REST APIs and RPG

Presented by

Scott Klement

http://www.scottklement.com

Fun Fact: If you took everything posted on twitter (X) every day and put it into a book, that book would be 10 million pages long.

The Agenda

1. REST API Concepts
   - What is an API?
   - What makes an API RESTful?
   - Terminology
   - URLs, methods, status codes
   - XML and JSON messages
2. Consuming APIs
   - Working with a testing tool
   - What is needed to consume from RPG?
   - Samples of the different methods
   - A more complex/complete example
3. Providing APIs
   - Introduction to the Integrated Webservices (IWS) tool
   - Creating an IWS server
   - IWS Example
   - Introduction to Do It Yourself (DIY)
   - Creating an Apache server
   - DIY Example

REST API Concepts

What is an API?

API = Application Programming Interface

Technically, any sort of routine (program, subprocedure, SQL function, web service, etc.) that's designed to be called from another program is an API.

- A program that you call from other programs
- Example: Program that calculates sales tax, called from several other programs when they need to have tax calculated.
- We have all written APIs! IBM provides many with the OS!

However, in recent years, the term "API" has become short for "REST API", which is a type of web service.

What is a Web Service?

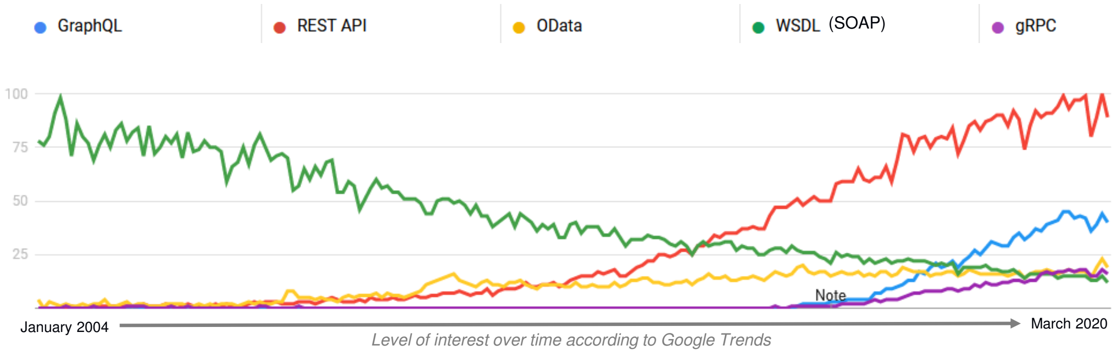
A Web Service is an API designed to be called over a network

- Typically using the HTTP (or HTTPS) network protocol.
- Not to be confused with a web page or web site!
  - ○ No HTML, CSS or JavaScript, here!
  - ○ You don't use a web browser!

Useful for:

- Interconnecting applications across systems
- Web page to back-end server (system of record)
- Cloud application to/from traditional on-premises system
- Communication between businesses (EDI-like, B2B e-Commerce)
- Between different packages.

Types of Web Services



Level of interest over time according to Google Trends

My observations:

- *REST is easily the most popular*
- *GraphQL may be up-and-coming*
- *WSDL (SOAP) was the most popular but has nearly died out*

Lets Take An Example

We want to translate text from English to Spanish.

IBM Watson offers language translation on IBM Cloud!

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
```

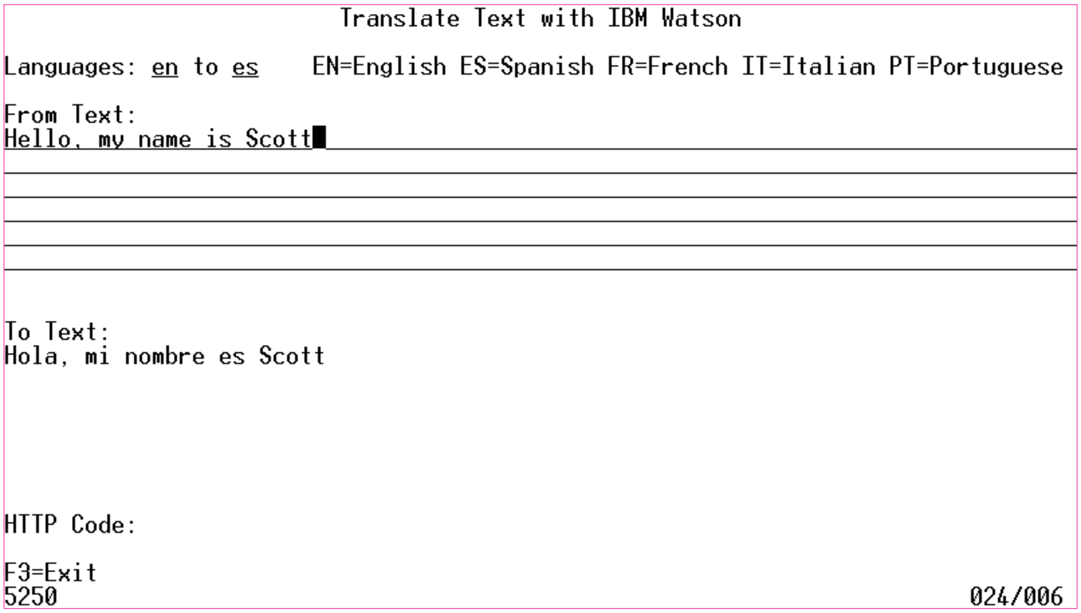
Output parameter:

Translated text: `'Hola'`

You can think of it like this:

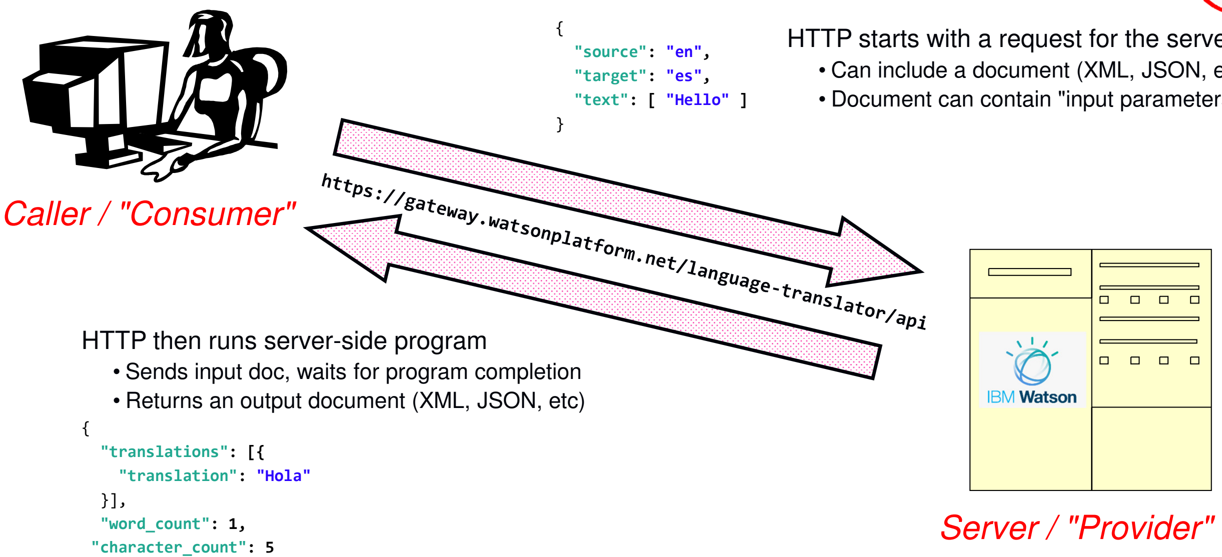
```
CALL PGM(TRANSLATE) PARM('en-es' 'Hello' &RESULT)
```

An Example RPG Screen



```
                        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
```

Overview Of An API Call



```
{
  "source": "en",
  "target": "es",
  "text": [ "Hello" ]
}
```

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

- Sends input doc, waits for program completion
- Returns an output document (XML, JSON, etc)

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

What is the REST Architecture?

REST = REpresentational State Transfer

The concept comes from the doctoral dissertation of Roy Fielding, UC-Irvine
https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

The REST architectural style describes six constraints:

1. Uniform Interface
2. Stateless
3. Cacheable
4. Client-Server
5. Layered System
6. Optionally, Code-on-Demand

Resource Based

- Things vs. Actions
- Nouns vs Verbs
- vs. SOAP or RPC which are action based
- Resources are identified by URIs
  - Possible for multiple URIs to refer to the same resource
- Separate from their representations
  - Different shapes of data, or representations, can still represent the same resource.

Representational

- How things are manipulated
- Part of the state of the resource
- Typically represented as JSON or XML (but other forms, such as CSV are valid)
- Example:
  - Resource: person (Scott)
  - Service: contact information (GET)
  - Representation: name, address, phone, e-mail, etc
  - JSON or XML format

Uniform Interface

- Defines interface between client/server
- Simplifies and decouples the architecture
- Fundamental to RESTful design
- For us this means:
  - HTTP verbs (GET, PUT, POST, DELETE)
  - URIs
  - HTTP Response (status, body)

Stateless

- Server contains no client state
- Each request contains enough context to process the message.
  - Self-descriptive messages
- Any session state is held on the client
- Though, sometimes APIs are only REST-like
- No using QTEMP!!

Client / Server

- Assume a disconnected system
- Separation of concerns
- Uniform interface is the link between the two

Cacheable

- Server responses (representations) are cacheable
  - Implicitly
  - Explicitly
  - Negotiated
- For example, server may decide to cache answer rather than re-read database
- Or It may say how old the item is
- Or the client may request a cached vs non-cached item

Layered System

- Client can't assume direct connection to server (could be cached or handled by an intermediary)
- Software or hardware intermediaries between client/server
- Improves scalability

Code On Demand (optional)

- Server can temporarily extend client
- Transfer logic to client
- Client executes logic
  - Flash
  - Java applets
  - JavaScript
- This constraint is optional
- Not normally used with APIs

REST Architecture Summary

- Violating any means you aren't (strictly speaking) RESTful
  - Example: Three-legged OAUTH2

- Compliance with REST constraints allows:
  - Scalability
  - Simplicity
  - Modifiability
  - Visibility
  - Portability
  - Reliability

This architectural information "borrowed" from:

https://www.restapitutorial.com/

Uniform Resource Identifier

- Works Over HTTP
- Specifies which computer/server/device to connect to
- Specifies the resource within that device
- … so whole URI (http://example.com) represents the "resource"
  - The thing you are working with
  - A "customer" or a "product", etc.
  - Unique ID -- like a key
  - Best when hierarchical… consider this conceptually:

```
http://example.com/apis                              (all apis)
http://example.com/apis/customers                    (all customers)
http://example.com/apis/customers/1234               (one customer, etc.)
http://example.com/apis/customers/1234/orders
http://example.com/apis/customers/1234/orders/5321
http://example.com/apis/customers/1234/orders/5321/lineItems
http://example.com/apis/customers/1234/orders/5321/lineItems/1
```

Terminology: URL vs URI

- URI = Uniform Resource Identifier
  - The more general of the two terms
  - Think of it like a "data structure"
    - "scheme" (http://) -- identifies a specific type of URI, in this case HTTP
    - "node" (example.com) -- identifies the address within the network
    - "path" (/apis/customers/1234) -- identifies the resource within the node
  - Together, these parts identify something specific
  - This is the "noun" in the REST architecture

- URL = Uniform Resource Locator
  - More commonly heard
  - A specific type of URI
  - Identifies how to "locate" or get to something
  - Such as a directory on a hard drive

HTTP Methods

If the URI specifies the "noun" (the thing/resource you're working with) what specifies the verb?

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

The action that's taken on the resource ("the verb") is determined by the HTTP method. There are four common HTTP methods:

- 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 is a term that tends to confuse people. (Not exactly a word you use every day!)
It means you can do it multiple times but have the same result.

Idempotence

Idempotence (UK: /ˌɪdɛmˈpoʊtəns/, US: /ˌaɪdəm-/) is the property of certain operations in mathematics and computer science whereby they can be applied multiple times without changing the result beyond the initial application. The concept of idempotence arises in a number of places in abstract algebra (in particular, in the theory of projectors and closure operators) and functional programming (in which it is connected to the property of referential transparency).

Wait a minute!

- Suppose you have a cow, but you want more
- You hire a breeding/siring service
- Now you want still more…
- ... can a cow get "more pregnant"?

Idempotent vs. Non-idempotent

Non-idempotent	Idempotent
Charging a credit card	Counting your money
Creating an invoice	Storing a customer's address
Writing/inserting a record	Updating a record
Adding 10 to a number	Setting a number to 10

If you do the same thing multiple times, and the resulting state is the same, it is idempotent

If you do things multiple times, and each time it alters the state, it is non-idempotent.

REST/CRUD analogy

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

```
http://my-server/apis/customers/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.

Messages / Representations

If a URI *identifies* a resource, then a message is the current *representation* of that resource.

For example, we can get the representation of a customer, or set the representation of a new customer

```
GET http://my-server/apis/customers/495
```

```
{
  "custno": 495,
  "name": "Acme Foods",
  "address": {
    "street": "1100 NW 33rd Street",
    "city": "Pompano Beach",
    "state": "FL",
    "postal": "33064-2121"
  }
}
```

```
POST http://my-server/apis/customers
```

```
{
  "custno": 1234,
  "name": "Scott Klement",
  "address": {
    "street": "8825 S Howell Ave",
    "city": "Oak Creek",
    "state": "WI",
    "postal": "53154"
  }
}
```

Messages as Parameters

Another way to think of it is to think of the messages as a set of parameters passed to a routine

```
POST https://gateway.watsonplatform.net/language-translator/api
```

Input message ("input parameters")

```
{
  "source": "en",
  "target": "es",
  "text": [ "Hello" ]
}
```

Output message ("output parameters")

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

A purist might argue that this isn't *truly* "REST"

- URL doesn't really identify a resource, but a routine to call.
- Messages don't represent the resource

However, this RPC style of "REST-like" interface is extremely commonplace and popular. It is a convenient way to think about things.

Data Formats of Messages (XML and JSON)

REST allows messages in any data format, but XML and JSON are the most popular

Both XML and JSON are widely used in web services / APIs:

- 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 web services

JSON:

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

JSON and XML Messages That Represent Data

```
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

In JSON:

- [ ] characters start/end an array
- { } characters start/end an "object" (data structure)
- Within an object fieldname: value
- Commas separate elements

```
<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

In XML:

- <name></name> represents an element
- <name> is the starting tag
- </name> is the ending tag
- They can be nested or repeated to represent structures or arrays

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!
…and the syntax is simpler!
…and JSON has become more popular (MUCH) than XML in recent years

HTTP Status Codes

- URI identifies the resource we are working with and how to get to it
- HTTP method identifies what operation to perform on the resource
- How do we describe whether the operation succeeded?
- …with http status codes! Here are some examples:

Status	Meaning
200	Success (general)
201	Success (something created)
401	Unauthorized; you need to send credentials (such as user/password)
403	Forbidden; you sent valid credentials, but aren't authorized to this operation
404	Not found; the resource doesn't exist
405	Method not allowed; not due to authority -- we never allow this method.
500	Error found on server ("catch all" for any unknown error)

Find more here: https://www.restapitutorial.com/httpstatuscodes.html

REST API Concept Summary

- What an API is
- What REST is
- The REST architecture -- the constraints to being "truly" REST
- URIs vs URLs
- Importance of the URI as the "noun" or "resource"
- HTTP methods as the "verb" or "action"
- Idempotence
- Messages as representations of your data
- Using a Remote Procedure Call (RPC) REST-like architecture
- Messages as representations of parameters
- XML and JSON, the most common formats for messages
- HTTP status codes… did it succeed or fail, and why?

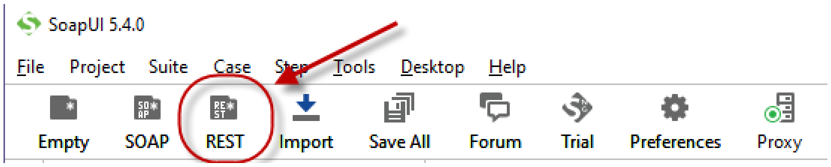
Consuming REST APIs from RPG

How Can We Try Consuming?

