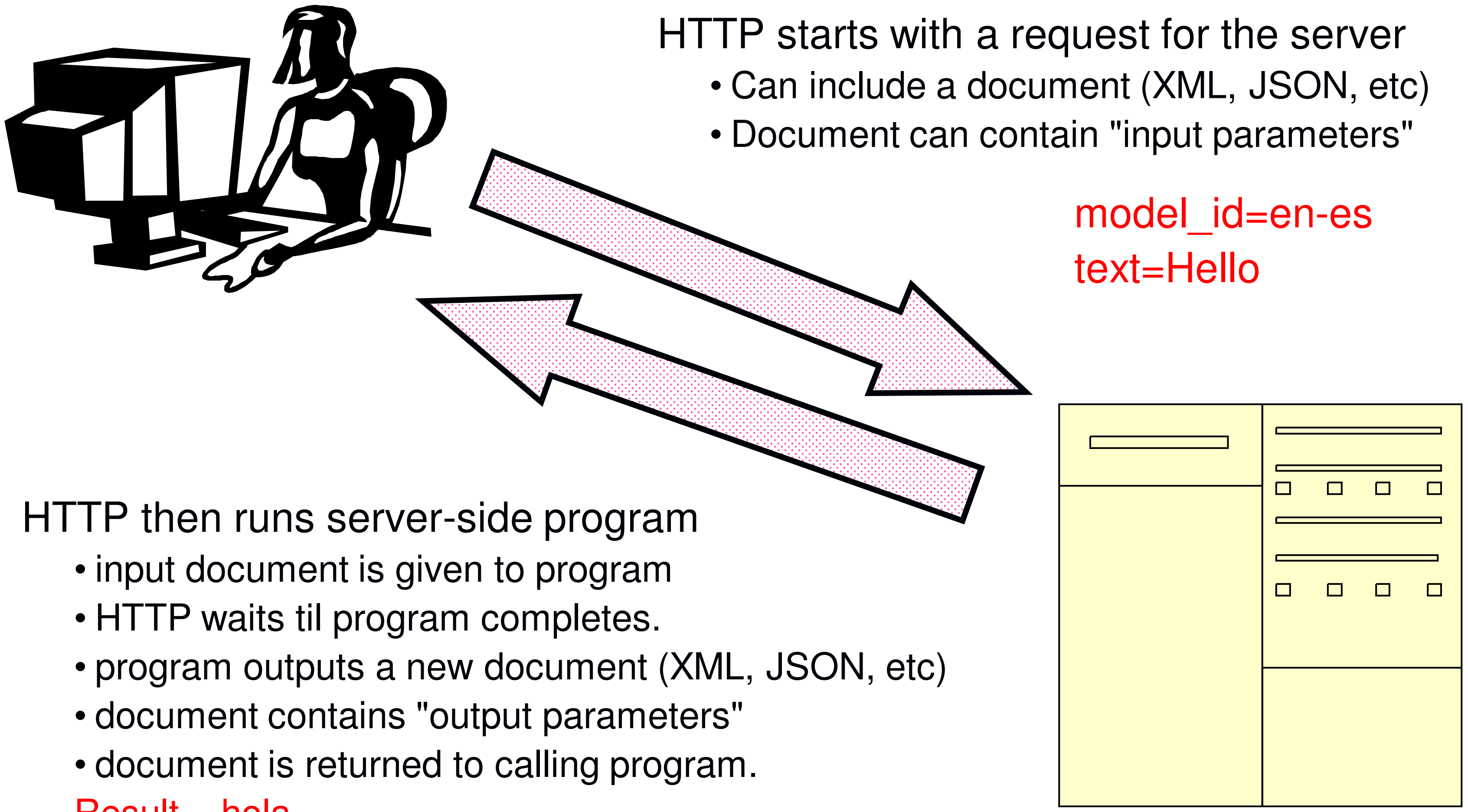
Output parameter:

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.

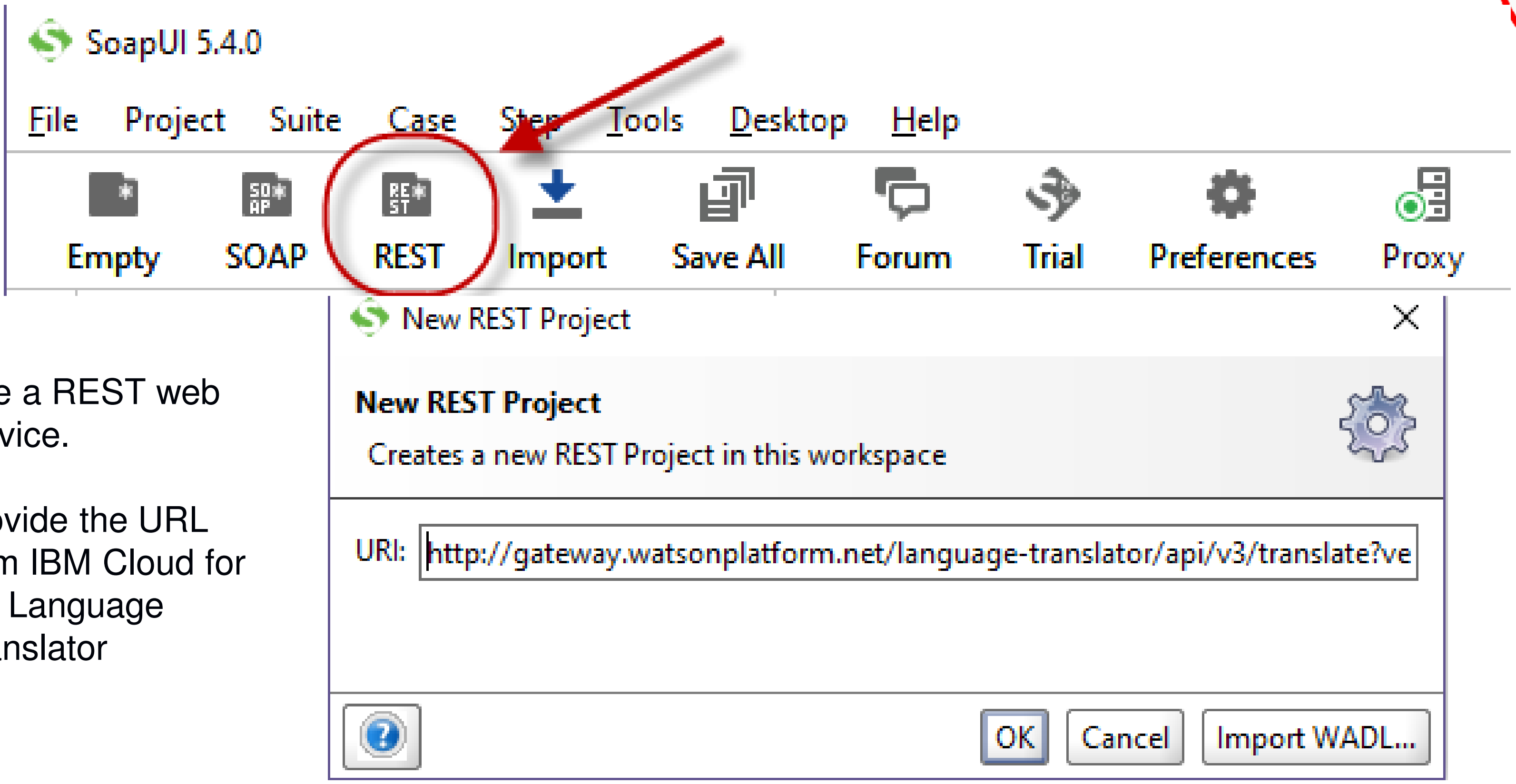
Result = hola

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


SoapUI 5.4.0
File Project Suite Case Step Tools Desktop Help
Empty SOAP REST Import Save All Forum Trial Preferences Proxy
New REST Project
OK Cancel Import WADL...

New REST Project

Creates a new REST Project in this workspace

URI: http://gateway.watsonplatform.net/language-translator/api/v3/translate?ve

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

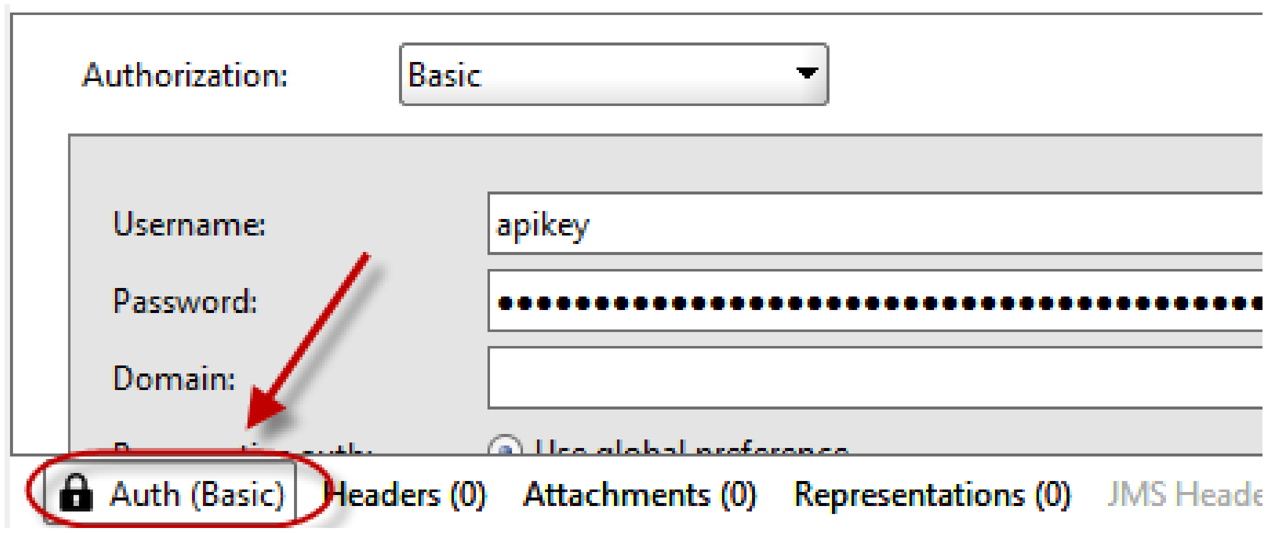
Note: This URL is too long to appear on the screen, but the box scrolls left/right to fit it all.

