Providing RPG Web Services

on IBM i

Presented by

Scott Klement

http://www.scottklement.com

© 2012-2019, Scott Klement

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

Agenda for this session:

1. Introduction
   • What's a web service?
   • Why web services?
   • Types (REST/SOAP/XML/JSON)

2. SOAP web service with IBM's IWS

3. REST web service with IBM's IWS

4. Writing your own from the ground-up with Apache.

5. Discussion/wrap-up
I am a Web Service. What Am I?

A routine that can be called over a TCP/IP network.

• 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!
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)

...this session focuses on providing.
A “program call” (or subprocedure call) that works over the Web.

- Very similar in concept to the CALL command.
  
  ```
  CALL PGM(EXCHRATE) PARM('us' 'euro' &DOLLARS &EUROS)
  ```

- Runs over the Web, so can be called from programs on other computers anywhere in the world.

- Maybe a web front-end?
  - (Java, .NET, PHP, JavaScript framework, even another RPG.)

- Maybe a thick-client program (windows program, mobile app, etc.)
To be called from a program

Designed to be called from other programs, instead of interfacing directly with the user.

- Web services do not display a screen, or prompt a user
- All input comes from “parameter” data.
- All output is sent via “parameter” data
- Often referred to as an “API”
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.
**REST vs SOAP**

**REST:** Web service where the URL identifies a “resource” to work with.
- Input/output documents may be in any format. (Most commonly XML or JSON)
- Often, all input is within the URL
- Technically, the HTTP method should be the “verb” (type of action to take), but many web services do not use this approach, and still refer to themselves as REST
- Much simpler/runs faster than SOAP.

**SOAP:** XML-based web service where document format is standardized
- Input/output documents are always XML in SOAP format
- The “verb” (or action to perform) is given in a separate “soap-action” keyword.
- An accompanying WSDL document describes the SOAP details, including networking details and schema
- Much more complex than REST
- Many more tools are available (vs REST) which can make SOAP easier to code than REST.
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.
In a 2013 study done by the ProgrammableWeb (web service directory and community), we can see JSON growing while XML is declining.

As a percentage of the overall directory (left) XML is higher, but it's close.

For new APIs, JSON is much higher
JSON and XML to Represent a DS

**Array of data structures in RPG...**

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

[  
  {  
    "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

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

\[
<\text{list}><\text{cust}><\text{custno}>1000</\text{custno}><\text{name}>ACME, Inc</\text{name}></\text{cust}><\text{cust}><\text{custno}>2000</\text{custno}><\text{name}>Industrial Supply Limited</\text{name}></\text{cust}></\text{list}>
\]

92 bytes

142 bytes

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

50 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!
IBM's Integrated Web Services Server

We could do SOAP the same way as POX -- but we'd have to develop the WSDL file manually, and that would be difficult. Fortunately, IBM provides a Web Services tool with IBM i at no extra charge!

*The tool takes care of all of the HTTP and XML work for you!*

It's called the *Integrated Web Services* tool.

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

- Can be used to provide web services
- Can also be used to consume them -- but requires in-depth knowledge of C and pointers -- I won't cover IBM's consumer tool today.

Requirements:
- IBM i operating system, version 5.4 or newer.
- 57xx-SS1, opt 30: QShell
- 57xx-SS1, opt 33: PASE
- 57xx-JV1, opt 8: J2SE 5.0 32-bit (Java)
- 57xx-DG1 -- the HTTP server (powered by Apache)

*Make sure you have the latest cum & group PTFs installed.*
Let's Get Started!

The HTTP server administration tool runs in a special HTTP server called *ADMIN, and you use it from your browser.

- If this isn’t already started, you can start it with:
  ```
  STRTCPSVR SERVER(*HTTP) HTTPSVR(*ADMIN)
  ```
- Point browser at:
  ```
  http://your-system:2001/
  ```
- Sign-in
- Click “Internet Configurations” (if IBM i 6.1 or higher)
- Click “IBM Web Administration for i"
Click "Internet Configurations"
Internet Configurations

IBM Web Administration for i

IBM i Internet Configurations allows you to perform internet configuration tasks.