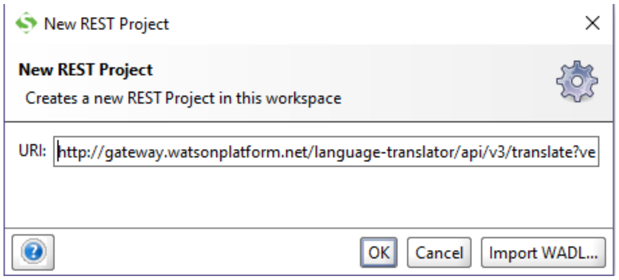
- APIs are meant for program-to-program communication
- Normally, to use them, you must write a program!
- A web service testing tool allows testing without writing a program.
  - Postman http://www.getpostman.com (REST GUI)
  - SoapUI http://www.soapui.com (SOAP/REST GUI)
  - CURL https://curl.haxx.se/ (command-line driven)

You wouldn't use a testing tool in a production scenario, but they're very useful for making sure the API works

Setting It Up in SoapUI



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



New REST Project

Creates a new REST Project in this workspace

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

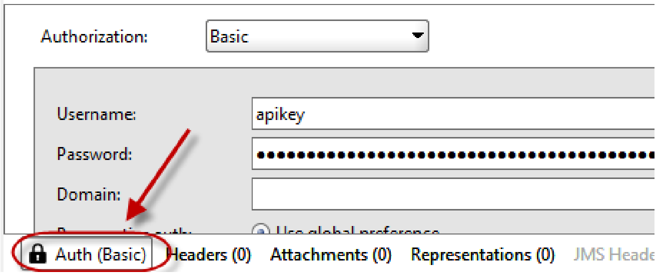
Note: This URL is too long to appear on the screen, but the box scrolls left/right to fit it all.

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

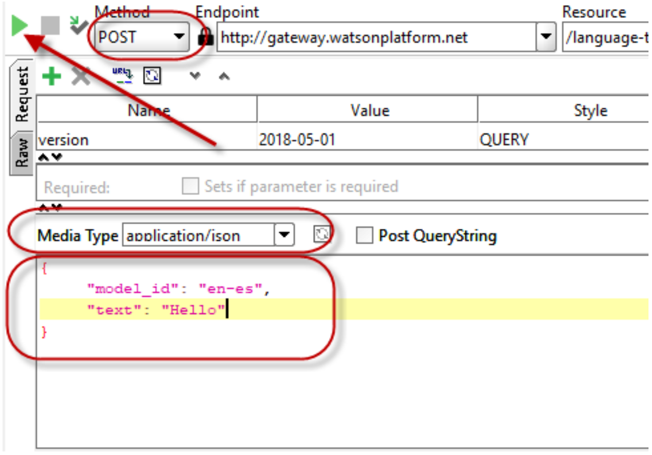
Username: apikey

Password: ••

Domain:

○ Use global preference

Trying It Out in SoapUI



POST http://gateway.watsonplatform.net /language-t

Name	Value	Style
version	2018-05-01	QUERY

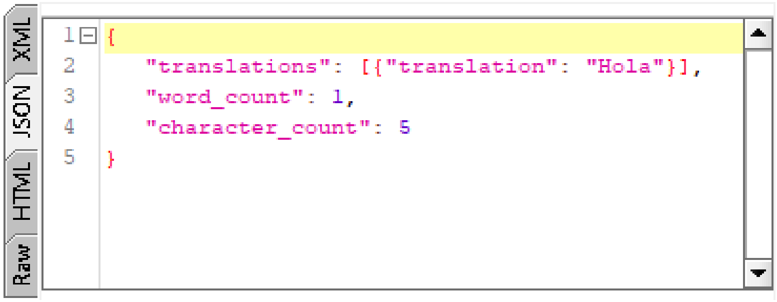
Required: □ Sets if parameter is required

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



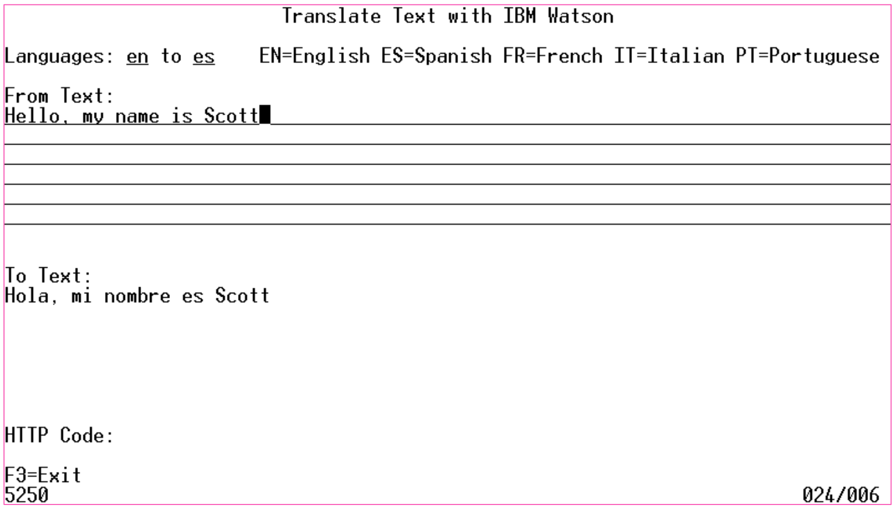
```
{
    "translations": [{"translation": "Hola"}],
    "word_count": 1,
    "character_count": 5
}
```

- 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 Might This Look Like from RPG?

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
```

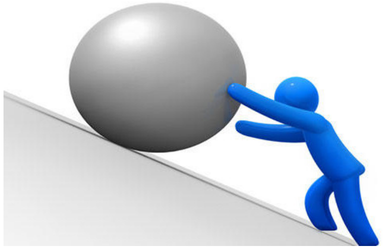
Challenges To Overcome

What problems would we need to solve to do it from RPG?

- Tool to create a JSON (or XML) document
- Some way to do http:// URL from RPG
- Tool to read the returned JSON (or XML)

Other things we might need?

- Ability to specify different methods (GET, POST, PUT, DELETE, etc)
- Somehow specify login credentials
- Transport Layer Security (TLS, often called by the older name “SSL”)
- Sometimes custom HTTP headers are needed
- Encodings sometimes needed (depending on the API)
    - XML and JSON, of course
    - URL-encoding (aka “web page forms”)
    - Multipart documents (aka “attachments”)
    - Base64 encoding



Free Options Available

Free Options Available for RPG

- Open Source HTTPAPI
- IBM-supplied SQL routines
- IBM-supplied AXIS routines

Other Languages

- Java, PHP, Ruby, Python, Node.js all provide options, here.

Commercial Options

- Various vendors provide tools. (example: Midrange Dynamics MDRest4i)
- I'm not familiar with all of the options available

HTTPAPI

Open Source (completely free tool)

- Created by Scott Klement, originally in 2001
- Written in native RPG code
- http://www.scottklement.com/httpapi

Provides Routines For

- HTTP and HTTPS (TLS/SSL) communications
- URL (web form) encoding
- Multipart (attachment) encoding
- Basic, Digest and NTLM2 authentication

Usually Used With Other Open Source Tools

- Expat for reading XML (or use XML-INTO) http://scottklement.com/expat
- YAJL for reading/writing JSON (works with DATA-INTO) http://scottklement.com/yajl
- BASE64 tool http://scottklement.com/base64

open source
initiative®
Approved License

http_string syntax

Making HTTP Requests

- http_req = general-purpose HTTP request, lots of options
- http_stmf = simplified HTTP request, where data is read from/written to IFS files
- http_string = simplified HTTP request where data is read/written from/to RPG strings

```
data-received = http_string( method : url : string-to-send : content-type )
```

- method = HTTP method (GET, POST, PUT, DELETE, etc)
- url = The URL to communicate with
- string-to-send = RPG char/varchar string to send to URL
- content-type = Internet media type (MIME type) of data you're sending
- data-received = RPG char/varchar string to contain data returned from server

Other Routines

- http_setAuth = set authentication (user/password)
- http_setOption = set various options
- http_error = retrieve error code, message, and http status code

Language Translation in RPG

```
http_setAuth( HTTP_AUTH_BASIC: 'apikey': '{your-api-key}');

request = '{"source":"en","target":"es","text":["Hello"]}';

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

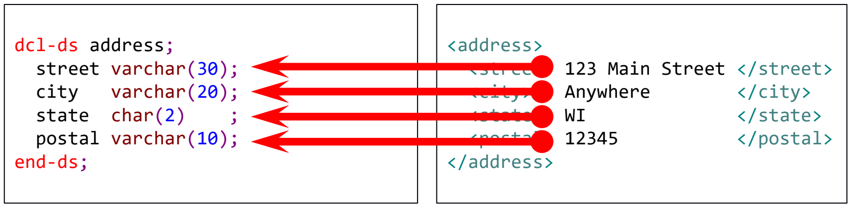
DATA-INTO – RPG opcode for parsing documents such as JSON

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

XML-INTO Concept

If parameters are passed in XML format, we can interpret it with XML-INTO. This opcode has been a part of RPG since V5R4.

Try thinking of your XML document as a "representation". Then consider the RPG representation of the same data.



```
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>
```

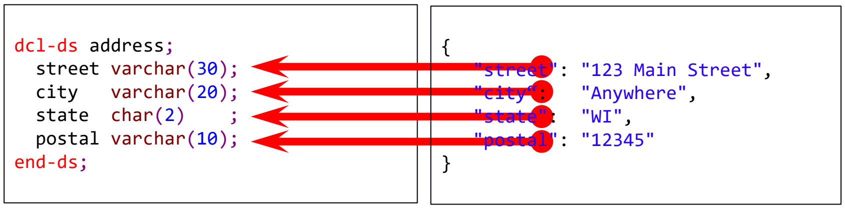
That’s what XML-INTO does!

- 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, etc.

The DATA-INTO Concept

DATA-INTO:

- Like XML-INTO, but requires a 3rd-party "parser"
- Parser determines the format of the data it understands
- Think of it like a printer driver in Windows.
- YAJLINTO is an open source (free) parser for JSON documents.



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

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

With YAJLINTO

- DATA-INTO can be used on JSON just as XML-INTO is on XML
- Very easy to read JSON documents in RPG

DATA-GEN Concept

DATA-GEN:

- DATA-INTO, but in reverse (creates document vs reads document)
- 3rd-party "generator" determines the document type
- YAJLDTAGEN is a free tool for generating JSON
- Remember, { } means "object" -- which is equivalent to an RPG data structure

```
dcl-ds address qualified;
  name   varchar(30)  inz('Scott Klement');
  street varchar(30)  inz('8825 S Howell Ave');
  city   varchar(20)  inz('Oak Creek');
  state  char(2)      inz('WI');
  postal varchar(10)  inz('53154');
end-ds;

dcl-s Json varchar(1000);
```

```
{
  "name": "Scott Klement",
  "street": "8825 S Howell Ave",
  "city": "Oak Creek",
  "state": "WI",
  "postal": "53154"
}
```

```
 DATA-GEN address %DATA(Json) %GEN('YAJLDTAGEN');
```

The preceding DATA-GEN statement will place a document like the one above in the variable named Json

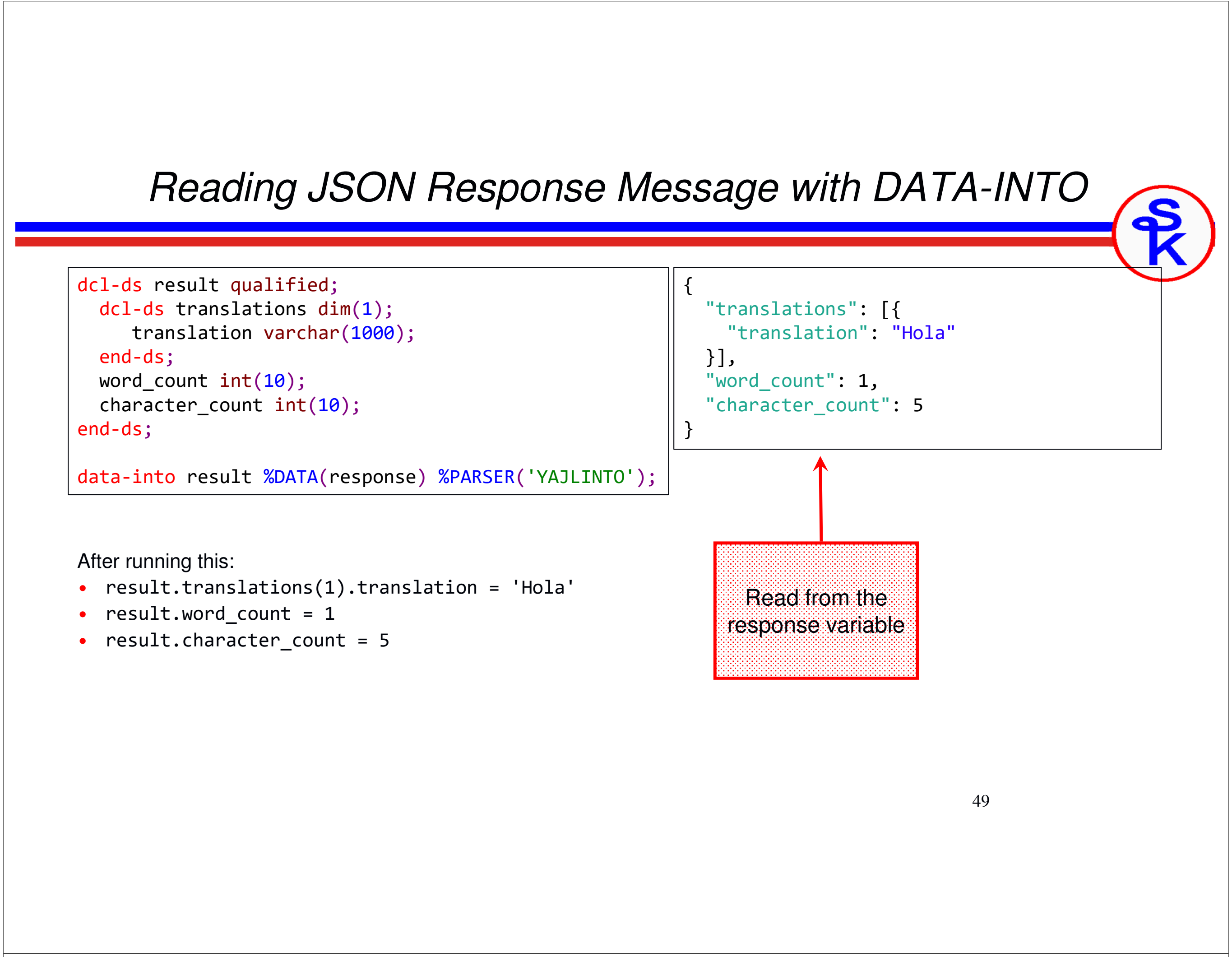
Generating JSON Input Message With DATA-GEN

```
    dcl-ds reqds qualified;
      source varchar(2)    inz('en');
      target varchar(2)    inz('es');
      text   varchar(1000) dim(1);
    end-ds;

    reqds.text(1) = 'Hello';

    data-gen reqds %data(request) %gen('YAJLDTAGEN');
```

```
{
  "source": "en",
  "target": "es",
  "text": [ "Hello" ]
}
```

Placed into request variable

- DCL-DS (start of data structure) generates {
- END-DS (end of data structure) generates }
- DIM generates [and] to indicate array
- Otherwise, fields are generated according to their name/type.