The full URL is http://gateway.watsonplatform.net/language-translator/api/v3/translate?version=2018-05-01

Authorizing SoapUI

Watson requires you to have an account set up on IBM Cloud that is used to run this service.

In SoapUI you can put your login credentials (usually 'apikey' for the userid plus your password) under 'Auth' at the bottom.



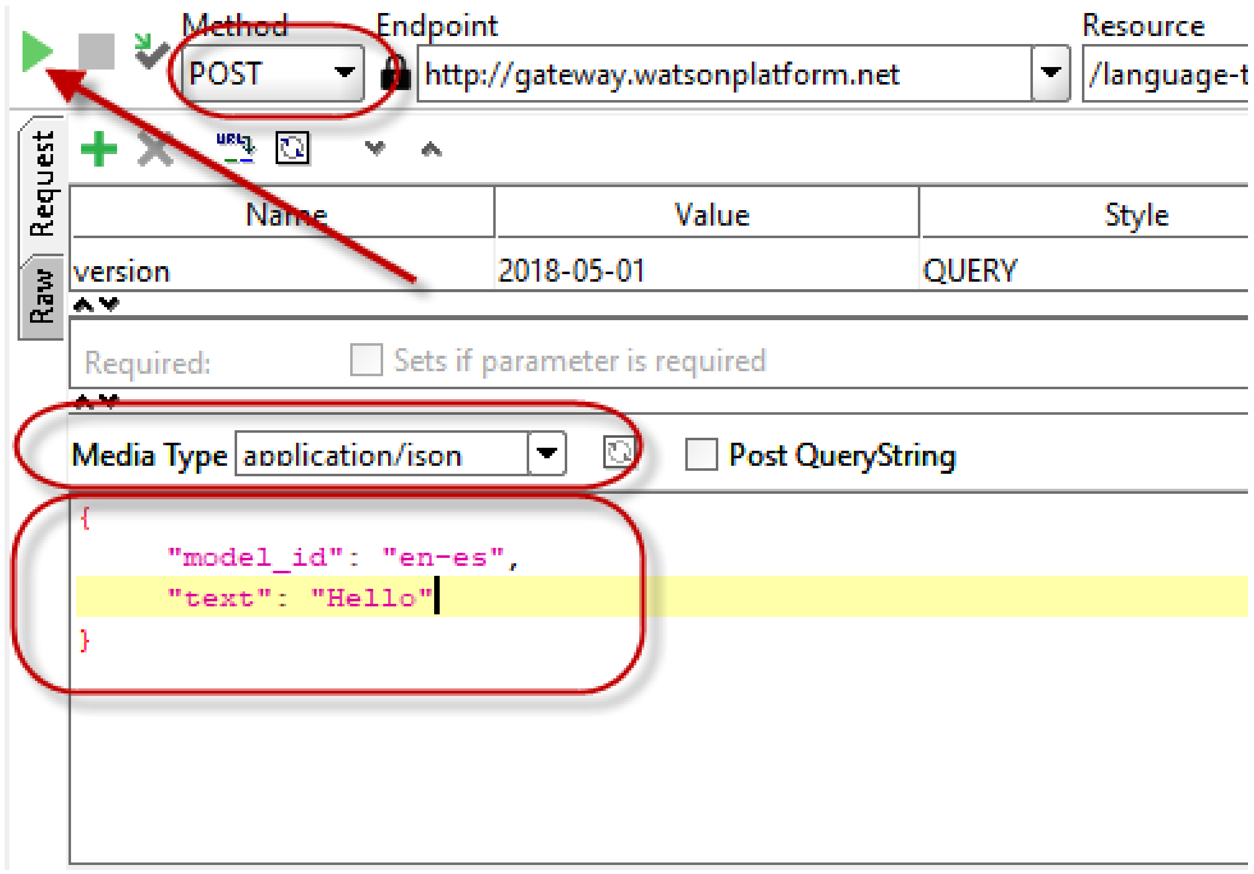
Authorization: Basic

Username: apikey

Password: ●●

Domain:

Trying It Out in SoapUI



Method: POST

Endpoint: http://gateway.watsonplatform.net

Resource: /language-t

Name	Value	Style
version	2018-05-01	QUERY

Required: Sets if parameter is required

Media Type: application/json

Post QueryString

```
{
    "model_id": "en-es",
    "text": "Hello"
}
```

- Use the "method" dropdown to pick "POST"
- Make sure the media type is "application/json"
- Type the parameters in JSON format into the box
- Click the green "Play" button (upper-left) to run it.

Results

```
{
  "translations": [{
    "translation": "Hola"
  }],
  "word_count": 1,
  "character_count": 5
}
```

- On the right you have tabs to view the result as "Raw", "HTML", "JSON" or "XML
- Watson services use JSON (as do most newer APIs)
- The result is shown in the box.

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/IBM Cloud example.

Using It Yourself

- Fully supports commercial use
- 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: 'apikey': '{your-api-key}');

request = '{"text":["Hello"],"model_id":"en-es"}';

url = 'https://gateway.watsonplatform.net/language-translator/api'
      + '/v3/translate?version=2018-05-01

response = http_string('POST': url: request: 'application/json');