- **IBM Web Administration for i**
  Allows you to manage and configure HTTP servers and application servers.

- **Digital Certificate Manager**
  Allows you to create, distribute, and manage Digital Certificates.

- **IBM IPP Server for i**
  Allows you to configure the IBM IPP Server.

- **Internet Setup Wizard**
  Allows you to use the internet setup wizard to configure your internet environment.
Web Administration for i

The IWS is under "Create New Web Services Server"

The same link is up here as well – and is available throughout the tool from this link.
Create IWS Server (1 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.
Two servers are needed
1. One to run Java (application server)
2. One that handles the web communications (HTTP server)

A third port is used to communicate commands between them.
Port numbers must be unique system-wide.
The wizard will provide defaults that should work.
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.

The Web services server provides a convenient way to externalize existing programs running on IBM i, such as RPG and COBOL programs, as Web services. Web service clients can then interact with these IBM i program based services from the Internet or intranet using Web service based industry standard communication protocols such as SOAP. The clients can be implemented using a variety of platforms and programming languages such as C, C++, Java and .NET. An easy to use wizard is provided to configure the Web services server and the services for IBM i program objects. Other management functions such as starting, stopping and deleting services are also provided.

For more information, please visit: http://www-01.ibm.com/support/docview.wss?uid=isg3T1026868

Note: To update the status, click Refresh
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
• 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.
SOAP Web Services

- Always XML (you could have a different "payload", but it'd be embedded in XML under the covers)
- SOAP is the XML format for the "parameters" when making a call
- URL and SoapAction HTTP header define the program to call.
- WSDL document describes the details (contains network info as well as an XML schema)

To understand Web Services Description Language (WSDL), think "how would you tell the world"?

- Documentation? (Word Doc, PDF, etc?)
- Sample programs?
- Or… info that can be used to generate programs?
WSDL Skeleton

```xml
<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.)`
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 perhaps a misnomer!

- Not as simple as RPG parameter lists.
- Not as simple as REST
Here's the skeleton of a SOAP message:

```xml
<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>
```
Sample SOAP Documents

Input Message

```xml
<soapenv:Envelope>
  <soapenv:Body>
    <xsd:getcust>
      <xsd:args0>
        <xsd:CUSTNO>495</xsd:CUSTNO>
      </xsd:args0>
    </xsd:getcust>
  </soapenv:Body>
</soapenv:Envelope>
```

Output Message

```xml
<soapenv:Envelope>
  <soapenv:Body>
    <ns:getcustResponse>
      <ns:return>
      </ns:return>
    </ns:getcustResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

Some details removed for brevity....
H DFTACTGRP(*NO) ACTGRP('SOAP') PGMINFO(*PCML: *MODULE)

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

D CUST E DS qualified
defname(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.
GETCUST RPG Program (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!
Our GETCUST example gets input and output as normal parameters. To use these with IWS, we need to tell IWS what these parameters are. This is done with yet another XML document.

**PCML = Program Call Markup Language**

- A flavor of XML that describes a program's (or *SRVPGM's) parameters.

- Can be generated for you by the RPG compiler, and stored in the IFS:

```sql
CRTBNDRPG PGM(xyz) SRCFILE(QRPGLESRC) 
PGMINFO(*PCML) 
INFOSTMF('/path/to/myfile.pcml')
```

- Or can be embedded into the module/program objects themselves, with an H-spec:

```sql
H PGMINFO(*PCML:*MODULE)
```
GETCUST as SOAP Service

To add a program (such as our 'Get Customer' example) click "Deploy New Service"
We'll do SOAP first, so select SOAP from the choices here.
Remember the PGMINFO(*PCML:*MODULE)?

When the PCML is inside the module, you can just point the web service server to the ILE program or service program object.

If the PCML was saved to the IFS, however, choose the "Browse" option, and provide the IFS path name instead.
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.
It knows the parameters from the PCML. But, I need to tell it which ones are input, and which are output.
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.
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.
If you check the boxes here, IWS will set environment variables containing various additional information.

If you need this – go ahead and check the box(es).