Reading JSON Response Message with DATA-INTO

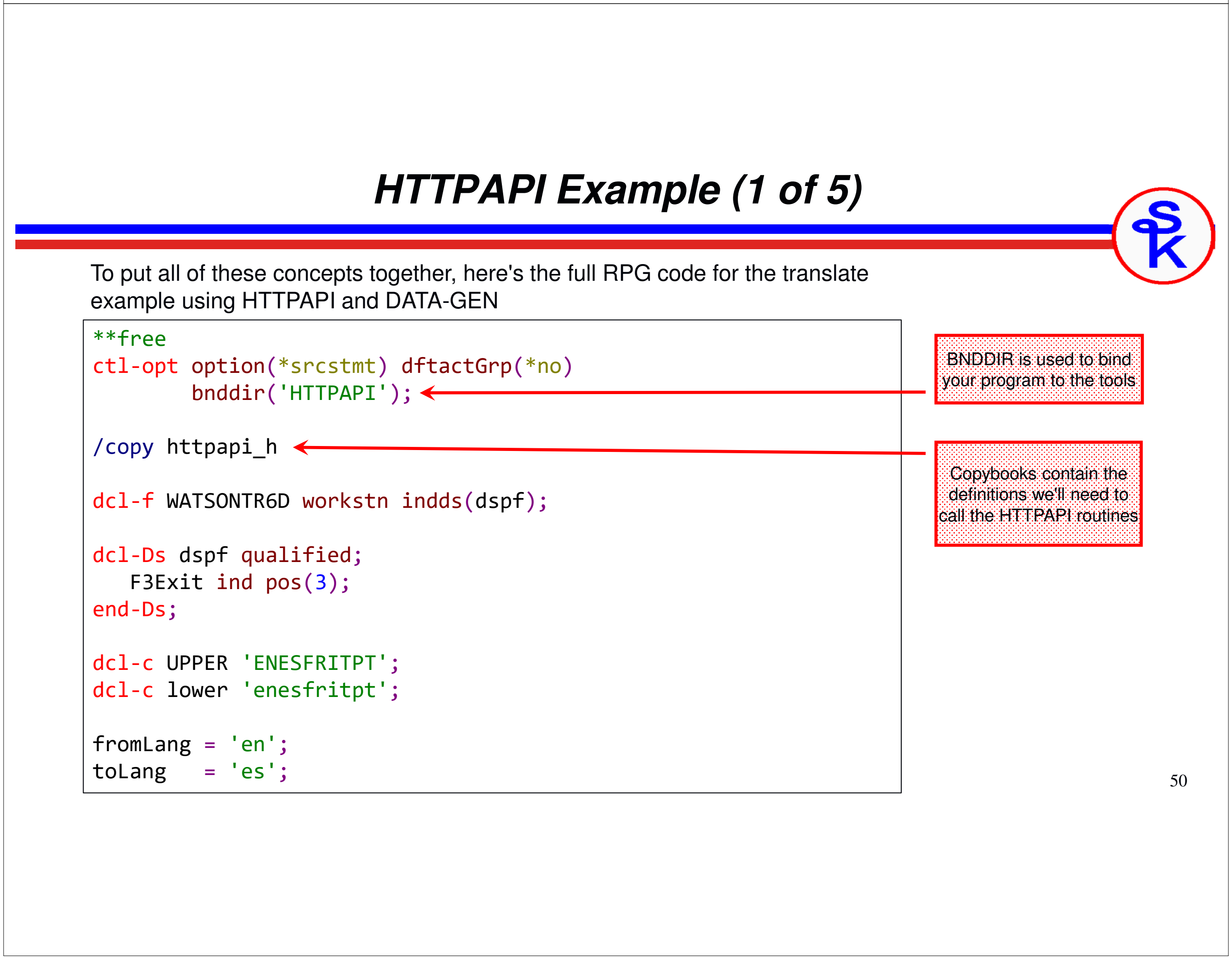
```
dcl-ds result qualified;
  dcl-ds translations dim(1);
     translation varchar(1000);
  end-ds;
  word_count int(10);
  character_count int(10);
end-ds;

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

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

Read from the response variable

After running this:

- `result.translations(1).translation = 'Hola'`
- `result.word_count = 1`
- `result.character_count = 5`



HTTPAPI Example (1 of 5)

To put all of these concepts together, here's the full RPG code for the translate example using HTTPAPI and DATA-GEN

```
**free
ctl-opt option(*srcstmt) dftactGrp(*no)
        bnddir('HTTPAPI');

/copy httpapi_h

dcl-f WATSONTR6D workstn indds(dspf);

dcl-Ds dspf qualified;
   F3Exit ind pos(3);
end-Ds;

dcl-c UPPER 'ENESFRITPT';
dcl-c lower 'enesfritpt';

fromLang = 'en';
toLang   = 'es';
```

BNDDIR is used to bind your program to the tools

Copybooks contain the definitions we'll need to call the HTTPAPI routines

HTTPAPI Example (2 of 5)

Main loop controls the flow of the program, repeating the screen until F3 key is pressed.

```
dou dspf.F3Exit = *on;

   exfmt screen1;
   if dspf.F3exit = *on;
      leave;
   endif;

   fromLang = %xlate(UPPER:lower:fromLang);
   toLang   = %xlate(UPPER:lower:toLang);
   toText = translate( fromLang: toLang: %trim(fromText) );

enddo;

*inlr = *on;
return;
```

the translate procedure is what actually calls Watson

HTTPAPI Example (3 of 5)

```
dcl-proc translate;

   dcl-pi *n varchar(1000);
      fromLang char(2)       const;
      tolang   char(2)       const;
      fromText varchar(1000) const;
   end-pi;

   dcl-s url      varchar(2000);
   dcl-s request  varchar(2000);
   dcl-s response varchar(5000);
   dcl-s httpstatus int(10);

   dcl-ds result qualified;         // {
     dcl-ds translations dim(1);    //   "translations": [{
        translation varchar(1000);  //      "translation": "{string}"
     end-ds;                        //   },
     word_count int(10);            //   "word_count": {number},
     character_count int(10);       //   "character_count": {number}
   end-ds;                          // }
```

Most of this slide is just ordinary RPG definitions

Data structure must match the JSON format for the output parameters.

HTTPAPI Example (4 of 5)

```
dcl-ds reqds qualified;          // {
  source varchar(2);             //   "source": "{string}",
  target varchar(2);             //   "target": "{string}",
  text   varchar(1000) dim(1);   //   "text": [ "{string}" ]
end-ds;                          // }

// Generate the JSON document to send

reqds.source  = fromLang;
reqds.target  = toLang;
reqds.text(1) = fromText;

data-gen reqds %data(request) %gen('YAJLDTAGEN');
```

This RPG data structure matches the format of the JSON that is to be sent to Watson

Populate the data structure with languages and text passed into this subprocedure.

Generate the JSON into a variable named 'request'

HTTPAPI Example (5 of 5)

```
   http_debug(*on: '/tmp/watson-diagnostic-log.txt');

   http_setAuth( HTTP_AUTH_BASIC
               : 'apikey'
               : 'your-Watson-api-key-goes-here');

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


   monitor;
      response = http_string('POST': url: request: 'application/json');
   on-error;
      httpcode = http_error();
   endmon;

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

   return result.translations(1).translation;

end-proc;
```

Enable a diagnostic ("trace") of HTTP session.

Set User/Password

Send 'request' (input) and get back 'response' (output)

Load output into 'result' using data-into

Return the first string translation back to mainline of program

Error Handling with HTTPAPI

http_string throws an exception if there's an error. If you don't mind the user receiving an exception when something goes wrong, you can code as follows (and let the OS handle it.)

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

To handle it yourself, use RPG's monitor/on-error opcodes.

```
monitor;
  response = http_string('POST': url: request: 'application/json');
on-error;
  errorMsg = http_error();
endmon;
```

http_error() returns the last error message. You can also use it to get the last error number and HTTP status code by passing optional parameters.

```
  dcl-s msg varchar(100);
  dcl-s errnum int(10);
  dcl-s status int(10);

  msg = http_error( errnum : status );
```

SQL QSYS2 HTTP Functions

Included in IBM's QSYS2 schema (library)

- Added in September 2021 (7.3 TR11, 7.4 TR5, 7.5 at GA)
- Updated in subsequent TRs and group PTFs
- The best part? Nothing to install!
- The next best? Easy to use!

Unlike SYSTOOLS, Doesn't Use Java!!

- Therefore HTTP_POST is much faster than HTTPPOSTCLOB (same for other similar operations, HTTP_GET, HTTP_PUT, HTTP_DELETE run better than HTTPxxxCLOB versions.)
- Need a "real" CCSID. Your job should not be 65535. This is because data is sent/received in Unicode

Provides:

- HTTP routines
- Routines for reading/writing XML/JSON
- URLENCODE and BASE64 routines

SQL Functions Available

HTTP Routines

- HTTP_GET(), HTTP_POST, HTTP_PUT(), HTTP_DELETE(), HTTP_PATCH()
- HTTP_GET_VERBOSE(), HTTP_POST_VERBOSE(), HTTP_PUT _VERBOSE(), HTTP_DELETE_VERBOSE(), HTTP_PATCH_VERBOSE()

JSON/XML Routines

- JSON_TABLE
- JSON_OBJECT, JSON_ARRAY, et al
- XMLTABLE
- BASE64ENCODE or BASE64DECODE
- URLENCODE or URLDECODE

https://www.ibm.com/docs/en/i/7.5?topic=programming-http-functions-overview

Same Example with SQL

Included in IBM's QSYS2 schema (library)

- No need to rewrite whole program
- Just re-write the `translate()` subprocedure.

We need to

- Create a JSON object (JSON_OBJECT function) as a character string
- Send the character string via HTTP POST method (HTTP_POST)
- Receive the response as a character string
- Interpret the received JSON string (JSON_TABLE)

NOTE:

- Its not required that we use the SQL JSON together with the SQL HTTP routines
- We could use YAJL for JSON and SQL for HTTP
- Or SQL for JSON and HTTPAPI for HTTP
- etc.

HTTP_POST Syntax

HTTP_POST is an SQL function (UDF) you can call from within another SQL statement. (Typically a VALUES or SELECT statement.)

```
HTTP_POST( url, requestMessage, options )
```

- url = an expression containing the URL to connect to
- requestMessage = an expression containing the message to send
- options = a string expression (formatted as JSON) containing options that control the request.

Returns: A CLOB(2g) CCSID 1208 containing the response from the server

Note: All of the above are UTF-8 (CCSID 1208). SQL will automatically perform conversions, so be sure your job CCSID is set properly.

For example, the EBCDIC typically used in the USA is CCSID 37. If your QCCSID system value isn't set properly, you can override it temporarily in the job like this:

```
CHGJOB CCSID(37)
```

SQL HTTP Options

Options are

- Formatted as JSON
- If an option has multiple parameters, they are separated with commas

```
"option-name": "option parameter 1,option parameter 2"
```

```
{
  "basicAuth":  "MyUserId,MyPassword",
  "connectTimeout": 180,
  "header": "Content-type,application/json; charset=UTF-8",
  "header": "Accept,application/json,*",
  "redirect": 5
}
```

Some options are:

- basicAuth = userid/password needed to log in with basic authentication
- connectTimeout = seconds to wait for connection before timing out
- redirect = number of times to follow a redirect before failing
- header = HTTP header to include (may be specified multiple times)

All options are documented here:
https://www.ibm.com/docs/en/i/7.4?topic=functions-http-get#rbafzscahttpget__HTTP_options

Simple HTTP_POST Example:

```
request = '{ "test": "json" }';

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

options = '{ "basicAuth": "apikey,my-password-here", +
             "header": "content-type,application/json" }';

exec SQL
    values QSYS2.HTTP_POST(:url, :request, :options)
      into :response;
```

This will

- Connect to the given URL
- Log in as userid=apikey, password=my-password-here
- Tell the server at the URL to expect data in application/json format
- Send the (mocked up example) JSON
- Receive the response into the "response" variable

SQL JSON Publishing (1 of 2)

Create a JSON object:

```
JSON_OBJECT( KEY 'name' VALUE 'val', KEY 'name2' VALUE 'val2')
JSON_OBJECT( 'name' VALUE 'val', 'name2' VALUE 'val2' )
JSON_OBJECT( 'name': 'val', 'name2': 'val2' )
```

Result:

{ "name": "val", "name2": "val2" }

- The three syntaxes all do the same thing. (The word KEY is optional, and the word VALUE can be replaced with a colon.)
- Instead of a character string, the value can be a number, another json object, or a json array.
- Remember: These are SQL functions, used within an SQL statement.

SQL JSON Publishing (2 of 2)

Create a JSON array:

```
JSON_ARRAY( 'val1', 'val2', etc )
JSON_ARRAY( full-select )
```

Result:

```
[ "val1", "val2", "val3" ]
```

- Instead of a character string, the values can be numbers or other json object/arrays
- The full-select is an SQL select statement. It must return only a single column.
- If one full-select is given, it may return multiple rows. Each row becomes its own array entry.
- It's possible to list multiple select statements or combine them with values. In that case, the select statement must return only one row.

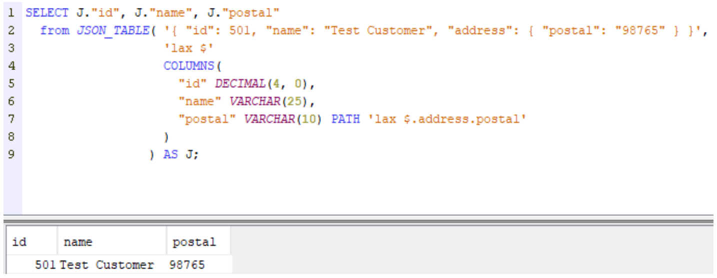
SQL Reading JSON

JSON_TABLE is an SQL table function (UDTF)

This is mean to read a JSON document and treat the output as an SQL table, allowing you to query it, use it in a program, etc.

```
JSON_TABLE( json-document, path COLUMNS( column-definitions ))
```

- json-document = the json document as a char, varchar, clob, etc
- path = path within the JSON document to be read
- column-definitions = defines each column and how to retrieve it



```
SELECT J."id", J."name", J."postal"
  from JSON_TABLE( '{ "id": 501, "name": "Test Customer", "address": { "postal": "98765" } }',
                   'lax $'
                   COLUMNS(
                     "id" DECIMAL(4, 0),
                     "name" VARCHAR(25),
                     "postal" VARCHAR(10) PATH 'lax $.address.postal'
                   )
                 ) AS J;
```

```
id   name           postal
  501 Test Customer  98765
```

SQL HTTP Example (1 of 4)

```
dcl-proc translate;

   dcl-pi *n varchar(1000);
      fromLang char(2)       const;
      tolang   char(2)       const;
      fromText varchar(1000) const;
   end-pi;

   dcl-s userid   varchar(10);
   dcl-s password varchar(200);
   dcl-s url      varchar(2000);
   dcl-s request  varchar(2000);
   dcl-s response varchar(5000);
   dcl-s retval   varchar(1000);
   dcl-s options  varchar(1000);
```

Just some definitions, here…

SQL HTTP Example (2 of 4)

```
exec sql values json_object(
                'source' value lower(:fromLang),
                'target' value lower(:toLang),
                'text' value json_array(:fromText)
              )
          into :request;
if %subst(sqlstt:1:2) <> '00' and %subst(sqlstt:1:2) <> '01';
  // Handle error
endif;

userid = 'apikey';
password = 'password';

exec sql values json_object(
               'basicAuth' value :userid || ',' || :password,
               'header' value 'Content-Type,application/json'
              )
          into :options;
if %subst(sqlstt:1:2) <> '00' and %subst(sqlstt:1:2) <> '01';
  // Handle error
endif;
```

```
{
  "source": "en",
  "target": "es",
  "text": [ "Hello" ]
}
```

```
{
  "basicAuth": "apikey,password",
  "header": "Content-type,application/json"
}
```

SQL HTTP Example (3 of 4)

```
   url = 'https://+
          api.us-south.language-translator.watson.cloud.ibm.com+
          /instances/66f38a33-6f74-492a-8025-8a2e1759a228+
          /v3/translate?version=2018-05-01';

   exec SQL
     values QSYS2.HTTP_POST(:url, :request, :options)
       into :response;

   if %subst(sqlstt:1:2) <> '00' and %subst(sqlstt:1:2) <> '01';
      retval = '**ERROR IN HTTP_POST: SQLSTT=' + sqlstt;
      return retval;
   endif;
```

This will

- Connect/Login with the options from the previous slide
- Send the JSON document created on the previous slide
- Return the output from the server into "response"

SQL HTTP Example (4 of 4)

```
   exec SQL SELECT J."translation"
              into :retval
              from JSON_TABLE(:response, 'lax $'
                     COLUMNS(
                       "translation" VARCHAR(1000)
                         PATH 'lax $.translations[0].translation'
                     )
                   ) as J;

   if %subst(sqlstt:1:2) <> '00' and %subst(sqlstt:1:2) <> '01';
      retval = '** ERROR READING JSON: SQLSTT=' + sqlstt;
      return retval;
   endif;

   return retval;    // Will contain: Hola

end-proc;
```

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

Alternately, Combine All SQL Into One Statement

```
exec SQL SELECT J."translation"
         into :retval
         from JSON_TABLE(
                HTTP_POST(
                  :url,
                  json_object(
                    'source' value lower(:fromLang),
                    'target' value lower(:toLang),
                    'text' value json_array(:fromText)
                  ),
                  json_object(
                    'basicAuth' value :userid || ',' || :password,
                    'header' value 'Content-Type,application/json'
                  )
                ),
                'lax $' COLUMNS(
                  "translation" VARCHAR(1000)
                    PATH 'lax $.translations[0].translation'
                )
              ) as J;
if %subst(sqlstt:1:2) <> '00' and %subst(sqlstt:1:2) <> '01';
   retval = '** ERROR CALLING API: SQLSTT=' + sqlstt;
   return retval;
endif;
```

Db2 SYSTOOLS (OLDER SQL)

Included in IBM's SYSTOOLS schema (library)

- First added in 2014, just after IBM i 7.2 release.
- Updated several times in Technology Refreshes for 7.1+
- The best part? Nothing to install!
- The next best? Easy to use!

Uses Java Under the Covers

- You must have a JVM (1.6 or newer) installed
- Starts the JVM in each job (performance considerations)
- Need a "real" CCSID. Your job should not be 65535.

Provides:

- HTTP routines
- Routines for reading/writing XML/JSON
- URLENCODE and BASE64 routines

SQL Functions in SYSTOOLS

HTTP Routines

- HTTPxxxBLOB or HTTPxxxCLOB functions *(xxx can be GET, POST, PUT or DELETE)*
- HTTPBLOB or HTTPCLOB functions
- HTTPxxxBLOBVERBOSE or HTTPxxxCLOBVERBOSE table functions
- HTTPHEAD

JSON/XML Routines

- JSON_TABLE
- JSON_OBJECT, JSON_ARRAY, et al
- XMLTABLE
- BASE64ENCODE or BASE64DECODE
- URLENCODE or URLDECODE

https://www.ibm.com/support/knowledgecenter/ssw_ibm_i_74/rzajq/rzajqudfhttpclob.htm

Same Example with SYSTOOLS

Included in IBM's SYSTOOLS schema (library)

- No need to rewrite whole program
- Just re-write the `translate()` subprocedure.

We need to

- Create a JSON object (JSON_OBJECT function) as a character string
- Send the character string via HTTP POST method (HTTPPOSTCLOB)
- Receive the response as a character string
- Interpret the received JSON string (JSON_TABLE)

NOTE:

- Its not required that we use the SQL JSON together with the SQL HTTP routines
- We could use YAJL for JSON and SQL for HTTP
- Or SQL for JSON and HTTPAPI for HTTP
- etc.

HTTPPOSTCLOB Syntax

HTTPPOSTCLOB is an SQL function (UDF) you can call from within another SQL statement. (Typically a SELECT statement.)