DATA-INTO result %DATA(response) %PARSER('YAJLINTO');
```

http_setAuth() – sets the userid/password used.

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

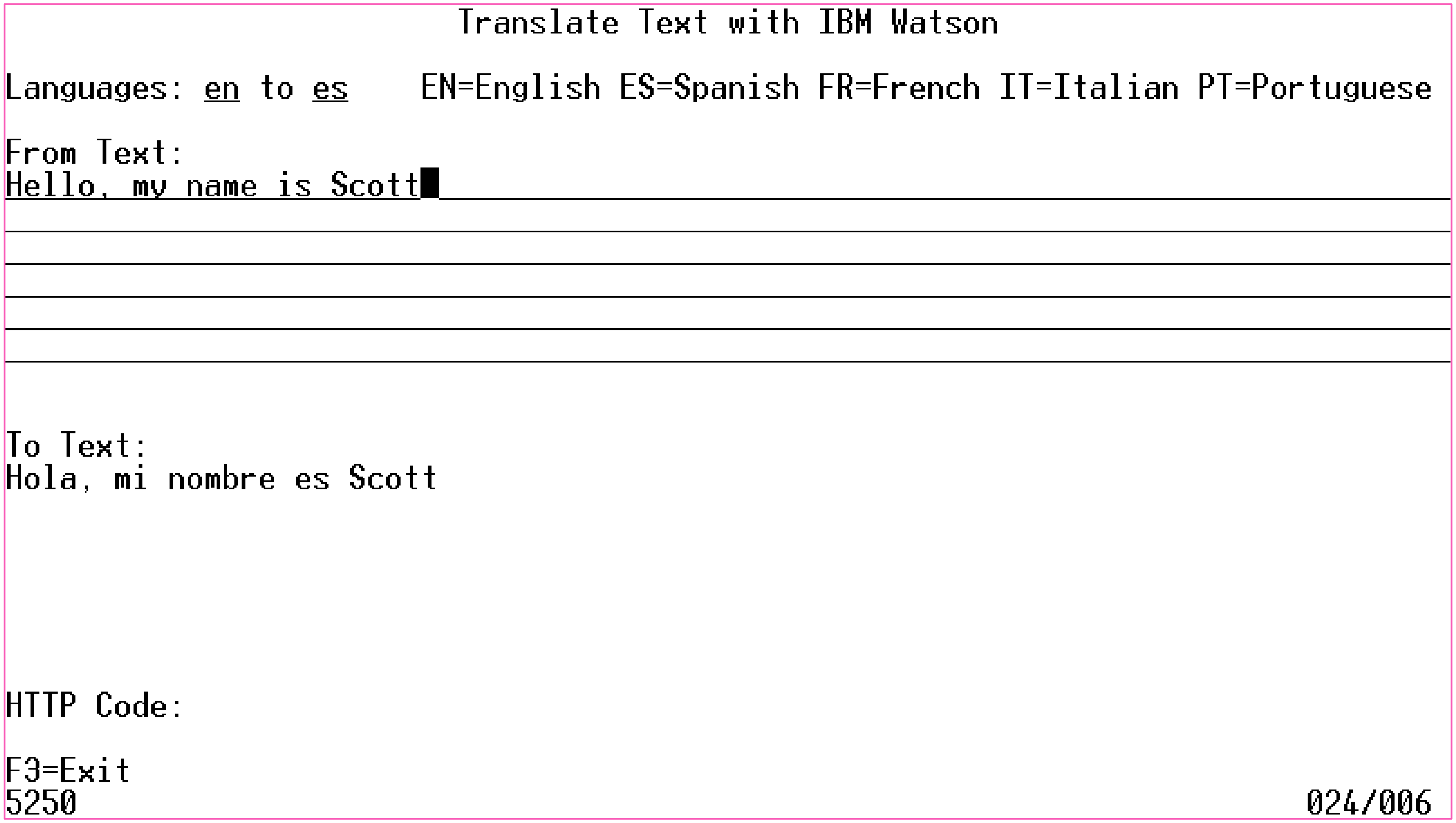
DATA-INTO – RPG opcode for parsing documents such as JSON

`request`, **`url`** and **`response`** are standard RPG VARCHAR fields. (CHAR would also work)

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

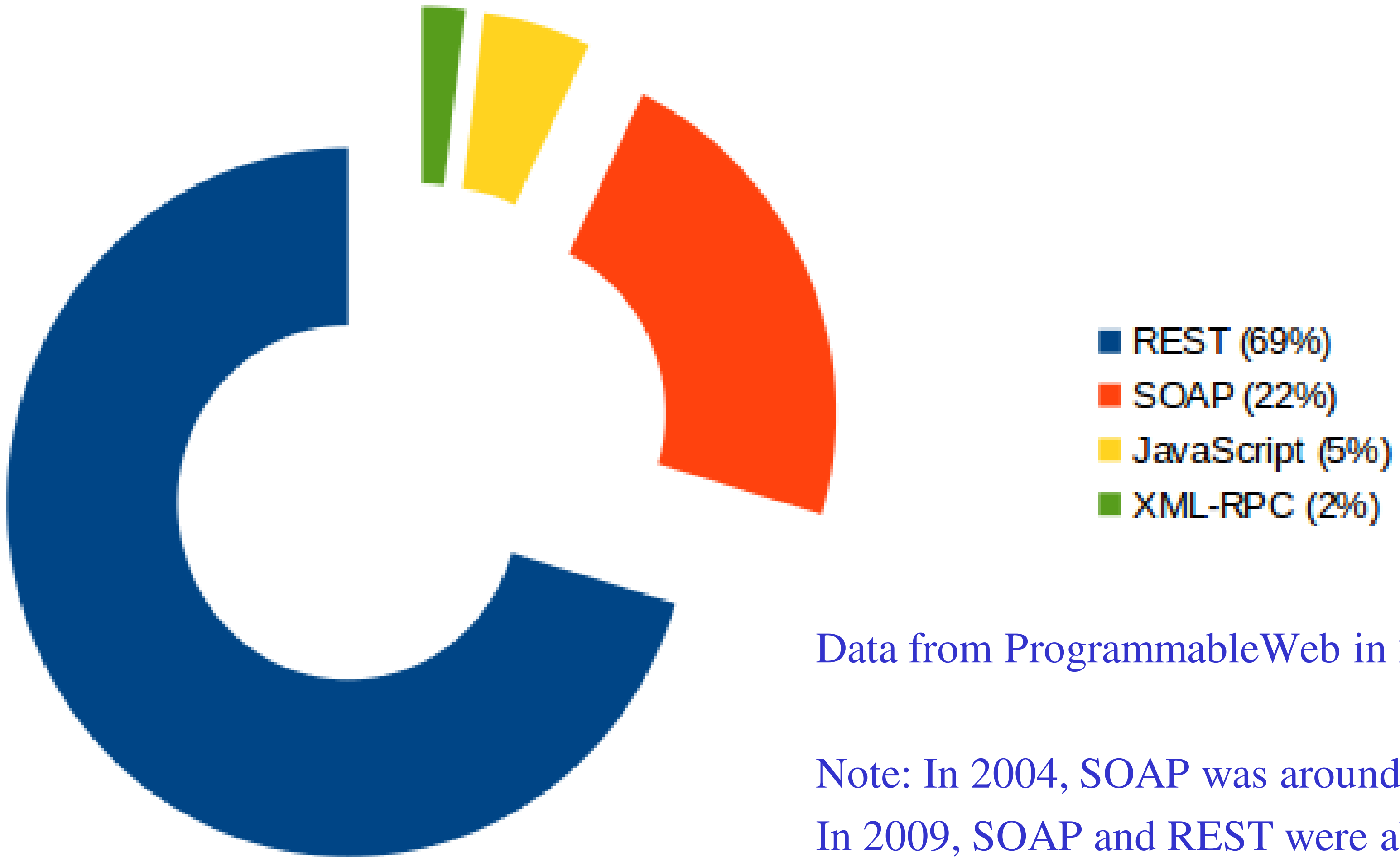
REST

- 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.

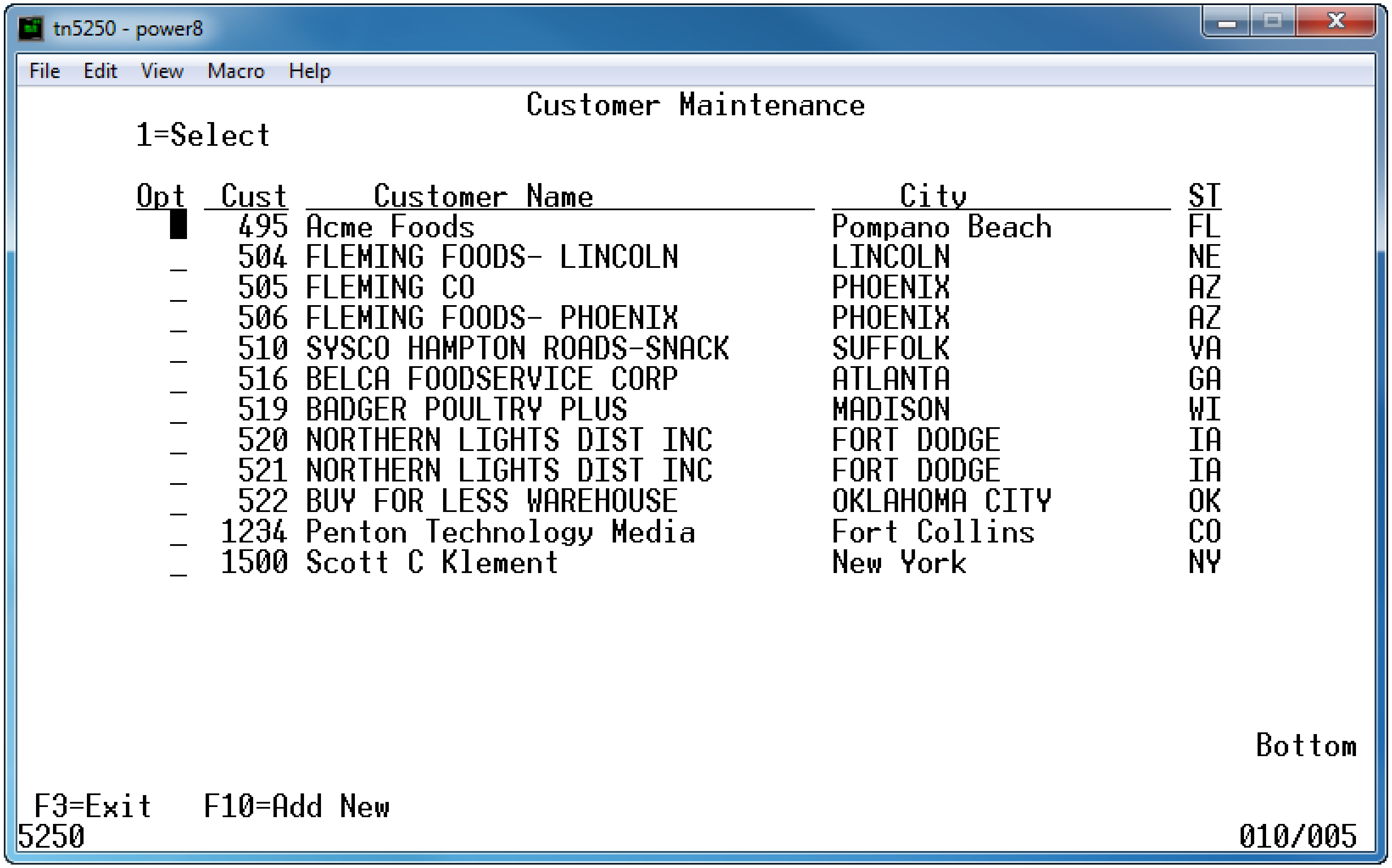
```
http://my-server/webservices/xml/cust/1234
```

- 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.