Otherwise, just take the default.
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.
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.
Testing SOAP with SoapUI (1 of 4)

Step 1: Click SOAP on the ribbon (toolbar)

Step 2: Paste in URL to WSDL (from the "View WSDL" link) into the Initial WSDL field
Testing SOAP with SoapUI (2 of 4)

Step 3:
Expand tree til you find the 'Request 1'. Double click it to see SOAP request.

Step 4:
Enter the customer number into the SOAP message for the input parms.
Testing SOAP with SoapUI (3 of 4)

Step 5: Click the small green triangle -- SoapUI will send the request over HTTP to the IWS server!
Testing SOAP with SoapUI (4 of 4)

Step 6:
View the returned SOAP message (output parms): it worked!
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.
Example REST Input

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.
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.
As with the SOAP example, PCML will be used to learn about the program’s parameters.
resource name is 'cust', because we want /cust/ in the URL.

description can be whatever you want.

PATH template deserves its 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/
Like SOAP, we have to identify which parameters are input or output.
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.
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
Here you can change the method and the resource ("noun") easily, and click the green "play" button to try it.

It can also help make XML, JSON or HTML output "prettier" by formatting it for you.
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.
Create an HTTP Server

Click “Setup” to create a new web server.

Do not create a web services server at this time. That is for IBM’s Integrated Web Services tool, currently used only for SOAP.

Instead, create a “normal” HTTP server.
The “Server Name” controls:
- The job name of the server jobs
- The IFS directory where config is stored
- The server name you select when editing configs
- The server name you select when starting/stoping the server.
Server Root

The “server root” is the spot in the IFS where all the files for this server should go. By convention, it’s always /www/ + server name.
The “document root” is the default location of files, programs, images, etc. Anything in here is accessible over a network from your HTTP server. By convention, it’s always specified as /www/ + server name + /htdocs.
Set Port Number

This is where you specify the port number that we determined on the “Manage / All Servers” screen.
An “access log” will log all accesses made to the HTTP server. Useful to track server activity.
Access Log Retension

Over time, access logs can get quite large. The HTTP server can automatically delete data over a certain age. I like to keep mine for about a week.
Summary Screen

This screen summarizes the settings you provided. When you click “Finish”, it will create the server instance.
URL Tells Apache What to Call

To get started with REST, let's tell Apache how to call our program.

```xml
ScriptAlias /cust /qsys.lib/restful.lib/custinfo.pgm
<Directory /qsys.lib/restful.lib>
    Require all granted
</Directory>
```

- Just add the preceding code to an already working Apache instance on IBM i.
- **ScriptAlias** tells apache that you want to run a program.
- If URL starts with `/cust`, Apache will **CALL PGM(RESTFUL/CUSTINFO)**
- Our REST web service can be run from any IP address (Allow from all).

http://ibmi.example.com/cust/495

- Browser connects to: *ibmi.example.com*
- Apache sees the /cust and calls RESTFUL/CUSTINFO
- Our program can read the 495 (customer number) from the URL itself.
Apache 2.4 Update

Starting with IBM i 7.2, we have Apache 2.4. They recommend using "require" instead of "Order"

Newer syntax:

```html
<Directory /qsys.lib/restful.lib>
    Require all granted
</Directory>
```

Older syntax:

```html
<Directory /qsys.lib/restful.lib>
    Order allow,deny
    Allow from all
</Directory>
```

If you are using an older release, use this second syntax.
Scroll down to the “Tools” section.

Use “edit configuration file” to enter Apache directives.

Tip: You can use “Display configuration file” to check for errors in the Apache configuration.
Alternate Recipe

The last slide shows how to make /cust always do a call restful/custinfo.

But, perhaps you’d rather not have to key a separate Apache configuration for each restful web service you want to run? There are pros and cons to this:

- Don’t have to stop/start server to add new service.
- Any program left in RESTFUL library can be run from outside. If the wrong program gets compiled into this library, it could be a security hole.