```
HTTPPOSTCLOB( url, headersXML, requestMessage )
```

- url = a varchar(2048) containing the URL to connect to
- headersXML = a CLOB(10k) containing an XML document that specifies any custom HTTP headers. *(Can be null if you don't wish to customize the headers)*
- requestMessage = a CLOB(2G) containing the message to send

Returns: A CLOB(2g) containing the response from the server

Note: All of the above are UTF-8 (CCSID 1208). SQL will automatically perform conversions, so be sure your job CCSID is set properly.

For example, the EBCDIC typically used in the USA is CCSID 37. If your QCCSID system value isn't set properly, you can override it temporarily in the job like this:

```
    CHGJOB CCSID(37)
```

SQL JSON Publishing (1 of 2)

Create a JSON object:

```
JSON_OBJECT( KEY 'name' VALUE 'val', KEY 'name2' VALUE 'val2')
JSON_OBJECT( 'name' VALUE 'val', 'name2' VALUE 'val2' )
JSON_OBJECT( 'name': 'val', 'name2': 'val2' )
```

Result:

{ "name": "val", "name2": "val2" }

- The three syntaxes all do the same thing. (The word KEY is optional, and the word VALUE can be replaced with a colon.)
- Instead of a character string, the value can be a number, another json object, or a json array.
- Remember: These are SQL functions, used within an SQL statement.

SQL JSON Publishing (2 of 2)

Create a JSON array:

```
JSON_ARRAY( 'val1', 'val2', etc )
JSON_ARRAY( full-select )
```

Result:

```
[ "val1", "val2", "val3" ]
```

- Instead of a character string, the values can be numbers or other json object/arrays
- The full-select is an SQL select statement. It must return only a single column.
- If one full-select is given, it may return multiple rows. Each row becomes its own array entry.
- It's possible to list multiple select statements or combine them with values. In that case, the select statement must return only one row.

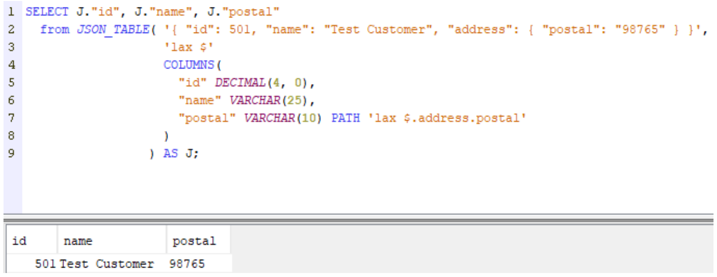
SQL Reading JSON

JSON_TABLE is an SQL table function (UDTF)

This is mean to read a JSON document and treat the output as an SQL table, allowing you to query it, use it in a program, etc.

```
JSON_TABLE( json-document, path COLUMNS( column-definitions ))
```

- json-document = the json document as a char, varchar, clob, etc
- path = path within the JSON document to be read
- column-definitions = defines each column and how to retrieve it



```
SELECT J."id", J."name", J."postal"
  from JSON_TABLE( '{ "id": 501, "name": "Test Customer", "address": { "postal": "98765" } }',
                    'lax $'
                    COLUMNS(
                      "id" DECIMAL(4, 0),
                      "name" VARCHAR(25),
                      "postal" VARCHAR(10) PATH 'lax $.address.postal'
                    )
                  ) AS J;
```

```
id     name           postal
   501 Test Customer  98765
```

Db2 SQL Example (1 of 4)

```
dcl-proc translate;

   dcl-pi *n varchar(1000);
      fromLang char(2)       const;
      tolang   char(2)       const;
      fromText varchar(1000) const;
   end-pi;

   dcl-s userid   varchar(10);
   dcl-s password varchar(200);
   dcl-s hdr      varchar(200);
   dcl-s url      varchar(2000);
   dcl-s request  varchar(2000);
   dcl-s response varchar(5000);
   dcl-s retval   varchar(1000);
```

Most of this slide is just ordinary RPG definitions

Db2 SQL Example (2 of 4)

```
   exec sql select json_object(
                   'source' value :fromLang,
                   'target' value :toLang,
                   'text' value json_array(:fromText)
                  )
            into :request
            from SYSIBM.SYSDUMMY1;

   if %subst(sqlstt:1:2) <> '00' and %subst(sqlstt:1:2) <> '01';
      retval = '**ERROR CREATING: SQLSTT=' + sqlstt;
      return retval;
   endif;
```

Error checking is done the same as any other SQL statement.

```
json_object(
  'source' value 'en',
  'target' value 'es',
  'text' value json_array('Hello')
)
```

⇨

```
{
  "source": "en",
  "target": "es",
  "text": [ "Hello" ]
}
```

Db2 SQL Example (3 of 4)

```
userid = 'apikey';
password = 'your-Watson-api-key-goes-here';

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

hdr = '<httpHeader>+
       <header name="Content-Type" value="application/json" />+
       </httpHeader>';

exec SQL
  select SYSTOOLS.HTTPPOSTCLOB(:url, :hdr, :request)
    into :response
    from SYSIBM.SYSDUMMY1;

if %subst(sqlstt:1:2) <> '00' and %subst(sqlstt:1:2) <> '01';
   retval = '**ERROR IN HTTP: SQLSTT=' + sqlstt;
   return retval;
endif;
```

The easiest way to do user/password is add them to the URL

The SYSTOOLS http functions only support Basic authentication

Error checking is done the same as any other SQL statement.

It is a challenge to get the HTTP status code with HTTPPOSTCLOB

Db2 SQL Example (4 of 4)

```
   exec SQL SELECT J."translation"
            into :retval
            from JSON_TABLE(:response, 'lax $'
                   COLUMNS(
                     "translation" VARCHAR(1000)
                                   PATH 'lax $.translations[0].translation'
                   )
                 ) as J;

   if %subst(sqlstt:1:2) <> '00' and %subst(sqlstt:1:2) <> '01';
      retval = '** ERROR READING: SQLSTT=' + sqlstt;
      return retval;
   endif;

   return retval;

end-proc;
```

JSON_TABLE is a syntax for mapping JSON into a virtual table.

Once it is viewed as a table, you can SELECT INTO to get it into an RPG variable

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

Error Handling with Db2 SQL

Since the HTTP, JSON, XML, etc functions in Db2 are simply SQL statements, you can tell if something failed by checking SQLSTATE (SQLSTT) or SQLCODE (SQLCOD) the same as you would a regular SQL statement.

```
   exec SQL  (any SQL statement here);

   if %subst(sqlstt:1:2) <> '00' and %subst(sqlstt:1:2) <> '01';
      retval = '** SQL ERROR: SQLSTT=' + sqlstt;
      return retval;
   endif;
```

However, this does not provide a lot of detail about the problem.

Calling the VERBOSE table functions (example: HTTPPOSTCLOBVERBOSE) does provide a little more information but does not provide in-depth diagnostics.

For example, if you provide an invalid URL, you simply get back a null.
But if you connect to a valid host and it returns "404 Not Found" you can get that message from the VERBOSE function.

Db2 SQL HTTP Functions

Links to details for the various SQL functions in the IBM Knowledge Center

SQL HTTP routines:
https://www.ibm.com/support/knowledgecenter/ssw_ibm_i_74/rzajq/rzajqhttpoverview.htm

JSON_OBJECT
https://www.ibm.com/support/knowledgecenter/ssw_ibm_i_74/db2/rbafzscajsonobject.htm

JSON_ARRAY
https://www.ibm.com/support/knowledgecenter/ssw_ibm_i_74/db2/rbafzscajsonarray.htm

JSON_TABLE
https://www.ibm.com/support/knowledgecenter/ssw_ibm_i_74/db2/rbafzscajsontable.htm

Don't forget, these won't work if you have sysval QCCSID = 65535 unless you set the CCSID in your job!

```
chgjob ccsid(37)
```

AXIS Transport API

IBM-supplied

- Comes with the IBM HTTP server, so no need for third-party software
- Runs behind the old wsdl2ws.sh/wsdl2rpg.sh SOAP code
- Designed for C, but IBM provides RPG prototypes
- Shipped with the IWS client code starting in 2008

Documentation

- *https://www.ibm.com/systems/power/software/i/iws/*
- Under “Documentation”, click “Web Services Client for ILE Programming Guide”
- Most of this PDF is aimed at SOAP with IBM’s generator.
- Needed Transport APIs are in Chapter 17, under “Transport C APIs”

IBM-supplied Examples With RPG

- https://developer.ibm.com/articles/i-send-receive-user-defined-soap-rest-messages-trs/
- https://www-01.ibm.com/support/docview.wss?uid=nas8N1022250

AXIS Routines We Can Call

AXIS Routines

- axiscTransportCreate = Create a handle for an HTTP connection
- axiscTransportDestroy = Destroy connection handle
- axiscSetProperty = Set properties for use in HTTP handle
- axiscGetProperty = Get properties from an HTTP handle
- axiscTransportSend = Connect with HTTP and send data.
- axiscTransportFlush = Data sent is buffered and may not be completely sent until the buffer is flushed (by calling this API)
- axiscTransportReceive = Receive results from HTTP. This may return only part of the data; call it repeatedly to get everything.
- axiscGetLastErrorCode = Retrieve the last error number that occurred
- axiscGetLastError = Retrieve the last error message that occurred
- axiscAxisStartTrace = Create detailed trace of HTTP connection to IFS file

NOTE: The AXIS Transport API does not provide any routines for handling XML, JSON, URL-encoding, Base64 encoding, etc. You would need to use routines from elsewhere.

AXIS Procedure

To use the AXIS routines, the following is needed:

1. Create a handle.
2. Set properties for:
   - HTTP method (GET, POST, PUT, DELETE)
   - Login credentials (Basic Authentication)
   - Content-Type HTTP Header
   - TLS/SSL options
3. Send data, then flush send buffer
4. Receive data in a loop until there's no more to receive
5. Get the property for the HTTP status code
6. Destroy handle
7. If any of the above returns an error, call the routines to get error number/message.

Same Example with AXIS

To Use AXIS C for HTTP

- No need to rewrite whole program
- Just re-write the `translate()` subprocedure.
- Except: We need to include the AXIS copybook and bind to the QAXIS10CC service program.

```
CRTBNDDIR BNDDIR(your-lib/AXIS)
ADDBNDDIRE BNDDIR(your-lib/AXIS) OBJ((QSYSDIR/QAXIS10CC *SRVPGM))
```

```
ctl-opt option(*srcstmt) dftactGrp(*no)
        bnddir('AXIS': 'YAJL');

/copy yajl_h
/copy /QIBM/ProdData/OS/WebServices/V1/client/include/Axis.rpgleinc
```

Since AXIS doesn't provide routines to work with JSON documents, we will:

- Use SQL to create the JSON
- Use YAJL with DATA-INTO to read the JSON

AXIS Example (1 of 9)

```
dcl-proc translate;

   dcl-pi *n varchar(1000);
      fromLang char(2)       const;
      tolang   char(2)       const;
      fromText varchar(1000) const;
   end-pi;

   dcl-s userid   varchar(10);
   dcl-s password varchar(200);
   dcl-s hdr      varchar(200);
   dcl-s url      varchar(2000);
   dcl-s request  varchar(2000);
   dcl-s response varchar(5000);
   dcl-s rcvBuf   char(5000);
   dcl-s rc       int(10);
   dcl-s propName char(200);
   dcl-s propVal  char(200);
   dcl-s transportHandle pointer;

   dcl-ds result qualified;
     dcl-ds translations dim(1);
        translation varchar(1000) inz('');
     end-ds;
     word_count int(10) inz(0);
     character_count int(10) inz(0);
   end-ds;
```

Most of this slide is just ordinary RPG definitions

Data structure must match the JSON format for the output parameters. (Same as earlier examples.)

AXIS Example (2 of 9)

```
exec sql select json_object(
                  'source' value :fromLang,
                  'target' value :toLang,
                  'text' value json_array(:fromText)
                )
             into :request
             from SYSIBM.SYSDUMMY1;

if %subst(sqlstt:1:2) <> '00' and %subst(sqlstt:1:2) <> '01';
   return '**ERROR CREATING: SQLSTT=' + sqlstt;
endif;
```

Using SQL To Create JSON Document (same as previous example)

AXIS Example (3 of 9)

```
axiscAxisStartTrace('/tmp/axistransport.log': *NULL);


userid = 'apikey';
password = 'your-Watson-api-key-here';

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

transportHandle = axiscTransportCreate(url: AXISC_PROTOCOL_HTTP11);
if (transportHandle = *null);
  failWithError(transportHandle: 'axiscTransportCreate');
endif;
```

Create detailed diagnostic ("trace") log of HTTP session

Set up transport to use the Watson URL and the HTTP 1.1 protocol. (This is the only supported protocol.)

AXIS Example (4 of 9)

```
propName = 'POST' + x'00';
axiscTransportSetProperty( transportHandle
                         : AXISC_PROPERTY_HTTP_METHOD
                         : %addr(propName));

propName = userid + x'00';
propVal  = password + x'00';
axiscTransportSetProperty( transportHandle
                         : AXISC_PROPERTY_HTTP_BASICAUTH
                         : %addr(propName)
                         : %addr(propVal) );

propName = 'Content-Type' + x'00';
propVal  = 'application/json' + x'00';
axiscTransportSetProperty( transportHandle
                         : AXISC_PROPERTY_HTTP_HEADER
                         : %addr(propName)
                         : %addr(propVal) );

propName = '*SYSTEM' + x'00';
propVal = x'00';
axiscTransportSetProperty( transportHandle
                         : AXISC_PROPERTY_HTTP_SSL
                         : %addr(propName)
                         : %addr(propVal) );
```

Use the POST method

Set user/password using basic auth

Set the content-type HTTP header

Tell AXIS to use default TLS/SSL settings from the *SYSTEM certificate store

AXIS Example (5 of 9)