```
                              Customer Maintenance
       1=Select

       Opt  Cust  Customer Name                 City                 ST
        █    495  Acme Foods                    Pompano Beach        FL
        _    504  FLEMING FOODS- LINCOLN        LINCOLN              NE
        _    505  FLEMING CO                    PHOENIX              AZ
        _    506  FLEMING FOODS- PHOENIX        PHOENIX              AZ
        _    510  SYSCO HAMPTON ROADS-SNACK     SUFFOLK              VA
        _    516  BELCA FOODSERVICE CORP        ATLANTA              GA
        _    519  BADGER POULTRY PLUS           MADISON              WI
        _    520  NORTHERN LIGHTS DIST INC      FORT DODGE           IA
        _    521  NORTHERN LIGHTS DIST INC      FORT DODGE           IA
        _    522  BUY FOR LESS WAREHOUSE        OKLAHOMA CITY        OK
        _   1234  Penton Technology Media       Fort Collins         CO
        _   1500  Scott C Klement               New York             NY




                                                                         Bottom

 F3=Exit    F10=Add New
5250                                                                    010/005
```

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.

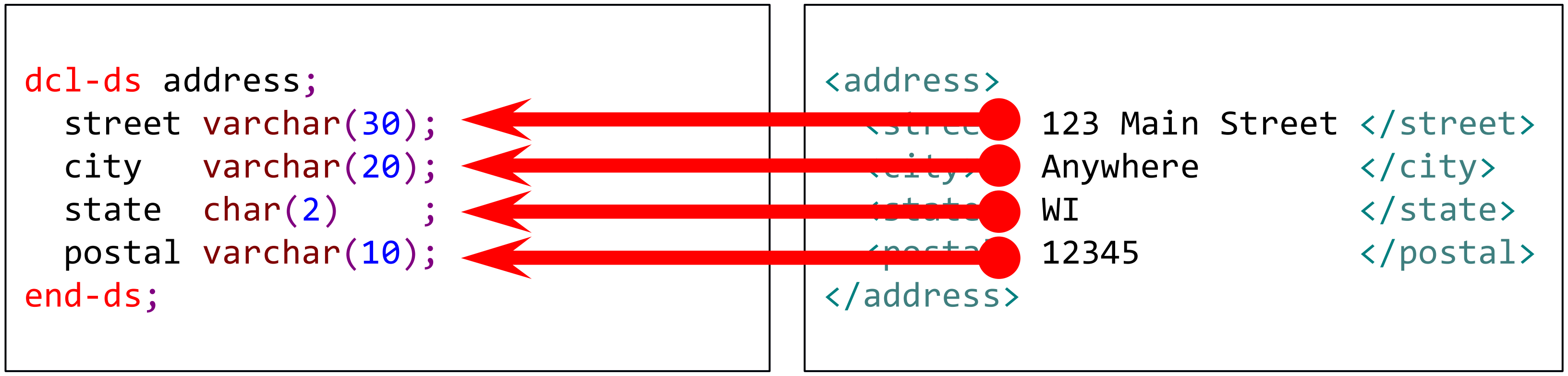
```
dcl-c BASEURL 'http://localhost:8500/webservices/xml/cust';
dcl-s xmlData varchar(100000);

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.

Think of XML Like a Data Structure

Think of XML like a data structure, it's one larger field (such as "address") that contains sub-fields (such as "street", "city", "state" and "postal")

It'd be helpful to be able to load the RPG DS from the XML.



```
dcl-ds address;
  street varchar(30);
  city   varchar(20);
  state  char(2)    ;
  postal varchar(10);
end-ds;
```

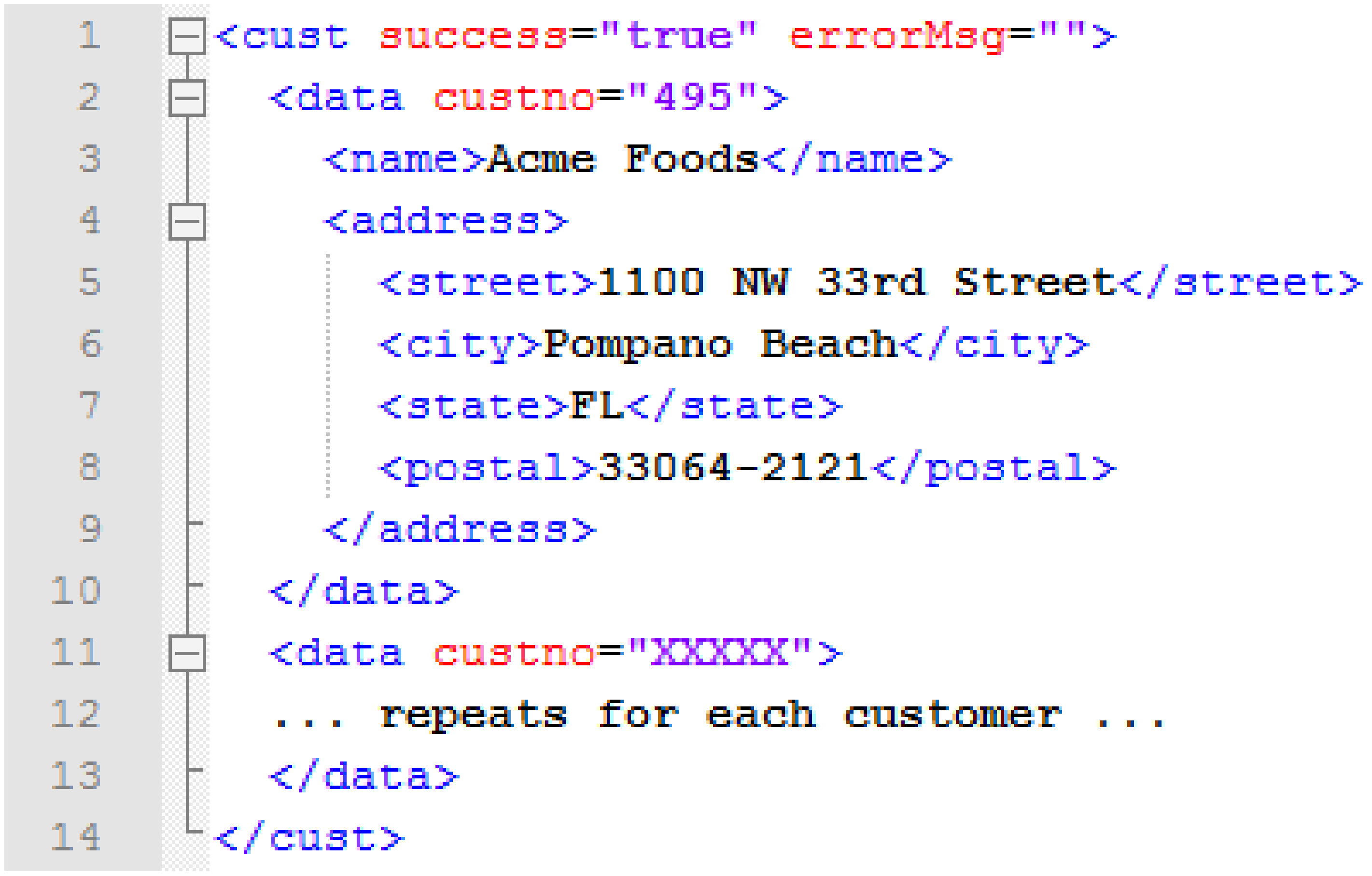
```
<address>
  <street> 123 Main Street </street>
  <city>   Anywhere        </city>
  <state>  WI              </state>
  <postal> 12345           </postal>
</address>
```

That's what XML-INTO does!

- Maps XML fields into corresponding DS fields
- Field names must match (special characters can be mapped into underscores if needed)
- Repeating elements can be loaded into arrays

The List of Customers Looks Like This

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

- This RPG data structure has the same definition as the XML in our web service.
- “data” is an array that can have up to 999 addresses.
- XML-INTO’s “countprefix” can tell us how many were found

```
dcl-ds cust qualified;
   success  ind                inz(*on);
   errorMsg varchar(500)       inz('');

   dcl-ds data dim(999);
      custno packed(5: 0)    inz(0);
      name   varchar(30)     inz('');
      dcl-ds address;
         street varchar(30) inz('');
         city   varchar(20) inz('');
         state  char(2)     inz('  ');
         postal varchar(10) inz('');
      end-ds;
   end-ds;
end-ds;

xml-into cust %xml(xmlData:'case=convert                     ');
```

“num_data” is “data” with the “num_” count prefix added, so RPG will count the number of “data” array elements, and put it into “num_data”

Loading the List Into the Subfile

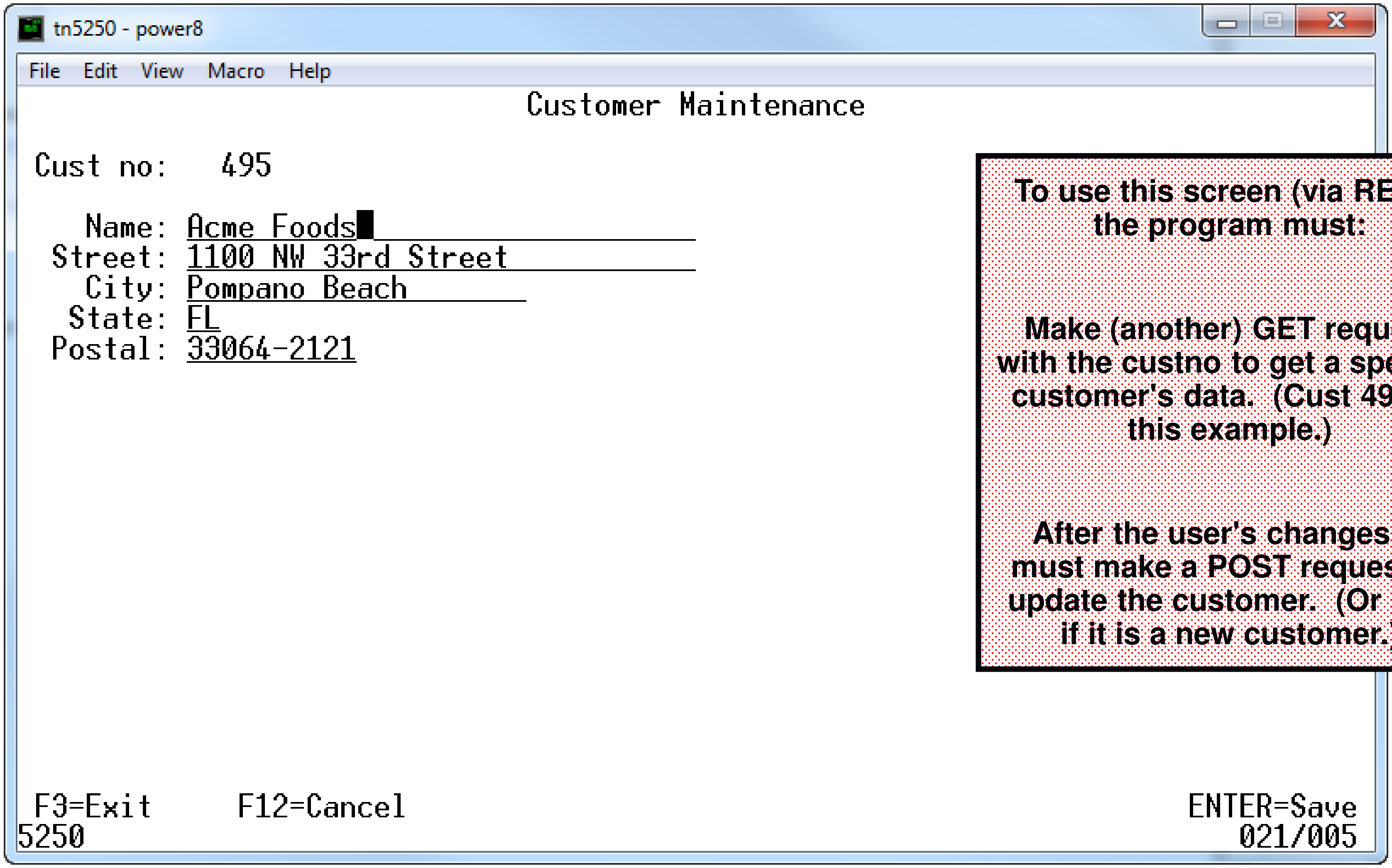
```
dcl-ds cust likeds(cust_t) inz(*likeds);

xml-into cust %xml(xmlData:'case=convert countprefix=num_');

clearSfl();

for i = 1 to cust.num_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