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

<Directory /qsys.lib/restful.lib>
  Require all granted
</Directory>
```

http://ibmi.example.com/rest/custinfo/495
Add Custom Directives

Scroll down to the bottom of the file. Type the directives (as shown) and click “Apply” to save your changes.
Start New Apache Server

Before starting, click “Display Configuration File” and make sure it does not show any errors.

Then, click the green “start” button at the top to start your new server.

You can also start from 5250 with:

\texttt{STRTCPSVR *HTTP HTTPSVR(MYDEMO)}
**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://i.scottklement.com:8001/cust/495
-or-
GET http://i.scottklement.com:8001/cust/495?op=retrieve

**Output**

```xml
<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 \(\rightarrow\) reads standard input
- getenv \(\rightarrow\) retrieves an environment variable.
- QtmhWrStout \(\leftarrow\) writes data to standard output.
## DIY REST Example (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(*vsize)
D DtaVarLen 10I0 const
D ErrorCode 8000A options(*vsize)

D err ds qualified
D bytesProv 10i0 inz(0)
D bytesAvail 10i0 inz(0)

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

D CRLF C x'0d25'
D pos s 10i0
D uri s 5000a varying
D data s 5000a
```
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.
DIY REST Example (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);
```

Status 200 means that all was well.

Here I send the XML Response.

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.)
Changes To Use W/Alt Recipe

To use the alternate Apache config (ScriptAliasMatch) change this code:

```plaintext
monitor;
    pos = %scan('/cust/': uri) + %len('/cust/');
    custno = %int(%subst(uri:pos));
on-error;
```

To this… it now works on anything after /cust/ or /custinfo/ in the URI

```plaintext
D ID1       c       '/cust/'
D ID2       c       '/custinfo/'
...
monitor;
    select;
    when %scan(ID1: uri) > 0;
        pos = %scan(ID1: uri) + %len(ID1);
    when %scan(ID2: uri) > 0;
        pos = %scan(ID2: uri) + %len(ID2);
    other;
        pos = 0;
    endl;
    custno = %int(%subst(uri:pos));
on-error;
```
Testing and More Examples

• There's nothing special about testing our DIY example. You call it the same as any other REST web service – just use SoapUI or the browser, just as we did with the IWS example.

• There are additional examples of REST and SOAP in the handout. These are for your benefit – due to time concerns, Scott will skip over these in a standard 75 minute presentation.
There's nothing special about testing a DIY example. You call it the same as any other (REST) web service – just use SoapUI or the browser, just as we did with the IWS example.
Although the previous slide had only one parameter, REST can have multiple parameters -- but they must all fit on the same URL.

http://i.scottklement.com:8001/rest/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 will scan for the slashes, get the parameter info from the URL, and build a JSON document that matches the criteria.
Our JSON Web Service Example

For our next example, we’ll create a report of all invoices for a customer.

http://i.scottklement.com:8001/rest/invoice/495/20100901/20100930

```
{  
  "success": false,  
  "errmsg": "Put Error Message Here"
}
```

```
{  
  "success": true,  
  "errmsg": 
  "list": [{  
    "invno": "xyz",  
    "date": "2012-01-23",  
    "name": "Acme Industries, Inc.",  
    "amount": 123.45,  
    "weight": 123.45,  
  },  
  { same fields again },  
  { same fields again },  
  { etc }
  ]
}
```

If an error occurs, we’ll output a JSON document like this.

If there’s no error, we’ll output data in JSON format, as a big array of data structures. There’s no limit to how many rows of data you can place in a JSON document.
## DIY JSON, RPG Code (1 of 5)

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D CRLF</td>
<td>C</td>
<td>x'0d25'</td>
</tr>
<tr>
<td>D data</td>
<td>s</td>
<td>5000a varying</td>
</tr>
<tr>
<td>D uri</td>
<td>s</td>
<td>5000a varying</td>
</tr>
<tr>
<td>D cust</td>
<td>s</td>
<td>4s 0</td>
</tr>
<tr>
<td>D sdate</td>
<td>s</td>
<td>8s 0</td>
</tr>
<tr>
<td>D edate</td>
<td>s</td>
<td>8s 0</td>
</tr>
<tr>
<td>d custpos</td>
<td>s</td>
<td>10i 0</td>
</tr>
<tr>
<td>d sdatepos</td>
<td>s</td>
<td>10i 0</td>
</tr>
<tr>
<td>d edatepos</td>
<td>s</td>
<td>10i 0</td>
</tr>
<tr>
<td>D jsonName</td>
<td>s</td>
<td>25a</td>
</tr>
<tr>
<td>D jsonDate</td>
<td>s</td>
<td>10a</td>
</tr>
</tbody>
</table>