```
rc = axiscTransportSend( transportHandle
                       : %addr(request: *data)
                       : %len(request)
                       : 0 );
if rc = -1;
  failWithError(transportHandle: 'axiscTransportSend');
endif;

rc = axiscTransportFlush(transportHandle);
if rc = -1;
  failWithError(transportHandle: 'axiscTransportFlush');
endif;
```

The network connection begins running here

The %ADDR and %LEN logic converts the 'request' variable into pointers for AXIS

Since data is buffered, it isn't fully sent until the buffer is flushed.

AXIS Example (6 of 9)

```
response = '';

dou rc < 1;

  rc = axiscTransportReceive( transportHandle
                            : %addr(rcvBuf)
                            : %size(rcvBuf)
                            : 0 );
  if rc >= 1;
    response += %subst(rcvBuf:1:rc);
  endif;

enddo;

if rc = -1;
  failWithError(transportHandle: 'axiscTransportReceive');
else;
  httpCode = getHttpStatus(transportHandle);
endif;

axiscTransportDestroy(transportHandle);
```

Data will not be received all at once. Keep calling the receive routine until there's no more data.

After each call, add any new data to the end of the response string

axiscTransportDestroy cleans up the transport when you're done

AXIS Example (7 of 9)

```
    if %len(response) > 0;
      data-into result %DATA(response) %PARSER('YAJLINTO');
    endif;

    return result.translations(1).translation;

end-Proc;
```

With data received, we can use DATA-INTO to interpret the JSON, just as the HTTPAPI example did.

(SQL's JSON_TABLE would've also worked.)

Its worth considering that you can mix/match the different tools:

- HTTPAPI, SQL and AXIS all send a character string
  - It doesn't matter if that string was built with SQL or DATA-GEN
- JSON_TABLE / XMLTABLE interpret a character string
  - It does not matter if that character string was received with HTTPAPI, SQL or AXIS
  - Or even if the string was read from a screen, file, etc.
- Same with DATA-GEN, DATA-INTO, JSON_OBJECT, XMLDOCUMENT, etc.

If you prefer DATA-GEN/DATA-INTO, use them -- even if you use SQL for HTTP
If you prefer HTTPAPI, use it -- even if you prefer SQL for JSON/XML

AXIS Example (8 of 9)

```
dcl-proc getHttpStatus;

  dcl-pi *n varchar(10);
    transportHandle pointer value;
  end-pi;

  dcl-s result varchar(10) inz('');
  dcl-s statusCode pointer;

  if transportHandle <> *null;
     axiscTransportGetProperty( transportHandle
                              : AXISC_PROPERTY_HTTP_STATUS_CODE
                              : %addr(statusCode) );
  endif;

  if statusCode <> *null;
    result = %str(statusCode);
  endif;

  return result;
end-proc;
```

axiscTransportGetProperty can be used to get the HTTP status code

200=OK
403=Forbidden
404=Not Found
500=Server-Side Error

AXIS Example (9 of 9)

```
      lastCode = axiscTransportGetLastErrorCode(transportHandle);
      lastMsg  = %str(axiscTransportGetLastError(transportHandle));

      if lastCode = EXC_TRANSPORT_HTTP_EXCEPTION;
         statusCode = getHttpStatus(transportHandle);
      endif;
```

To save time/space I won't show you the entire error checking routine, just the important parts.

This gets the error number and message.

If the message indicates an HTTP error, it also gets the HTTP status code.

Feature Comparison

Feature	HTTPAPI	SQL	AXIS
Easy to code	🟩 ✓	🟩 ✓	
Performs Well	🟩 ✓	new	🟩 ✓
Shipped with IBM i Operating System		🟩 ✓	🟩 ✓
Basic (plain text) Authentication	🟩 ✓	🟩 ✓	🟩 ✓
NTLM2 (encrypted) Authentication	🟩 ✓		
Retrieve HTTP Status Code	🟩 ✓	🟧 ✓	🟩 ✓
Retrieve Document When Status=Error	🟩 ✓	🟧 ✓	🟩 ✓
Detailed Diagnostic Log	🟩 ✓		🟩 ✓
URL/Forms Encoding Function	🟩 ✓	🟩 ✓	
Multipart/Attachment Encoding Function	🟩 ✓		
Supports Uncommon HTTP Methods	🟩 ✓		
Set Arbitrary HTTP Headers	🟩 ✓	🟩 ✓	🟩 ✓
Supports HTTP Cookies	🟩 ✓		

Conclusions:

- Very few RPGers use AXIS because the coding is complex and hard to maintain
- If you can install a 3rd-party, open-source tool, HTTPAPI offers the most features
- Otherwise, SQL can be a good choice

Legend	
🟩 ✓	Fully Supported
🟧 ✓	VERBOSE functions only
	Not Available

Customer Maintenance Example

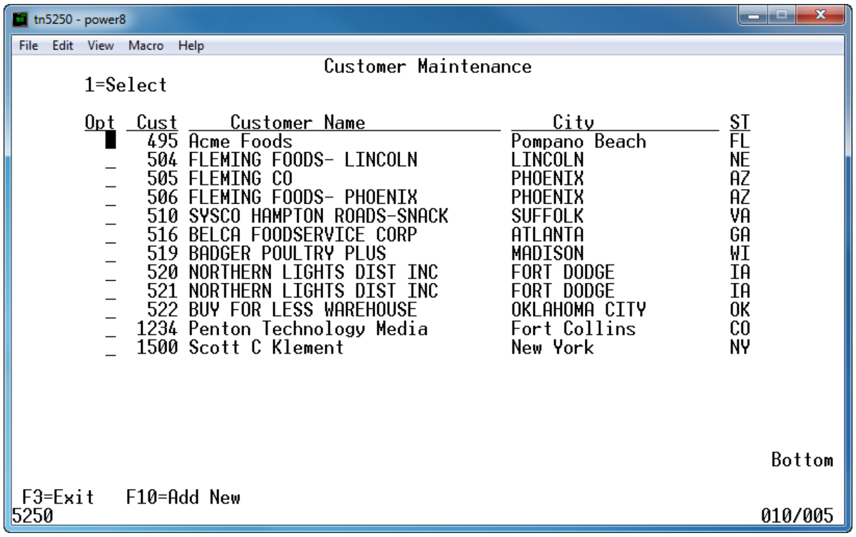
- The Watson example was REST-like, but not truly REST.
  - URI did not indicate the resource
  - POST was used for an idempotent operation
- The best way to fully-demonstrate rest is with a CRUD API
  - Not so easy to find for free on the Internet!
  - Using my own (from the providing section) as an example.
- Customer maintenance example
  - Allows either XML or JSON
  - URI identifies a customer record (the *resource* we're working with)

```
http://my-server/api/customers/1234
```

- GET = retrieve one or all customers (depending on if URI contains the number)
- PUT = update a customer
- POST = create a customer
- DELETE = delete a customer

Customer Maintenance – 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
```

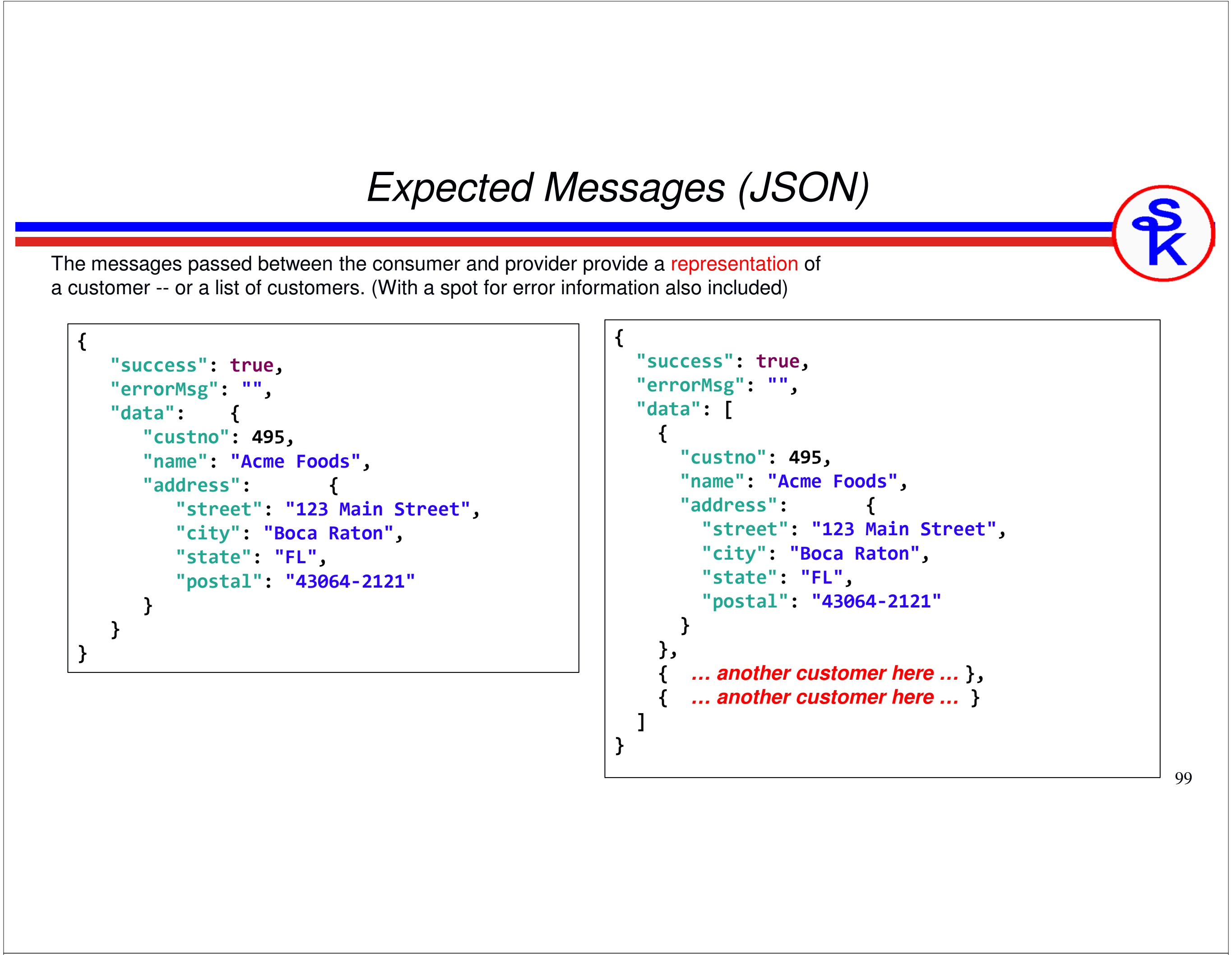
Remember: The REST architecture calls for a layered system.

We will not be accessing the database directly -- but instead, calling an API!

Adds scalability -- can have multiple jobs/servers handling APIs

Adds reusability. APIs can be called from anywhere.

- Other applications
- Web page
- Mobile apps
- etc.

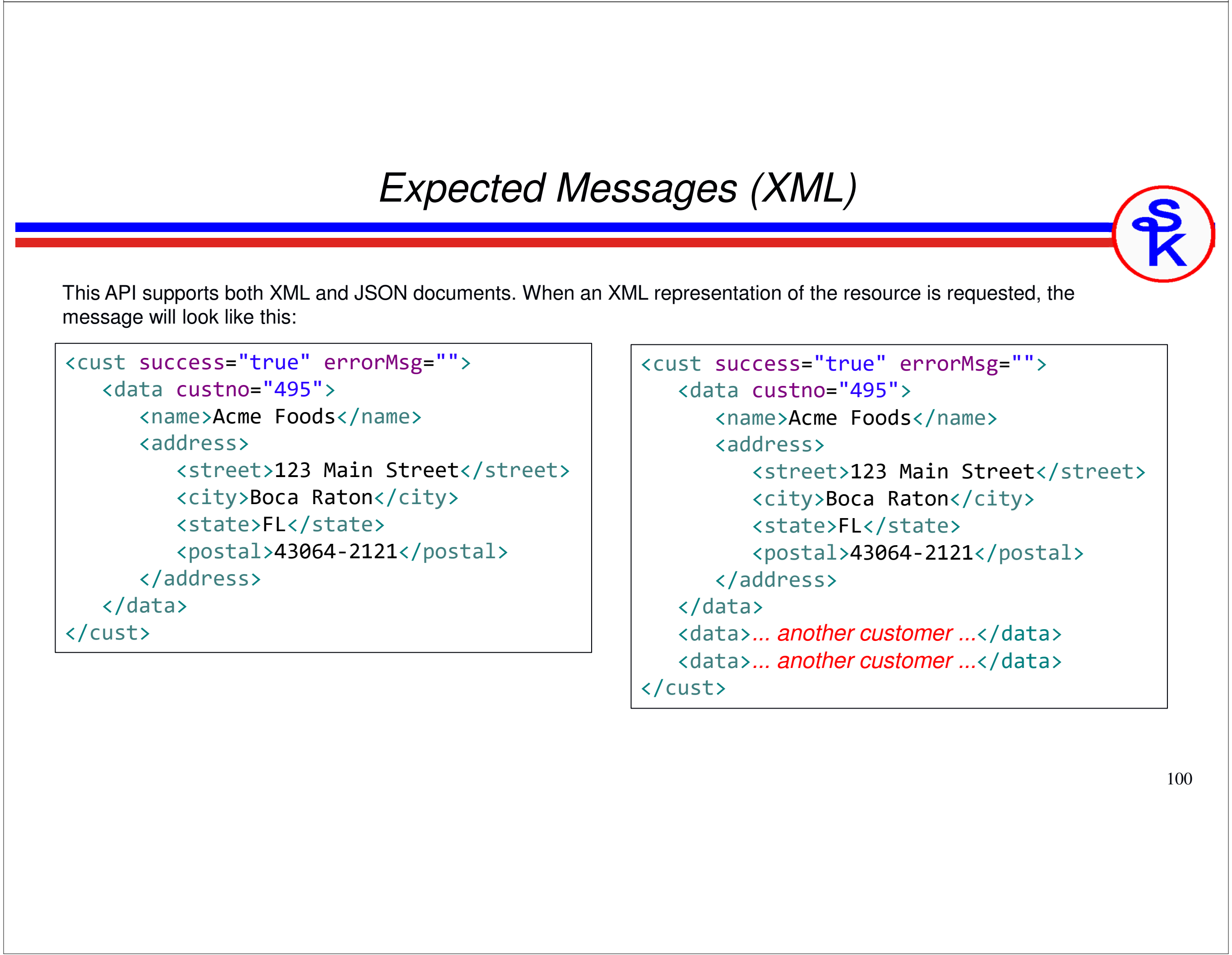


Expected Messages (JSON)

The messages passed between the consumer and provider provide a representation of a customer -- or a list of customers. (With a spot for error information also included)

```
{
   "success": true,
   "errorMsg": "",
   "data":    {
      "custno": 495,
      "name": "Acme Foods",
      "address":       {
         "street": "123 Main Street",
         "city": "Boca Raton",
         "state": "FL",
         "postal": "43064-2121"
      }
   }
}
```

```
{
  "success": true,
  "errorMsg": "",
  "data": [
    {
      "custno": 495,
      "name": "Acme Foods",
      "address":       {
        "street": "123 Main Street",
        "city": "Boca Raton",
        "state": "FL",
        "postal": "43064-2121"
      }
    },
    {  … another customer here … },
    {  … another customer here …  }
  ]
}
```



Expected Messages (XML)

This API supports both XML and JSON documents. When an XML representation of the resource is requested, the message will look like this:

```
<cust success="true" errorMsg="">
   <data custno="495">
      <name>Acme Foods</name>
      <address>
         <street>123 Main Street</street>
         <city>Boca Raton</city>
         <state>FL</state>
         <postal>43064-2121</postal>
      </address>
   </data>
</cust>
```

```
<cust success="true" errorMsg="">
   <data custno="495">
      <name>Acme Foods</name>
      <address>
         <street>123 Main Street</street>
         <city>Boca Raton</city>
         <state>FL</state>
         <postal>43064-2121</postal>
      </address>
   </data>
   <data>... another customer ...</data>
   <data>... another customer ...</data>
</cust>
```

Specifying Media Types

Since this API supports both XML and JSON, you need to tell it which format you wish to use. There is a standard for specifying document types used in HTTP (as well as other Internet media, such as E-mail) called media types.