```
                          Customer Maintenance

 Cust no:    495

    Name: Acme Foods
  Street: 1100 NW 33rd Street
    City: Pompano Beach
   State: FL
  Postal: 33064-2121




 F3=Exit      F12=Cancel                                     ENTER=Save
5250                                                          021/005
```

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=convert countprefix=num_');

// 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.

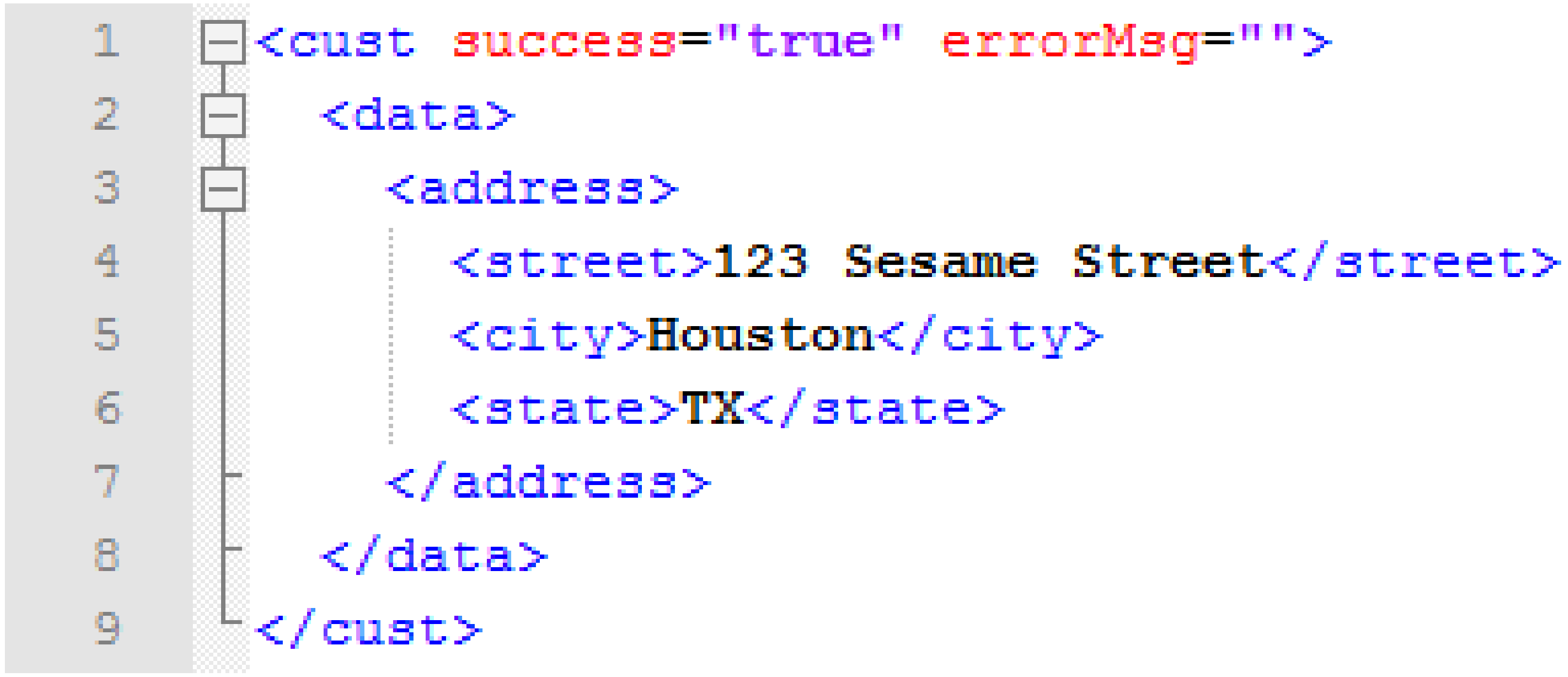
NOT SHOWN: If the user hit F10 = new customer, we skip this and just blank out the screen fields and "orig"

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:



```xml
<cust success="true" errorMsg="">
  <data>
    <address>
      <street>123 Sesame Street</street>
      <city>Houston</city>
      <state>TX</state>
    </address>
  </data>
</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 &lt; 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

However, it has DATA-INTO and DATA-GEN!

- DATA-INTO is like XML-INTO, maps a structured document into a DS, array, etc
- Requires IBM i 7.2 or newer (PTF needed for 7.2 and 7.3)
- A 3rd-party tool that understands the document format is needed
- (free) YAJL open source project has a 'YAJLINTO' tool for DATA-INTO

```
DATA-INTO ResultVariable
          %DATA(document: 'options')
          %PARSER('3RD-PARTY-PROGRAM': 'options’)
```

Because RPG doesn't interpret the document, it's possible to get a DATA-INTO parser for any structured format.

- JSON, XML, CSV, Property files, etc
- We'll use it with YAJLINTO for JSON

What is YAJL?

YAJL = Yet Another Json Library

- Very fast JSON reader
- *Completely Open Source* (free of charge)
- Cross-platform C code written by Lloyd Hilael of Mozilla
    - YAJL *SRVPGM = ILE C port of YAJL
- IBM i (ILE RPG) front end by Scott Klement
    - YAJLR4 *SRVPGM = ILE RPG front-end
- DATA-INTO interface program
    - YAJLINTO *PGM = DATA-INTO parser

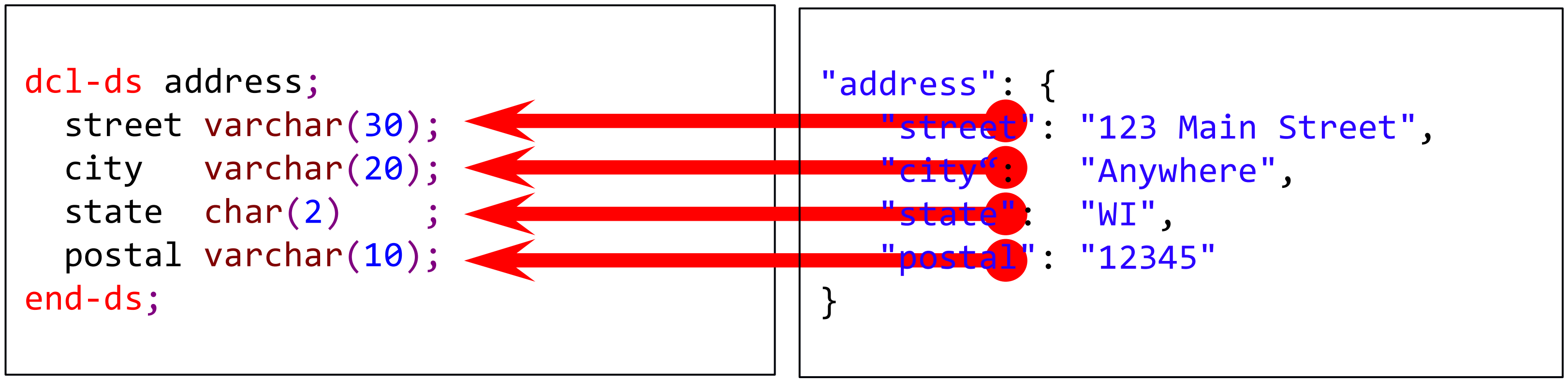
Download from Scott's web site:
http://www.scottklement.com/yajl/

Requires IBM i 6.1 or newer.   IBM i 7.2 or newer needed for DATA-INTO

Mapping JSON Format

JSON format:

- The { } characters indicate an “object” (same as RPG data structure)
- The [ ] characters indicate an array
- Just as with XML, we can map them into an RPG structure



```
dcl-ds address;
  street varchar(30);
  city   varchar(20);
  state  char(2)    ;
  postal varchar(10);
end-ds;
```

```
"address": {
   "street": "123 Main Street",
   "city":   "Anywhere",
   "state":  "WI",
   "postal": "12345"
}
```

DATA-INTO will do that when used with YAJLINTO (or similar)