* Unicode versions of {, }, [ and ], respectively.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D LBRACE</td>
<td>C</td>
<td>u'007b'</td>
</tr>
<tr>
<td>D RBRACE</td>
<td>C</td>
<td>u'007d'</td>
</tr>
<tr>
<td>D RSQB</td>
<td>C</td>
<td>u'005d'</td>
</tr>
<tr>
<td>D LSQB</td>
<td>C</td>
<td>u'005b'</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D row</td>
<td>ds</td>
<td>qualified</td>
</tr>
<tr>
<td>D inv</td>
<td>5a</td>
<td></td>
</tr>
<tr>
<td>D date</td>
<td>8s 0</td>
<td></td>
</tr>
<tr>
<td>D name</td>
<td>25a</td>
<td></td>
</tr>
<tr>
<td>D amount</td>
<td>9p 2</td>
<td></td>
</tr>
<tr>
<td>D weight</td>
<td>9p 1</td>
<td></td>
</tr>
</tbody>
</table>
/free
  exec SQL set option naming=*SYS;

  *inlr = *on;
  uri = %str(getenv('REQUEST_URI'));

  monitor;
  custpos = %scan('/invoice/': uri) + %len('/invoice/');
  sdatepos = %scan('/': uri: custpos) + 1;
  edatepos = %scan('/': uri: sdatepos) + 1;
  cust  = %int(%subst(uri: custpos: (sdatepos-custpos-1)));
  sdate = %int(%subst(uri: sdatepos: (edatepos-sdatepos-1)));
  edate = %int(%subst(uri: edatepos));

  on-error;
  data = 'Status: 500 Invalid URI' + CRLF
         + 'Content-type: text/json' + CRLF
         + CRLF
         + %char(LBRACE) + CRLF
         + '"success": false,' + CRLF
         + '"errmsg": "An unknown URI format was given"' + CRLF
         + %char(RBRACE) + CRLF;
  QtmhWrStout(data: %len(data): err);
  return;
endmon;
exec SQL declare C1 cursor for
   select aiOrdn, aiIDat, aiSNme, aiDamt, aiLbs
   from ARSHIST
   where aiCust = :cust
       and aiIDat between :sdate
       and :edate;

exec SQL open C1;
exec SQL fetch next from C1 into :row;

if sqlstt<>'00000'
   and %subst(sqlstt:1:2) <> '01'
   and %subst(sqlstt:1:2) <> '02';
data = 'Status: 500 Query Failed' + CRLF
    + 'Content-type: text/json' + CRLF
    + CRLF
    + %char(LBRACE) + CRLF
    + ""success": false,"
    + CRLF
    + ""errmsg": "SQL Failed with SQLSTT='+SQLSTT+'"" + CRLF
    + %char(RBRACE) + CRLF;
QtmhWrStout(data: %len(data): err); return;
endif;
• Each time I call QtmhWrStout(), it adds more data on to the end of what I’ve already sent.
• This part is just the start of the JSON document.
• Subsequent calls will write rows of data, and they will be added on to the end.
• Finally, we’ll call QtmhWrStout one last time to end the JSON document.
dow %subst(sqlstt:1:2)='00' or %subst(sqlstt:1:2)='01';
jsonName = %scanrpl( '"': '"': row.name );
jsonDate = %char( %date( row.date: *iso ): *iso );
data = %char(LBRACE) + CRLF
  + ' "invno": "' + row.inv + '",' + CRLF
  + ' "date": "' + jsonDate + '",' + CRLF
  + ' "name": "' + %trim(jsonName) + '",' + CRLF
  + ' "amount": "' + %char(row.amount) + '",' + CRLF
  + ' "weight": "' + %char(row.weight) + '"' + CRLF
+ %char(RBRACE);
QtmhWrStout(data: %len(data): err);

exec SQL fetch next from C1 into :row;
if %subst(sqlstt:1:2)='00' or %subst(sqlstt:1:2)='01';
data = ',' + CRLF;
else;
data = CRLF;
endif;
QtmhWrStout(data: %len(data): err);
enddo;
data = %char(RSQB) + %char(RBRACE) + CRLF;
QtmhWrStout(data: %len(data): err);
You can test this one with SoapUI's testing tool, too.
A SOAP Service With a List

The GETCUST service only returns one "record" so to speak. Can I do something like the "Invoice List" (the DIY example) using SOAP?