(Often known by the older name "MIME type")

Here are some examples:

Media type (MIME type)	Meaning
application/json	JSON document
text/xml	XML document
application/xml	Alternative way to specify XML document
image/png	Portable Network Graphic (.png) images
image/jpeg	JPEG (.jpg) images
text/plain	Plain text (.txt) file
text/csv	Comma Separated Values (.csv) file

Standard HTTP Headers for Media Types

The HTTP protocol provides a place to specify media types in two different scenarios:

- `content-type` = When sending data you use this to tell the API what type of document you are sending
- `accept` = Tells the API what type(s) of response document you're willing to accept

For example, to get a list of customers in XML representation:

```
GET http://ibmi.example.com/api/customers
Accept: text/xml
```

To get customer 500 in JSON representation:

```
GET http://ibmi.example.com/api/customers/500
Accept: application/json
```

To create a new customer by sending data in JSON format, but get back a response in XML format:

```
POST http://ibmi.example.com/api/customers/500
Accept: text/xml
Content-type: application/json

...data in JSON with representation of new
customer follows...
```

The method of specifying the content-type and accept headers will vary depending on the HTTP tool you use. I will demonstrate how to do it with HTTPAPI.

Time Savers For Next Example

For the Watson Language Translation API, I demonstrated how to use three different HTTP tools:

- HTTPAPI
- Db2 SYSTOOLS functions (HTTPGETCLOB, et al)
- AXIS C

I hope you found that interesting!

However, to save time on the *Customer Maintenance* example, I will:

- only show HTTPAPI
- only show key "snippets" of the code
    - not showing read/write screen, database, etc.
- provide full code for download from http://www.scottklement.com/presentations/

Retrieving All Customers As JSON

This API defaults its output to JSON, so its not necessary to specify the accept header for JSON data.

```
dcl-s  jsonData varchar(100000);
dcl-c  BASEURL 'http://localhost:8500/api/customers';

UserId = 'sklement';
Password = 'bigboy';

http_setAuth( HTTP_AUTH_BASIC: UserId: Password );

monitor;
  jsonData = http_string( 'GET' : BASEURL);
  msg = *blanks;
on-error;
  msg = http_error();
endmon;

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

XML-INTO or DATA-INTO Options

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

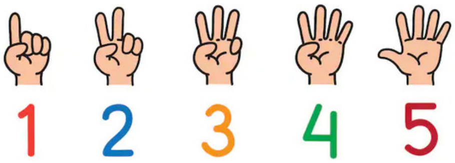
```
dcl-ds cust qualified;
   ...
   num_data int(10);
   dcl-ds data dim(999);
      custno packed(5: 0);
      name   varchar(30);
      ...
   end-ds;
end-ds;
```

case=convert

- upper/lower case in variable names do not need to match
- accented characters are converted to closest un-accented equivalent
- spaces or punctuation symbols are converted to underscores

countprefix=num_

- RPG will calculate a count of the JSON (or XML) elements
  - num_ is the prefix to use
  - fields with the prefix is where the counts are placed
- to get a count of "data" elements, add a "num_data" field
  - begins with the prefix, ends with the name of the element to count



Interpreting the JSON

```
dcl-ds cust qualified;                   // {
   success  ind              inz(*on);   //   "success": true|false,
   errorMsg varchar(500)     inz('');    //   "errorMsg": "{string}",
   num_data int(10)          inz(0);
   dcl-ds data dim(999);                 //   "data": [ {
      custno packed(5: 0)    inz(0);     //     "custno": {number},
      name   varchar(30)     inz('');    //     "name": "{string}",
      dcl-ds address;                    //     "address": {
         street varchar(30) inz('');     //       "street": "{string}",
         city   varchar(20) inz('');     //       "city": "{string}",
         state  char(2)     inz('  ');   //       "state": "{string}",
         postal varchar(10) inz('');     //       "postal": "{string}"
      end-ds;                            //     }
   end-ds;                               //   } ]
end-ds;                                  // }

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

// Now we can load our subfile from the data in 'cust'!
```

Updating a Customer (JSON)

Generating a JSON document is similar to reading it, except DATA-GEN is used instead of DATA-INTO.

```
   dcl-s jsonData varchar(10000);

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

   monitor;
      url = BASEURL + '/' + %char(custno);
      http_string( 'PUT': url: jsonData : 'application/json' );
   on-error;
      msg = http_error();
      return *off;
   endmon;
```

Omitting Fields When Updating

If you wanted to create a "deluxe" version of this program, you could code it so that it only sends the specific fields to be updated.

You can omit fields from the document created by DATA-GEN by using countprefix fields. For example, if you add a `num_name` field to the data structure, and set it to 0, no name element is added to the JSON document.

Advantages:

- Makes the JSON smaller, so quicker to send
- Avoids "phantom refreshes" if two people are updating the document at the same time

Disadvantages:

- The message doesn't contain a "complete" representation of the customer resource.

```
dcl-ds cust qualified;
   ...
   dcl-ds data dim(999);
      ...
      num_name int(10);
      name     varchar(30);
      ...
   end-ds;
end-ds;

cust.data.num_name = 0;

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

Retrieving All Customers As XML (1 of 2)

To retrieve the whole list of customers as XML, we'll need to pass the accept header that tells the API to return data in XML format.

In HTTPAPI you do this with an "xproc" (exit procedure). This is a subprocedure that is called during the HTTP transmission that can add additional headers into the HTTP transmission.

```
dcl-s xmlData varchar(100000);
dcl-c BASEURL 'http://localhost:8500/api/customers';
http_xproc(HTTP_POINT_ADDL_HEADER: %paddr(add_accept_header));

UserId = 'sklement';
Password = 'bigboy';

http_setAuth( HTTP_AUTH_BASIC: UserId: Password );

monitor;
  xmlData = http_string( 'GET' : BASEURL);
  msg = *blanks;
on-error;
  msg = http_error();
endmon;
```

```
dcl-proc add_accept_header;

  dcl-pi *n;
    extraHeader varchar(1024);
  end-pi;

  dcl-c CRLF x'0d25';

  extraHeader += 'Accept: text/xml' + CRLF;

end-proc;
```

Adds the Accept header for XML

Now the xmlData variable will contain the list of all customers in XML format!

Retrieving All Customers As XML (2 of 2)

Once the XML data has been retrieved from the API, you can use XML-INTO to interpret it (just as DATA-INTO was used for JSON)

```
dcl-ds cust qualified;                // <cust
  success varchar(5) inz('true');     //      success="{string}"
  errorMsg varchar(500) inz('');      //      errorMsg="{string}" >
  num_data int(10);
  dcl-ds data dim(999);               //    <data
    custno packed(5: 0) inz(0);       //      custno="{number}" >
    name varchar(30) inz('');         //      <name>{string}</name>
    dcl-ds address;                   //      <address>
      street varchar(30) inz('');     //        <street>{string}</street>
      city   varchar(20) inz('');     //        <city>{string}</city>
      state  char(2)     inz('  ');   //        <state>{string}</state>
      postal varchar(10) inz('');     //        <postal>{string}</postal>
    end-ds;                           //      </address>
  end-ds;                             //    </data>
end-ds;                               // </cust>


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

Now you can load the subfile from the data in the 'cust' data structure.

Generating XML With SQL (1 of 5)

Since IBM i 7.1, Db2 contains functions for creating (or "publishing" as IBM puts it) XML documents.

SQL has its own XML data types, including XML type columns in tables, etc. The XML functions are designed to work with these internal XML types (which, frankly, makes these functions harder to understand than the JSON ones.)

First we create the document as an XML type column with these functions:

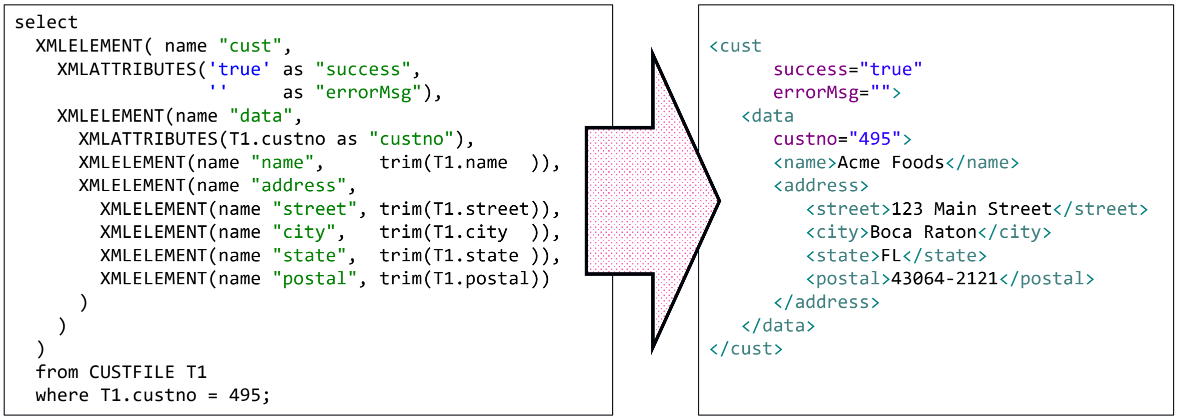
- XMLELEMENT = Creates an XML element ("XML tag") in an XML document
- XMLATTRIBUTES = Creates XML attributes in an XML element

Next, we create a string from the XML type column with XMLSERIALIZE

- XMLSERIALIZE = Creates (or "serializes") a string from XML data

Generating XML With SQL (2 of 5)

- XMLELEMENT = Creates an XML element ("XML tag") in an XML document
  - ○ `XMLELEMENT( name "cust", 'xxx' )` creates `<cust>xxx</cust>`
- XMLATTRIBUTES = Creates XML attributes in an XML element
  - `XMLELEMENT( name "data", XMLATTRIBUTES( '495' as "custno"))` creates `<data custno="495">`



```
select
  XMLELEMENT( name "cust",
    XMLATTRIBUTES('true' as "success",
                  ''     as "errorMsg"),
    XMLELEMENT(name "data",
      XMLATTRIBUTES(T1.custno as "custno"),
      XMLELEMENT(name "name",     trim(T1.name  )),
      XMLELEMENT(name "address",
        XMLELEMENT(name "street", trim(T1.street)),
        XMLELEMENT(name "city",   trim(T1.city  )),
        XMLELEMENT(name "state",  trim(T1.state )),
        XMLELEMENT(name "postal", trim(T1.postal))
      )
    )
  )
  from CUSTFILE T1
  where T1.custno = 495;
```

```
<cust
      success="true"
      errorMsg="">
   <data
      custno="495">
      <name>Acme Foods</name>
      <address>
         <street>123 Main Street</street>
         <city>Boca Raton</city>
         <state>FL</state>
         <postal>43064-2121</postal>
      </address>
   </data>
</cust>
```

Generating XML With SQL (3 of 5)

- XMLAGG = Allows us to aggregate data. In this example, for each database row, we want to repeat the group of XML tags that contain the customer information.

```
select
  XMLELEMENT(name "cust",
    XMLATTRIBUTES( 'true' as "success",
                   ''     as "errorMsg"),
    XMLAGG(
      XMLELEMENT(name "data",
        XMLATTRIBUTES(T1.custno as "custno"),
        XMLELEMENT(name "name",     trim(T1.name  )),
        XMLELEMENT(name "address",
          XMLELEMENT(name "street", trim(T1.street)),
          XMLELEMENT(name "city",   trim(T1.city  )),
          XMLELEMENT(name "state",  trim(T1.state )),
          XMLELEMENT(name "postal", trim(T1.postal))
        )
      )
    )
  )
  from CUSTFILE T1;
```

```
<cust
      success="true"
      errorMsg="">

   <data
      custno="495">
      <name>Acme Foods</name>
      <address>
         <street>123 Main Street</street>
         <city>Boca Raton</city>
         <state>FL</state>
         <postal>43064-2121</postal>
      </address>
   </data>

</cust>
```

These tags repeat for each row in CUSTFILE because they are inside XMLAGG

Generating XML With SQL (4 of 5)

- XMLSERIALIZE = Generates a character string from the XML data type created in the previous examples.

```
dcl-s data    sqltype(CLOB: 5000);

exec sql
   select
     XMLSERIALIZE(
       XMLELEMENT( name "cust",
         XMLATTRIBUTES('true' as "success",
                       ''     as "errorMsg"),
         XMLELEMENT(name "data",
           XMLATTRIBUTES(:custno as "custno"),
           XMLELEMENT(name "name",     trim(:name  )),
           XMLELEMENT(name "address",
             XMLELEMENT(name "street", trim(:street)),
             XMLELEMENT(name "city",   trim(:city  )),
             XMLELEMENT(name "state",  trim(:state )),
             XMLELEMENT(name "postal", trim(:postal))
           )
         )
       )
     AS CLOB(5000) CCSID 1208
     VERSION '1.0' INCLUDING XMLDECLARATION)
   into :data
   from SYSIBM/SYSDUMMY1 T1;
```

After the RPG (with embedded SQL) on the left runs, the "data" CLOB will contain the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<cust success="true" errorMsg="">
   <data custno="495">
      <name>Acme Foods</name>
      <address>
         <street>123 Main Street</street>
         <city>Boca Raton</city>
         <state>FL</state>
         <postal>43064-2121</postal>
      </address>
   </data>
</cust>
```

- INCLUDING XMLDECLARATION = adds the <?xml> to the output string.
- CCSID = determines the "encoding"
- VERSION = determines the "version"

Generating XML With SQL (5 of 5)

```
dcl-s sendDoc varchar(5000) inz('');

if data_len > 0;
  sendDoc = %subst(data_data:1:data_len);
else;
  senddoc = '';
endif;

url = BASEURL + '/' + %char(custno);

monitor;
   http_string( 'PUT': url: sendDoc: 'text/xml' );
on-error;
   msg = http_error();
   return *off;
endmon;
```

Since SQL VARCHAR is limited to 32K, I usually like to serialize XML into a CLOB field.

VARCHAR is more convenient to work with in RPG, though, so I use %subst() to convert the CLOB to a VARCHAR.

Then, finally, we can send the XML

Notice that the content-type is also set to indicate XML.

Consuming -- Conclusion

In this section, I have:

Shown a relatively simple API call with Watson Language Translation

- Worked with messages 2 different ways
  - Interpreted JSON with DATA-INTO
  - Interpreted JSON with SQL's JSON_TABLE
  - Created JSON with DATA-GEN
  - Created JSON with SQL's JSON_OBJECT, JSON_ARRAY
- Worked with HTTP 3 different ways
  - HTTPAPI
  - Db2 SQL SYSTOOLS http functions
  - AXIS C functions

Shown a more sophisticated (and "true" REST) Customer Maintenance API

- Worked with messages 4 different ways
  - Created JSON with DATA-GEN
  - Interpreted JSON with DATA-GEN
  - Interpreted XML with SQL's XMLTABLE
  - Created XML with SQL's XMLELEMENT, XMLATTRIBUTE, XMLAGG and XMLSERIALIZE

Providing REST APIs in RPG

IBM's Integrated Web Services Server

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/

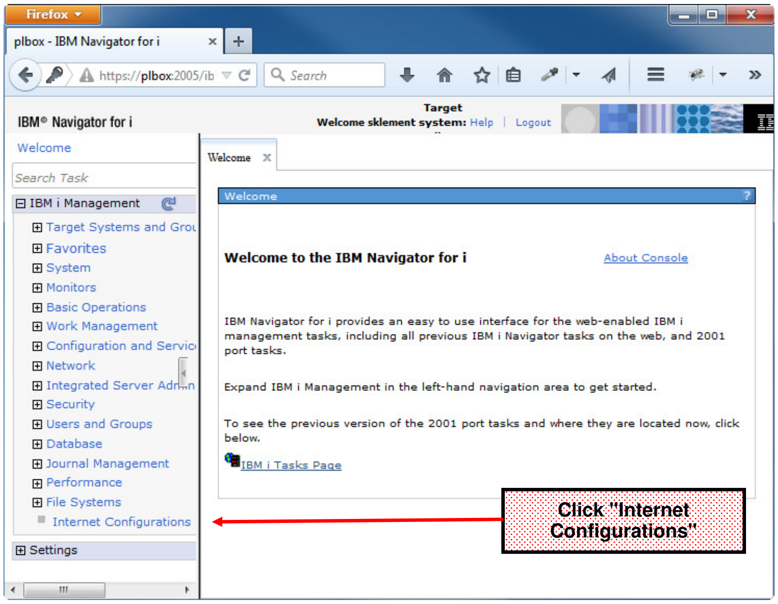
- Can be used to provide web services
- Can also be used to consume them -- but requires in-depth knowledge of C and pointers -- I won't cover IBM's consumer tool today.

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.