- Aside from needing the 3rd party parser, it’s almost identical to XML-INTO
- Options like case=convert and countprefix work here as well

Retrieving/Processing JSON

Communication is still done with HTTPAPI (or an alternative)

- URL is `http://your-server/webservices/json/cust`
- http_string() receives the JSON data into a variable (instead of XML)
- Here's the example of retrieving the customer list
- The RPG definition of the "cust" data structure is the same as the XML version

```
dcl-c BASEURL 'http://localhost:8500/webservices/json/cust';

jsonData = http_string( 'GET': BASEURL);

data-into cust %DATA( jsonData
                    : 'case=convert countprefix=num_')
                    %PARSER('YAJLINTO');
```

YAJL Can Also Generate JSON

YAJL can also be used with RPG's DATA-GEN opcode to generate JSON documents.

```
DATA-GEN rpg-variable %DATA(result-variable : options)
                      %GEN(generator-program : options);
```

DATA-GEN was added to RPG in November 2019, via PTFs for 7.3/7.4

- DATA-GEN converts an RPG variable to a corresponding structured document. In this example, we'll generate a JSON document.
- The first parameter ("factor1") specifies the RPG variable containing the data to be converted.
- The %GEN BIF controls which program (think of it like a "driver") is used to generate the document.
  - YAJLDTAGEN is a %GEN program for generating JSON documents, included with YAJL. The YAJLDTAGEN program is what determines that the output is JSON (vs. other formats like XML, YAML, etc.)
- The %DATA BIF specifies the variable the result is placed in
- Many, many other options are available, just scratching the surface., here.

RPG Variable That Matches the JSON

DATA-GEN is basically DATA-INTO in reverse. You provide an RPG variable, and it creates JSON that matches that variable's layout, subfield names, etc.

```
dcl-ds cust qualified;                           // {
     success   ind               inz(*on);     //   "success": true,
     errorMsg varchar(500)      inz('');      //   "errorMsg": "{string}",
     dcl-ds data;                                //   "data": {
        num_custno int(10)        inz(0);
        custno     packed(5: 0) inz(0);      //     "custno": {number},
        num_name   int(10)        inz(0);
        name       varchar(30)  inz('');     //     "name": {string},
        dcl-ds address;                          //     "address": {
           num_street int(10)    inz(0);
           street   varchar(30)  inz('');     //       "street": "{string}",
           num_city   int(10)    inz(0);
           city     varchar(20)  inz('');     //       "city": "{string}",
           num_state  int(10)    inz(0);
           state    char(2)      inz('  '); //       "state": "{string}",
           num_postal int(10)    inz(0);
           postal   varchar(10)  inz('');     //       "postal": "{string}"
        end-ds;                                  //     }
     end-ds;                                     //   }
  end-ds;                                        // }
```

The num_xxx fields work with the 'countprefix' option to specify how many of a field should be generated.

Using CountPrefix to Omit Elements

DATA-GEN is basically DATA-INTO in reverse. You provide an RPG variable, and it creates JSON that matches that variable's layout, subfield names, etc.

```
if orig.name <> name;
   cust.data.num_name = 1;
   cust.data.name = %trim(name);
endif;

if orig.street <> street;
   cust.data.address.num_street = 1;
   cust.data.address.street     = %trim(street);
endif;
if orig.City <> city;
   cust.data.address.num_city = 1;
   cust.data.address.city     = %trim(city);
endif;
```

... (code above is repeated for state, postal, etc) ...

```
data-gen cust %data(jsonData: 'countprefix=num_')
              %gen('YAJLDTAGEN');
```

We only want to include fields in the JSON document that have been changed.

Num_ is the "countprefix", which means num_name is the number of name elements, num_street is the number of street elements, etc. You can set it to 0 if you want a field to be omitted.

DATA-GEN will place the JSON data into the 'jsonData' variable specified by %DATA

Calling DATA-GEN

Here is the DATA-GEN call from the last slide:

```
   data-gen cust %data(jsonData: 'countprefix=num_')
                 %gen('YAJLDTAGEN');
```

- The first parameter, "cust" is the name of the data structure to generate from.
- The %DATA BIF tells DATA-GEN to put the output into the jsonData variable. The second parameter is for options, we're using countprefix to enable support for including/omitting elements we don't want.
- The %GEN BIF controls which program (think of it like a "driver") is used to generate the document.
  - YAJLDTAGEN is a %GEN program for generating JSON documents, included with YAJL. The YAJLDTAGEN program is what determines that the output is JSON (vs. other formats like XML, YAML, etc.)
- There are many other options available to both DATA-GEN and YAJLDTAGEN. See the corresponding documentation for details.

Sending the JSON

```
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;
```

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

For More About YAJL

For people on older releases, YAJL has subprocedures that can be used to both create and interpret JSON documents, but they are not quite as easy to read or understand as DATA-INTO and DATA-GEN.

There are full examples of these on my web site.

As mentioned earlier, I don't have enough time to explain all of the details of YAJL in this talk. However, I do have other talks that focus 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 = Simple Object Access Protocol

Like all web services:

- Involves a send/receive of documents representing parameters.
- SOAP documents are always XML
- Highly standardized, extra XML often required to fit standards
- The "SOAP messages" are the XML documents containing the parameters
- Almost always uses HTTP POST method
- The "verb" comes from a separate header called SoapAction
- WSDL documents (another XML format) are provided to show all of the details of the service

WSDL = Web Services Description Lanuage *(pronounced "whiz-dull")*

- An additional XML format
- The documentation for the web service (instead of Word, PDF, etc docs)
- ...except these docs are meant for a computer to read!

Despite the name, SOAP is complex. SOAP-specific software is almost always needed.

Consuming a SOAP API

You can consume a SOAP Web Service the same way you would consume a REST one with HTTPAPI in RPG.

You will need to know this information:

- The URL ("endpoint") of the service
- The input SOAP message (XML parameter document)
- The output SOAP message
- The SoapAction string needed

Then you can build the SOAP messages in your RPG program and send them with http_string().

All of this information can be gleaned from the WSDL document.

Temperature Convert Example

I previously used public SOAP example sites, but unfortunately these free sites never seem to last, and are taken down. Instead, I will demonstrate an in-house service.

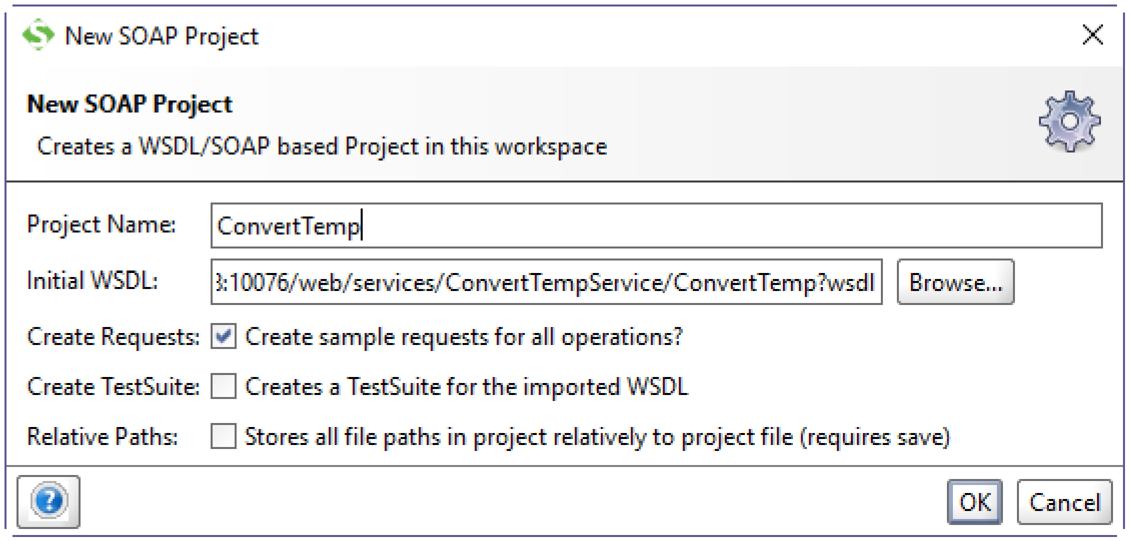
IBM provides an example web service that converts Fahrenheit to Celsius on IBM I on their Integrated Web Services server for IBM i. To learn more about IWS and providing web services (in general) please see my “Providing RPG Web Services on IBM I” session. The handout is online, here:
http://www.scottklement.com/presentations/#PROVIDING

Once the service has been set up, find the WSDL for this service on your IBM i:

- IBM Navigator for i (http://your-system:2001)
- Internet Configurations, IBM Web Administration for I
- Select the IWS server you configured
- Click “Manage Deployed Services”
- There will be a “View WSDL” link next to the ConvertTemp service

Use SoapUI to Read the WSDL

The WSDL is difficult to read in XML format, but much easier using a tool like SoapUI. http://www.soapui.org (available in both free and commercial versions)