• Q: How do I do that if I don't code the XML in the program?
  • A: With an array!

• Q: How do make an array that returns a list of "records" (more than one field per array element)?
  • A: Use an array of data structures.

• Q: What if the number of returned elements (i.e. the number of invoices in the list) varies? How can I specify the number of returned array elements?
  • A: If you code a "10i 0" parameter in your parameter list, IWS will let you use it to control the array size.
H OPTION(*SRCSTMT:  *NODEBUGIO) PGMINFO(*PCML:*MODULE)

<table>
<thead>
<tr>
<th>D row</th>
<th>ds</th>
<th>qualified inz</th>
</tr>
</thead>
<tbody>
<tr>
<td>D inv</td>
<td>5a</td>
<td></td>
</tr>
<tr>
<td>D date</td>
<td>8s 0</td>
<td></td>
</tr>
<tr>
<td>D name</td>
<td>25a</td>
<td></td>
</tr>
<tr>
<td>D amount</td>
<td>9p 2</td>
<td></td>
</tr>
<tr>
<td>D weight</td>
<td>9p 1</td>
<td></td>
</tr>
</tbody>
</table>

D SOAPINV     PR          ExtPgm('SOAPINV')
D CustNo      4p 0 const  |
D strDate     8p 0 const  |
D endDate     8p 0 const  |
D rtnCount    10i 0       |
D rtnList     likeds(row) dim(999) |

D SOAPINV     PI          |
D CustNo      4p 0 const  |
D strDate     8p 0 const  |
D endDate     8p 0 const  |
D rtnCount    10i 0       |
D rtnList     likeds(row) dim(999) |

rtnCount will tell IWS how many invoices are returned. (to a 999 maximum)

rtnList is the returned array. Notice: LIKEDS!

This is what needs to be returned for each invoice in the list
rtncount = 0;

exec SQL declare C1 cursor for
    select aiOrdn, aiIDat, aiSNme, aiDamt, aILbs
    from ARSHIST
    where aiCust = :CustNo
        and aiIDat between :strDate
            and :endDate;

exec SQL open C1;
exec SQL fetch next from C1 into :row;

dow sqlstt='00000' or %subst(sqlstt:1:2)='01';
    rtncount = rtncount + 1;
    rtnList(rtncount) = row;
exec SQL fetch next from C1 into :row;
endo;

exec SQL close C1;

CustNo, strDate and endDate are all input parameters passed by IWS.

For each record found, rtncount is updated, and rtnList() array contains a row.
Deploy new service adds another web service to the existing server.

The other screens will be the same as they were for GETCUST.

Except, that on the parameter screen, I have to tell IWS about the returned parameter count.
By default, the count for RTNLIST is 999, just like the DIM(999) in my RPG code.

But I can change it to "RTNCOUNT" because RTNCOUNT happens to be a 10i 0 field, IWS knows it can be used to specify the array size.

Unfortunately, there's no way to stop IWS from sending RTNCOUNT to the consumer, as well. (But if the consumer doesn't need it, it can ignore it.)
Discussion / Wrap Up

My thoughts:

• If your service is simple enough, and the performance is good enough, use the Integrated Web Services (IWS) tool.

• If you plan to have many consumers, choices are good!
  o You can do both REST and SOAP
  o REST version can provide both XML and JSON

• If only used internally, no need for WSDL, SOAP or multiple choices. Just make a simple REST Service.

• JSON works much better from web/ajax callers.

• Only need XML if you think your callers will want/need that format. Otherwise, use JSON.
This Presentation

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

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

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

Thank you!