IBM Navigator for i (old nav)



Welcome to the IBM Navigator for i

About Console

IBM Navigator for i provides an easy to use interface for the web-enabled IBM i management tasks, including all previous IBM i Navigator tasks on the web, and 2001 port tasks.

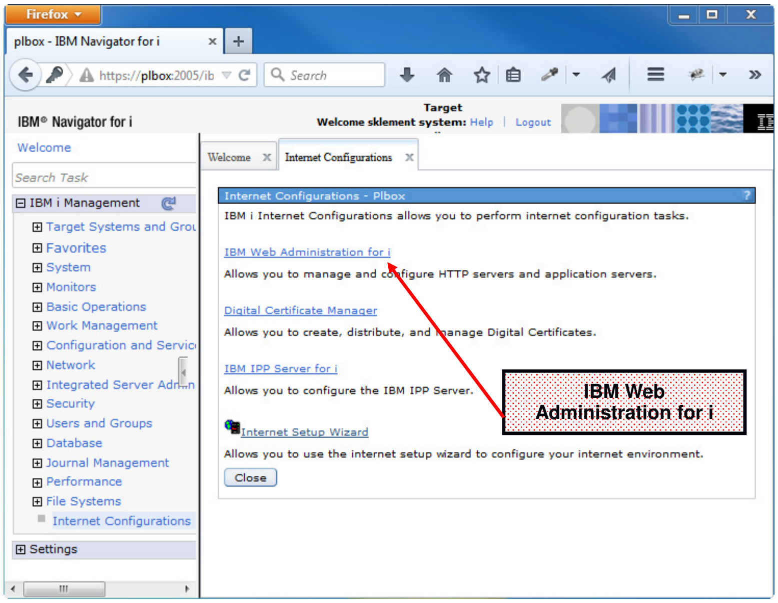
Expand IBM i Management in the left-hand navigation area to get started.

To see the previous version of the 2001 port tasks and where they are located now, click below.

IBM i Tasks Page

Click "Internet Configurations"

Internet Configurations (old nav)



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.

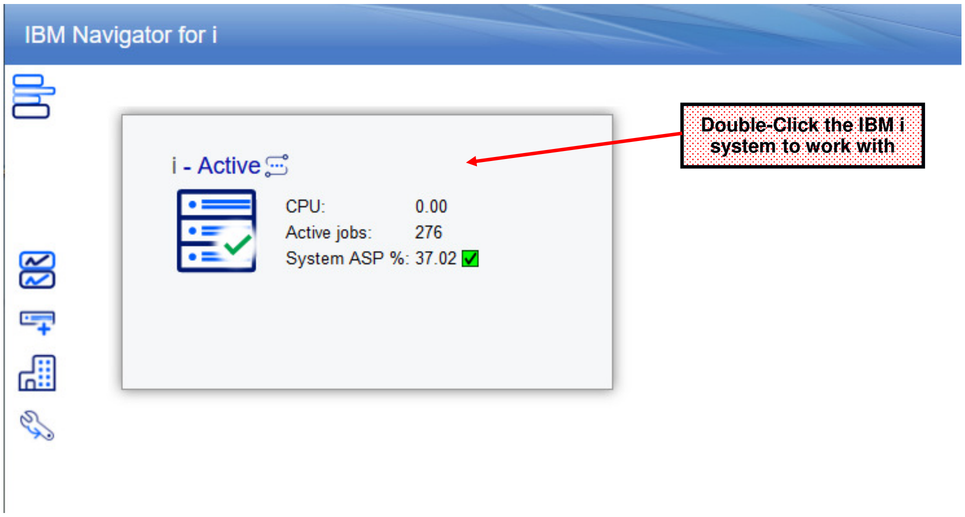
Internet Setup Wizard

Allows you to use the internet setup wizard to configure your internet environment.

Close

IBM Web Administration for i

IBM Navigator for i (new nav)



IBM Navigator for i

i - Active

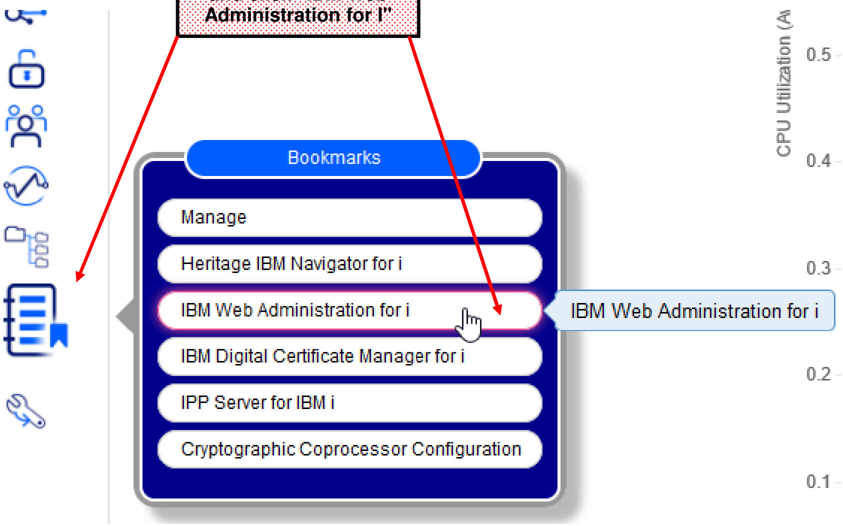
CPU: 0.00

Active jobs: 276

System ASP %: 37.02 ✓

Double-Click the IBM i system to work with

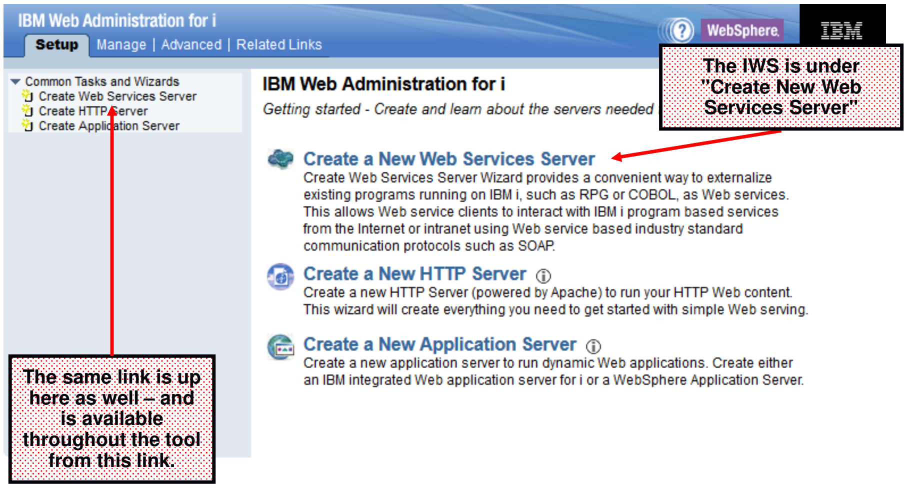
Bookmarks (new nav)



Open the "Bookmarks" item in the lower-left, and click "IBM Web Administration for I"

Bookmarks
Manage
Heritage IBM Navigator for i
IBM Web Administration for i
IBM Digital Certificate Manager for i
IPP Server for IBM i
Cryptographic Coprocessor Configuration
IBM Web Administration for i
CPU Utilization (A
0.5
0.4
0.3
0.2
0.1

Web Administration for i



IBM Web Administration for i

Setup | Manage | Advanced | Related Links

WebSphere. IBM

- Common Tasks and Wizards
  - Create Web Services Server
  - Create HTTP Server
  - Create Application Server

IBM Web Administration for i

Getting started - Create and learn about the servers needed

Create a New Web Services Server
Create Web Services Server Wizard provides a convenient way to externalize existing programs running on IBM i, such as RPG or COBOL, as Web services. This allows Web service clients to interact with IBM i program based services from the Internet or intranet using Web service based industry standard communication protocols such as SOAP.

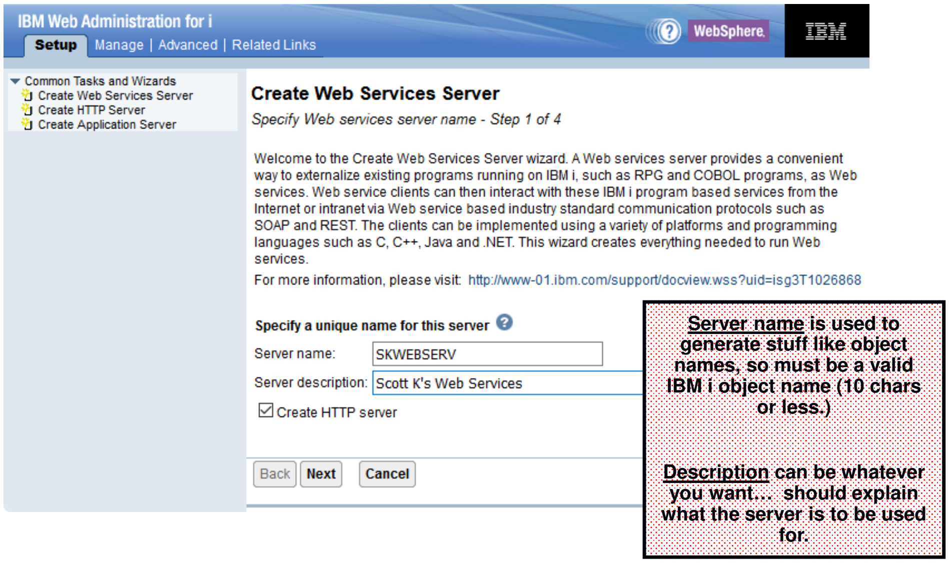
Create a New HTTP Server ⓘ
Create a new HTTP Server (powered by Apache) to run your HTTP Web content. This wizard will create everything you need to get started with simple Web serving.

Create a New Application Server ⓘ
Create a new application server to run dynamic Web applications. Create either an IBM integrated Web application server for i or a WebSphere Application Server.

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)



IBM Web Administration for i

Setup | Manage | Advanced | Related Links

WebSphere. IBM

- Common Tasks and Wizards
  - Create Web Services Server
  - Create HTTP Server
  - Create Application Server

Create Web Services Server

Specify Web services server name - Step 1 of 4

Welcome to the Create Web Services Server wizard. A 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 via Web service based industry standard communication protocols such as SOAP and REST. The clients can be implemented using a variety of platforms and programming languages such as C, C++, Java and .NET. This wizard creates everything needed to run Web services.

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

Specify a unique name for this server

Server name: SKWEBSERV

Server description: Scott K's Web Services

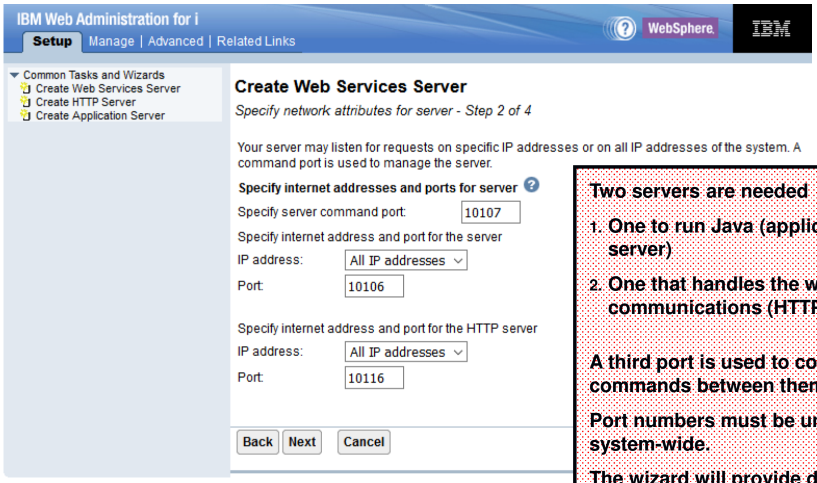
☑ Create HTTP server

Back | Next | Cancel

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

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

Create IWS Server (2 of 4)



IBM Web Administration for i
Setup | Manage | Advanced | Related Links
WebSphere
IBM
Common Tasks and Wizards
Create Web Services Server
Create HTTP Server
Create Application Server

Create Web Services Server

Specify network attributes for server - Step 2 of 4

Your server may listen for requests on specific IP addresses or on all IP addresses of the system. A command port is used to manage the server.

Specify internet addresses and ports for server

Specify server command port: 10107

Specify internet address and port for the server

IP address: All IP addresses

Port: 10106

Specify internet address and port for the HTTP server

IP address: All IP addresses

Port: 10116

Two servers are needed

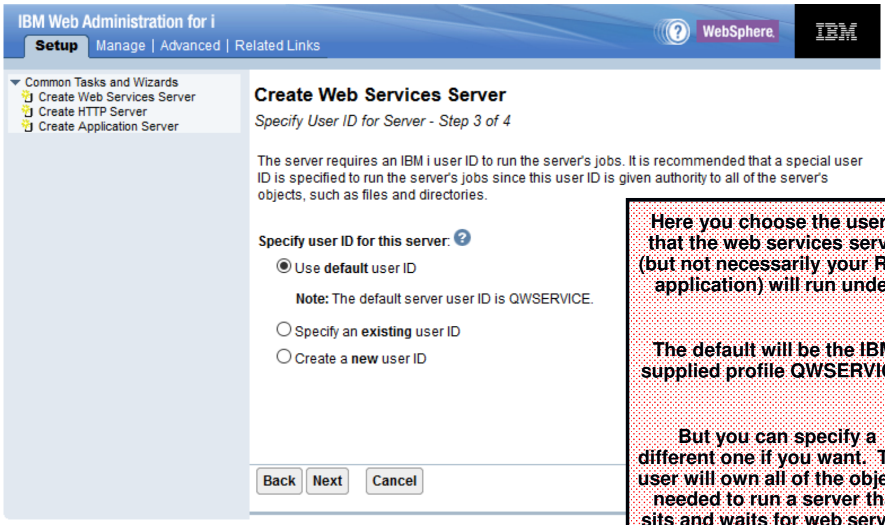
1. **One to run Java (application server)**
2. **One that handles the web communications (HTTP server)**

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)



IBM Web Administration for i
Setup | Manage | Advanced | Related Links
WebSphere
IBM
Common Tasks and Wizards
Create Web Services Server
Create HTTP Server
Create Application Server

Create Web Services Server

Specify User ID for Server - Step 3 of 4

The server requires an IBM i user ID to run the server's jobs. It is recommended that a special user ID is specified to run the server's jobs since this user ID is given authority to all of the server's objects, such as files and directories.

Specify user ID for this server:

● Use **default** user ID

Note: The default server user ID is QWSERVICE.

○ Specify an **existing** user ID

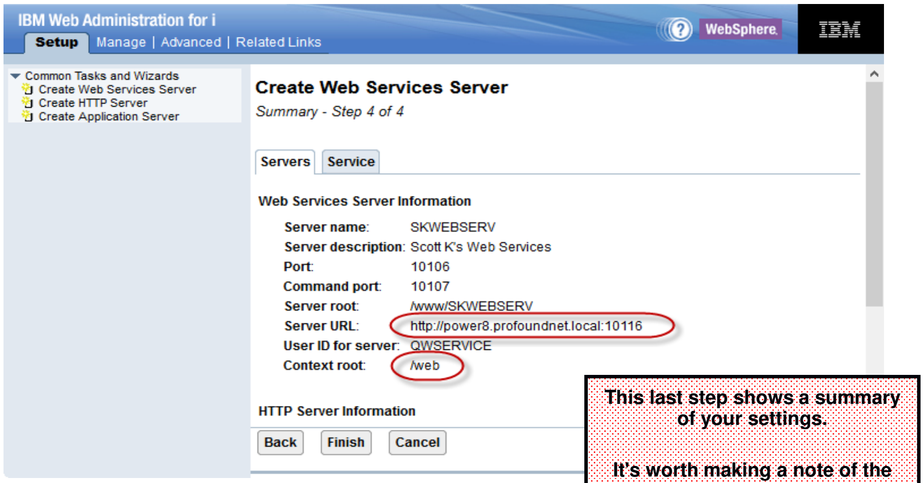
○ Create a **new** user ID

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)


IBM Web Administration for i
Setup | Manage | Advanced | Related Links
WebSphere
IBM
Common Tasks and Wizards
Create Web Services Server
Create HTTP Server
Create Application Server
Servers
Service
Back
Finish
Cancel

Create Web Services Server

Summary - Step 4 of 4

Web Services Server Information

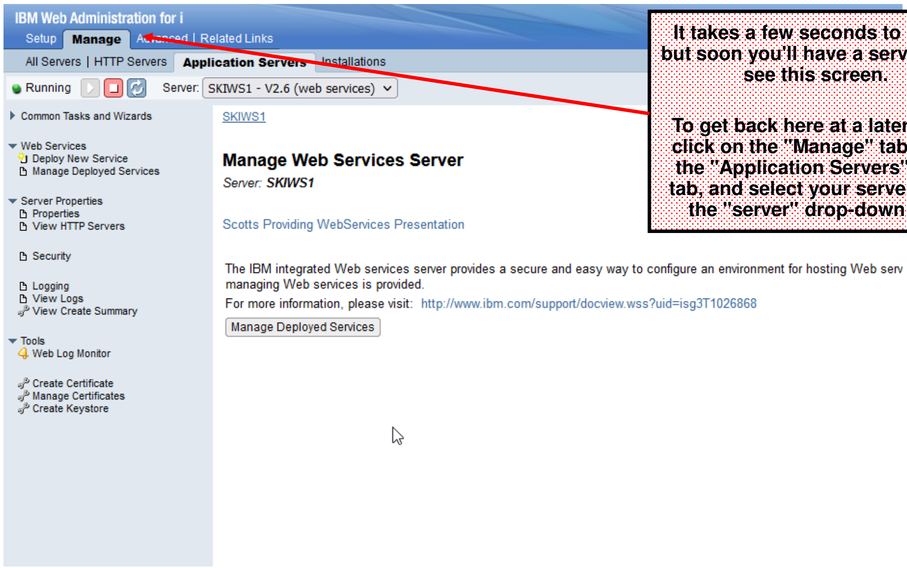
Server name: SKWEBSERV
Server description: Scott K's Web Services
Port: 10106
Command port: 10107
Server root: /www/SKWEBSERV
Server URL: http://power8.profoundnet.local:10116
User ID for server: QWSERVICE
Context root: /web

HTTP Server Information

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!