New SOAP Project

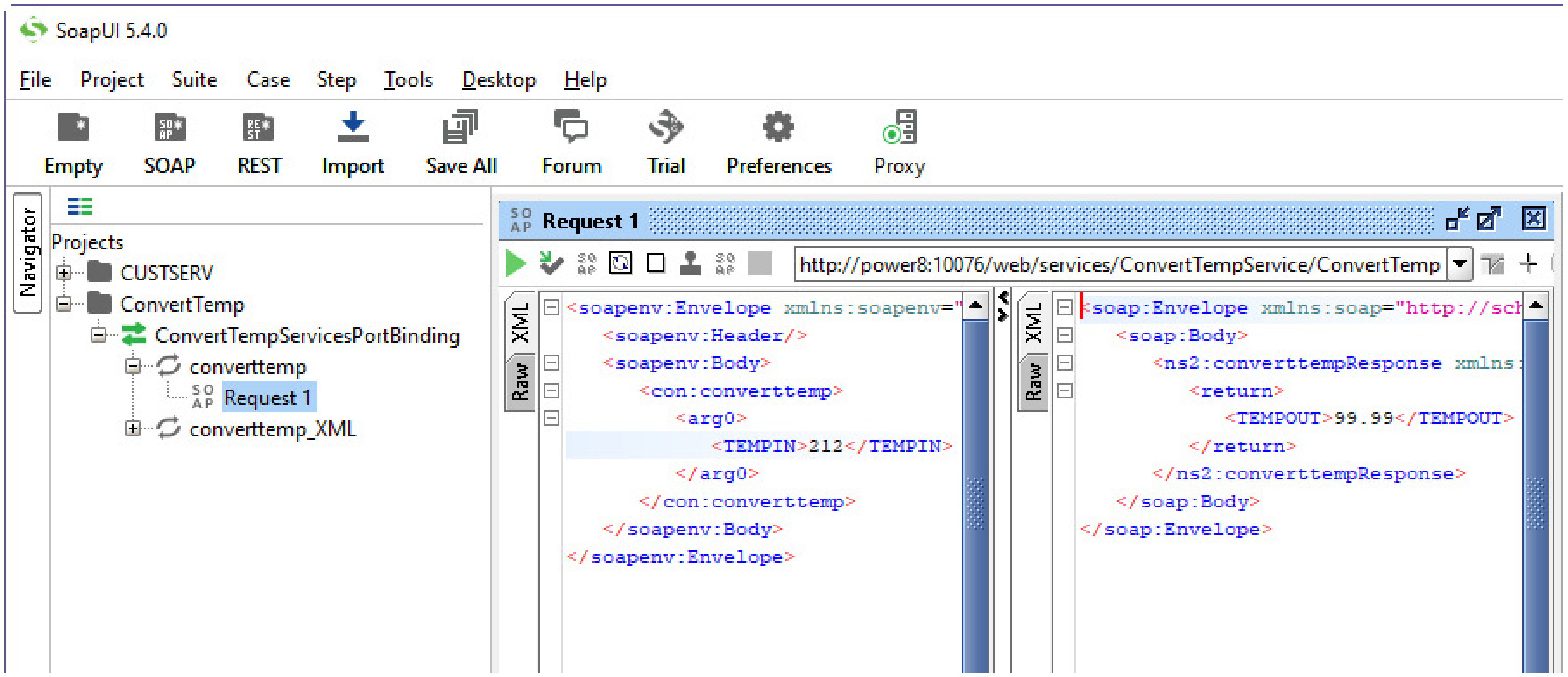
New SOAP Project
Creates a WSDL/SOAP based Project in this workspace

Project Name:	ConvertTemp
Initial WSDL:	:10076/web/services/ConvertTempService/ConvertTemp?wsdl
Create Requests:	[✓] Create sample requests for all operations?
Create TestSuite:	[ ] Creates a TestSuite for the imported WSDL
Relative Paths:	[ ] Stores all file paths in project relatively to project file (requires save)

- Start SoapUI
- Click "SOAP" in the toolbar/ribbon
- Copy/Paste the WSDL link into the "Initial WSDL" field as shown above.

Temperature Convert Example

- Expand the tree by opening “ConvertTempServicesPortBinding”, “converttemp”, and “Request 1”
- Notice that you can read various property details (from the WSDL) for each of the things you click (the binding, operation, request, etc)
- If you double-click “Request 1” it will show you what the SOAP message looks like, and let you try it out.

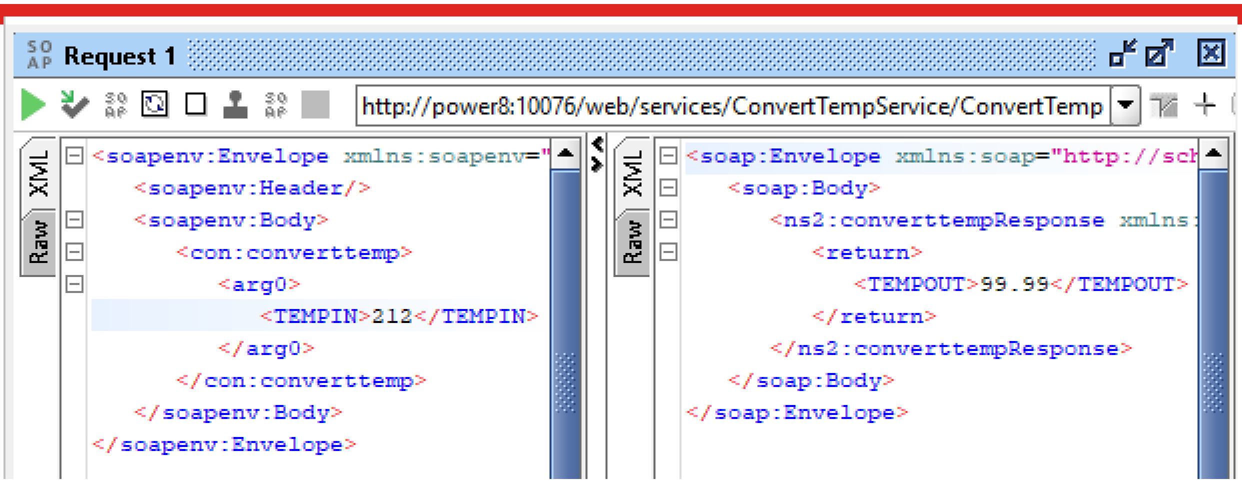


http://power8:10076/web/services/ConvertTempService/ConvertTemp

```
<soapenv:Envelope xmlns:soapenv="
   <soapenv:Header/>
   <soapenv:Body>
      <con:converttemp>
         <arg0>
            <TEMPIN>212</TEMPIN>
         </arg0>
      </con:converttemp>
   </soapenv:Body>
</soapenv:Envelope>
```

```
<soap:Envelope xmlns:soap="http://sch
   <soap:Body>
      <ns2:converttempResponse xmlns:
         <return>
            <TEMPOUT>99.99</TEMPOUT>
         </return>
      </ns2:converttempResponse>
   </soap:Body>
</soap:Envelope>
```

Request and SoapAction

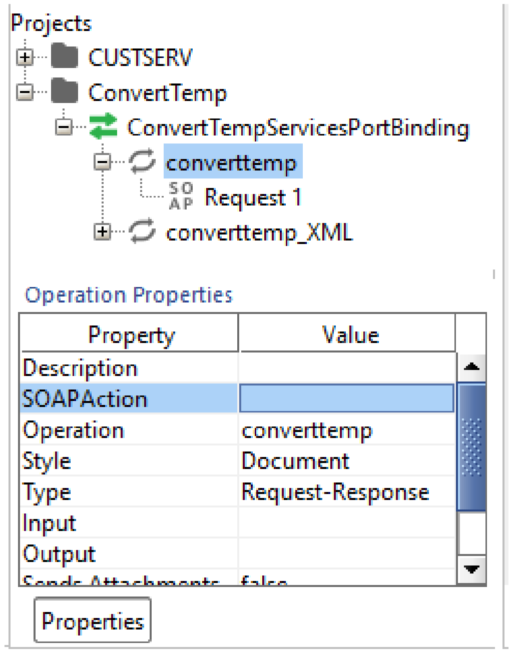


Request 1

http://power8:10076/web/services/ConvertTempService/ConvertTemp

```
<soapenv:Envelope xmlns:soapenv="
   <soapenv:Header/>
   <soapenv:Body>
      <con:converttemp>
         <arg0>
            <TEMPIN>212</TEMPIN>
         </arg0>
      </con:converttemp>
   </soapenv:Body>
</soapenv:Envelope>
```

```
<soap:Envelope xmlns:soap="http://sch
   <soap:Body>
      <ns2:converttempResponse xmlns:
         <return>
            <TEMPOUT>99.99</TEMPOUT>
         </return>
      </ns2:converttempResponse>
   </soap:Body>
</soap:Envelope>
```



Projects
- CUSTSERV
- ConvertTemp
  - ConvertTempServicesPortBinding
    - converttemp
      - Request 1
    - converttemp_XML

Operation Properties

Property	Value
Description	
SOAPAction	
Operation	converttemp
Style	Document
Type	Request-Response
Input	
Output	
Sends Attachments	false

Properties

In the request window, you could see the SOAP message (XML documents with parameters) as well as the URL.

You can try the request by clicking the green triangle ("play button")

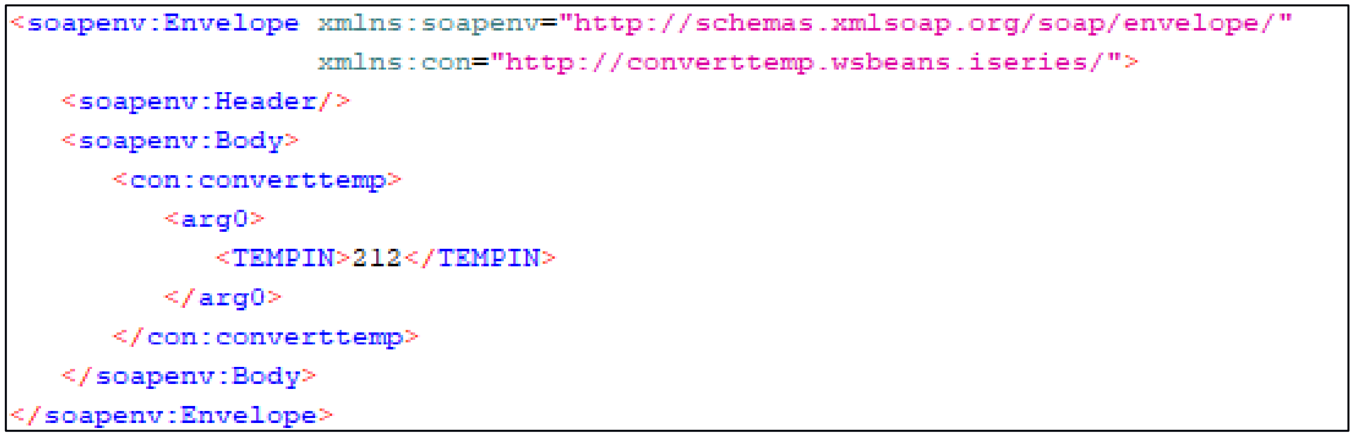
The other thing you'll eventually need is the SoapAction. You can find it under the properties for the operation ("converttemp" in this case)

The ConvertTemp example wants a blank SoapAction

Sample SOAP Documents

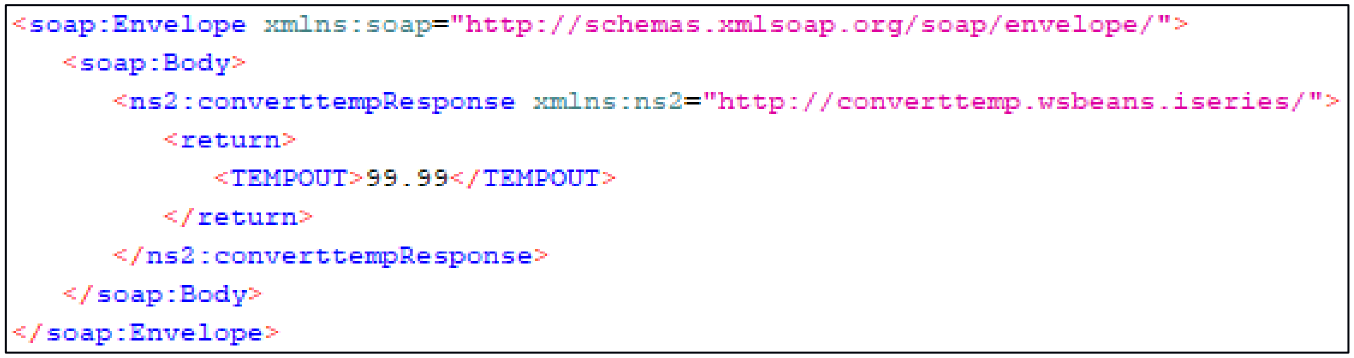
Here are example SOAP messages that I discovered by running the WSDL through SoapUI. Now that I know what these look like, I can do the same thing from RPG…

Input Message



```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:con="http://converttemp.wsbeans.iseries/">
   <soapenv:Header/>
   <soapenv:Body>
      <con:converttemp>
         <arg0>
            <TEMPIN>212</TEMPIN>
         </arg0>
      </con:converttemp>
   </soapenv:Body>
</soapenv:Envelope>
```

Output Message



```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Body>
      <ns2:converttempResponse xmlns:ns2="http://converttemp.wsbeans.iseries/">
         <return>
            <TEMPOUT>99.99</TEMPOUT>
         </return>
      </ns2:converttempResponse>
   </soap:Body>
</soap:Envelope>
```

Running SOAP from RPG (1/2)

```
     H DFTACTGRP(*NO) BNDDIR('HTTPAPI')
     D TEMPCONV        PR
     D   fahrenheit                  15p 5 const
     D TEMPCONV        PI
     D   fahrenheit                  15p 5 const

      /copy httpapi_h

     D URL             s            100a   varying
     D SOAP            s           1000A   varying
     D response        s           1000a   varying
     D TEMPOUT         s              7p 2

      /free

       http_debug(*ON);

       URL = 'http://power8:10076/web/services/ConvertTempService/ConvertTemp';

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

Running Soap from RPG (2/2)

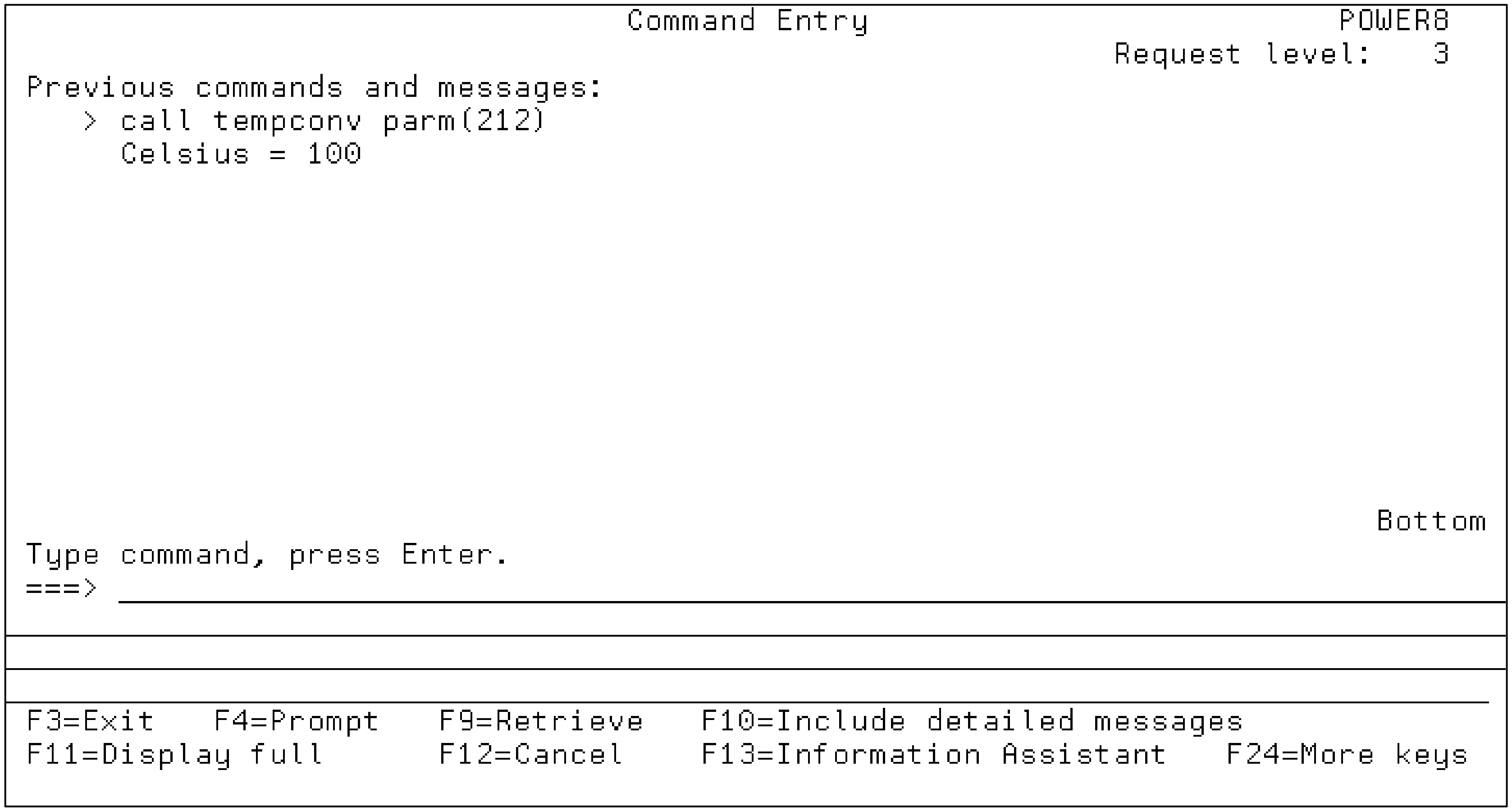
```
        SOAP = '+
          <soapenv:Envelope +
              xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" +
              xmlns:con="http://converttemp.wsbeans.iseries/">+
              <soapenv:Header/>+
              <soapenv:Body>+
              <con:converttemp>+
              <arg0>+
                <TEMPIN>' + %char(%inth(fahrenheit)) + '</TEMPIN>+
              </arg0>+
              </con:converttemp>+
          </soapenv:Body>+
         </soapenv:Envelope>';

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

        xml-into TEMPOUT %xml(response: 'case=convert ns=remove +
            path=Envelope/Body/converttempResponse+
                 /return/TEMPOUT');

        http_comp( 'Celsius = ' + %char(%inth(TEMPOUT)));

        *inlr = *on;

       /end-free
```

The ns=remove option strips the namespaces.

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

http_comp sends a "completion" message

Soap from RPG Output



```
                                 Command Entry                        POWER8
                                                         Request level:    3
Previous commands and messages:
   > call tempconv parm(212)
     Celsius = 100















                                                                       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
```

WSDL2RPG

For SOAP web services, 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/myUserId/CurrencyConvertor.wsdl')
         SRCFILE(MYLIB/QRPGLESRC)
         SRCMBR(CURRCONV)
```

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

- The RPG it generates often needs to be tweaked before it'll compile.
  - Usually string lengths
- 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 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

Scott's IBM I port of YAJL:

http://www.scottklement.com/yajl/

The original YAJL site (not IBM i oriented)

https://github.com/lloyd/yajl

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!