Manage Web Services Server

*Server: **SKIWS1***

Scotts Providing WebServices Presentation

The IBM integrated Web services server provides a secure and easy way to configure an environment for hosting Web serv managing Web services is provided.

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

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

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

Now What?

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

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

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

GETCUST RPG Program (1 of 2)

```
Ctl-Opt DFTACTGRP(*NO) ACTGRP('WEBAPI') PGMINFO(*PCML:*MODULE);

Dcl-F CUSTFILE Usage(*Input) Keyed PREFIX('CUST.');

Dcl-DS CUST ext extname('CUSTFILE') qualified End-DS;

Dcl-PI *N;
   CustNo                     like(Cust.Custno);
   Name                       like(Cust.Name);
   Street                     like(Cust.Street);
   City                       like(Cust.City);
   State                      like(Cust.State);
   Postal                     like(Cust.Postal);
End-PI;

Dcl-PR QMHSNDPM  ExtPgm('QMHSNDPM');
   MessageID      Char(7)     Const;
   QualMsgF       Char(20)    Const;
   MsgData        Char(32767) Const options(*varsize);
   MsgDtaLen      Int(10)     Const;
   MsgType        Char(10)    Const;
   CallStkEnt     Char(10)    Const;
   CallStkCnt     Int(10)     Const;
   MessageKey     Char(4);
   ErrorCode      Char(8192) options(*varsize);
End-PR;
```

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.

Since there's no DCL-PROC, the DCL-PI works like the old *ENTRY PLIST

GETCUST RPG Program (2 of 2)

```
Dcl-DS err  qualified;
   bytesProv      Int(10)    inz(0);
   bytesAvail     Int(10)    inz(0);
End-DS;

Dcl-S MsgDta       Varchar(1000);
Dcl-S MsgKey       Char(4);
Dcl-S x            Int(10);

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;
```

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 creating JSON or XML for me!

PCML so IWS Knows Our Parameters

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 an XML document that is generated by the RPG compiler.

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:

```
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 or CTL-OPT:

```
Ctl-Opt PGMINFO(*PCML:*MODULE);
```

GETCUST as a REST API

Remember that in REST (sometimes called 'RESTful') APIs:

- 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:

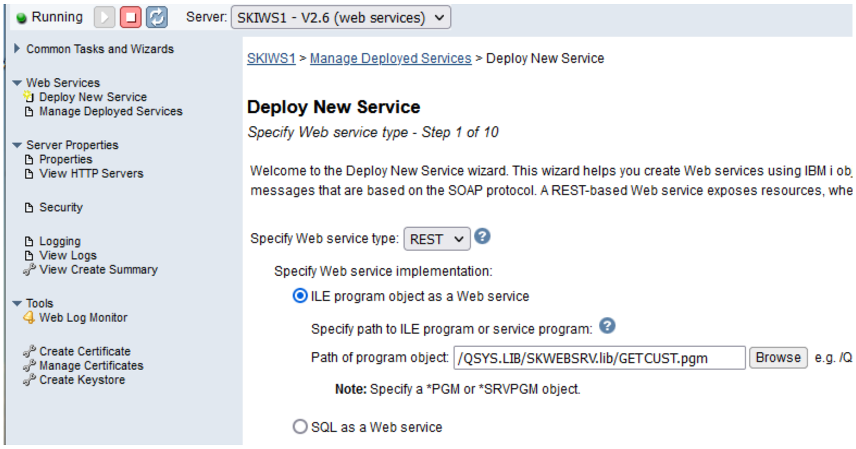
- /web/services is the context root.
- /cust is the root resource (and will point to our GETCUST program)
- /495 (or any other customer number) is the path template.

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

REST Wizard (1 of 10)

The type (dropdown) should be REST.

You can use a program or SQL statement – for this example, I'll specify an ILE program and type the IFS path of the GETCUST program.



SKIWS1 > Manage Deployed Services > Deploy New Service

Deploy New Service

Specify Web service type - Step 1 of 10

Welcome to the Deploy New Service wizard. This wizard helps you create Web services using IBM i ob
messages that are based on the SOAP protocol. A REST-based Web service exposes resources, whe

Specify Web service type: REST

Specify Web service implementation:

● ILE program object as a Web service

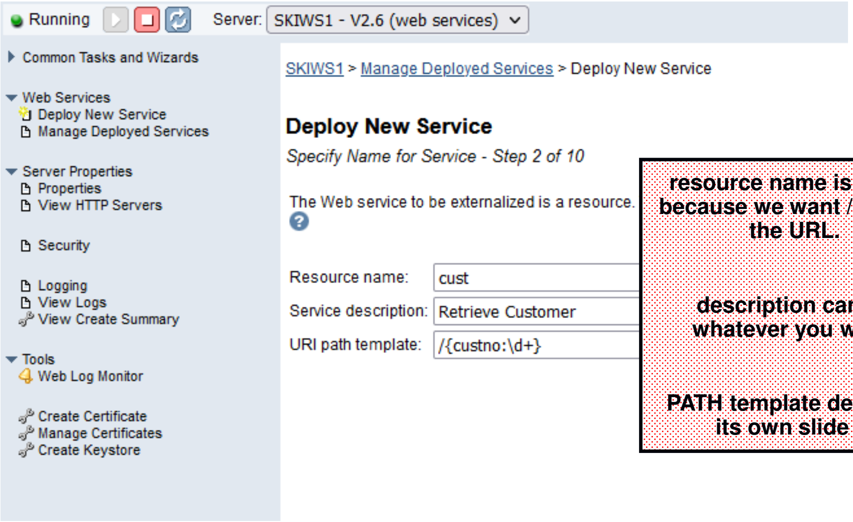
Specify path to ILE program or service program:

Path of program object: /QSYS.LIB/SKWEBSRV.lib/GETCUST.pgm Browse e.g. /Q

Note: Specify a *PGM or *SRVPGM object.

○ SQL as a Web service

REST Wizard (2 of 10)



SKIWS1 > Manage Deployed Services > Deploy New Service

Deploy New Service

Specify Name for Service - Step 2 of 10

The Web service to be externalized is a resource.

Resource name: cust

Service description: Retrieve Customer

URI path template: /{custno:\d+}

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:

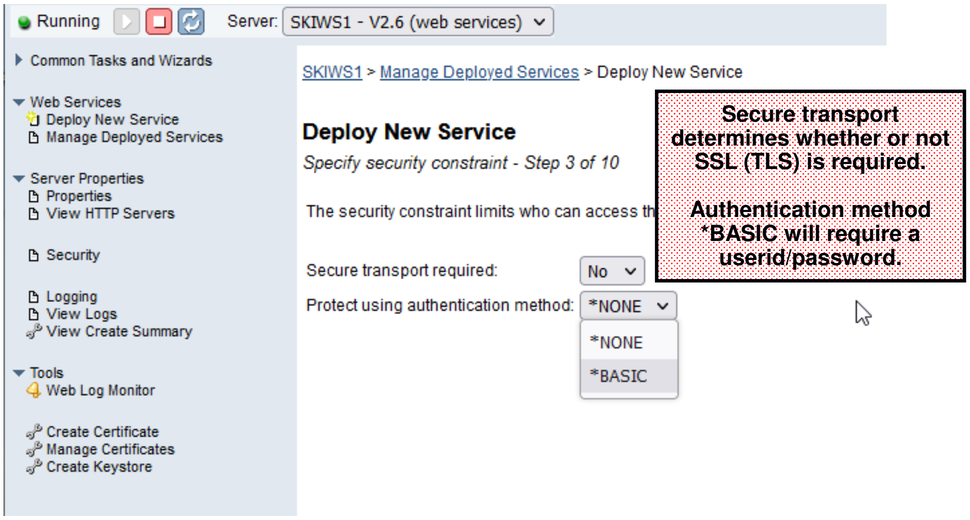
- Maybe there are multiple path variables separated by slashes
- Maybe they allow only numeric values
- Maybe they allow only letters, or only uppercase letters, or only lowercase, or both letters and numbers
- maybe they have to have certain punctuation, like slashes in a date, or dashes in a phone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

REST Wizard (3 of 10)



Deploy New Service

Specify security constraint - Step 3 of 10

The security constraint limits who can access th

Secure transport required: No

Protect using authentication method: *NONE

Secure transport determines whether or not SSL (TLS) is required.

Authentication method *BASIC will require a userid/password.

Path Template Examples

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

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

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

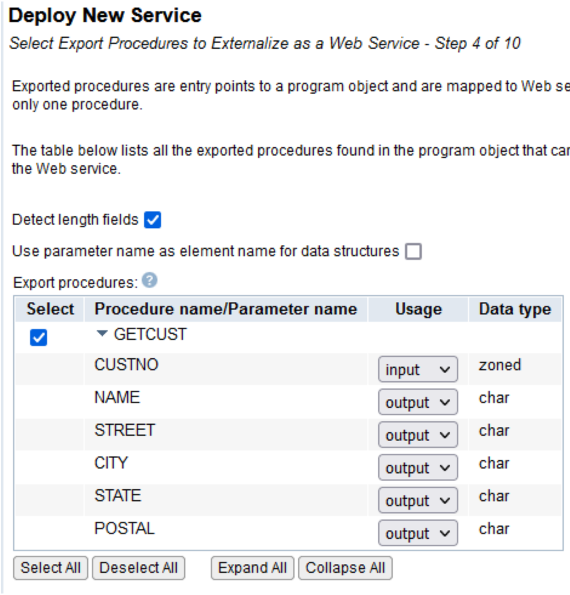
```
/Milwaukee/202/Freezer1/B/12/C
```

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

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

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

REST Wizard (4 of 10)



Deploy New Service

Select Export Procedures to Externalize as a Web Service - Step 4 of 10

Exported procedures are entry points to a program object and are mapped to Web service operations. A procedure is a set of only one procedure.

The table below lists all the exported procedures found in the program object that can be externalized through this Web servic the Web service.

Detect length fields ☑

Use parameter name as element name for data structures □

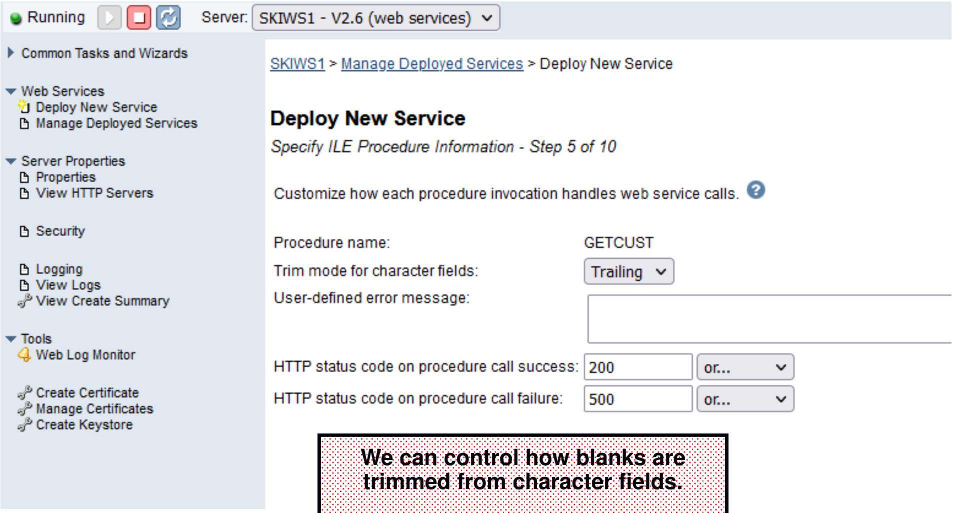
Export procedures:

Select	Procedure name/Parameter name	Usage	Data type
☑	▾ GETCUST		
	CUSTNO	input	zoned
	NAME	output	char
	STREET	output	char
	CITY	output	char
	STATE	output	char
	POSTAL	output	char

"Detect length fields" will look for fields named ending with _LENGTH and treat them as the number of elements for any arrays.

We also need to tell it which parameters are used for input and output from our program.

REST Wizard (5 of 10)



SKIWS1 > Manage Deployed Services > Deploy New Service

Deploy New Service

Specify ILE Procedure Information - Step 5 of 10

Customize how each procedure invocation handles web service calls.

Procedure name: GETCUST

Trim mode for character fields: Trailing

User-defined error message:

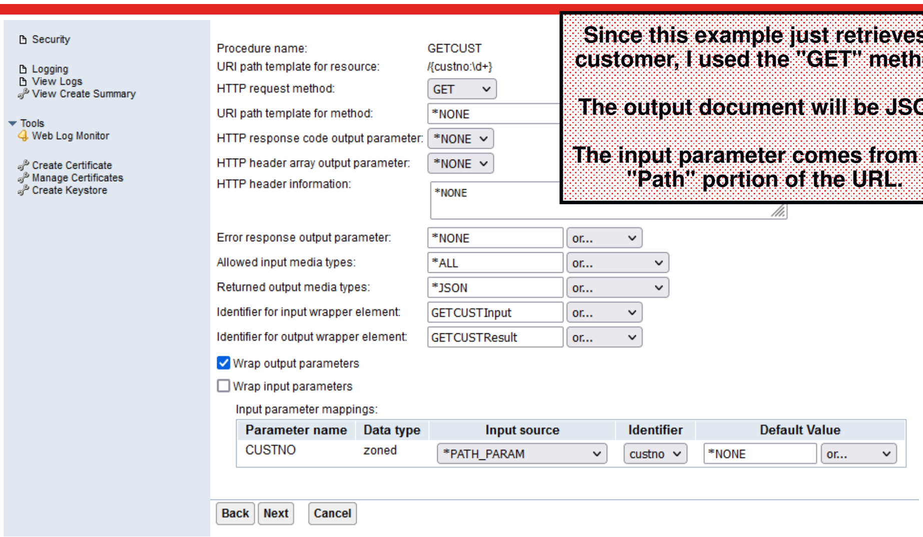
HTTP status code on procedure call success: 200 or...

HTTP status code on procedure call failure: 500 or...

We can control how blanks are trimmed from character fields.

We can also control which HTTP status codes are returned for success/failures.

REST Wizard (6 of 10)



Procedure name: GETCUST

URI path template for resource: /{custno:\d+}

HTTP request method: GET

URI path template for method: *NONE

HTTP response code output parameter: *NONE

HTTP header array output parameter: *NONE

HTTP header information: *NONE

Error response output parameter: *NONE or...

Allowed input media types: *ALL or...

Returned output media types: *JSON or...

Identifier for input wrapper element: GETCUSTInput or...

Identifier for output wrapper element: GETCUSTResult or...

☑ Wrap output parameters

☐ Wrap input parameters

Input parameter mappings:

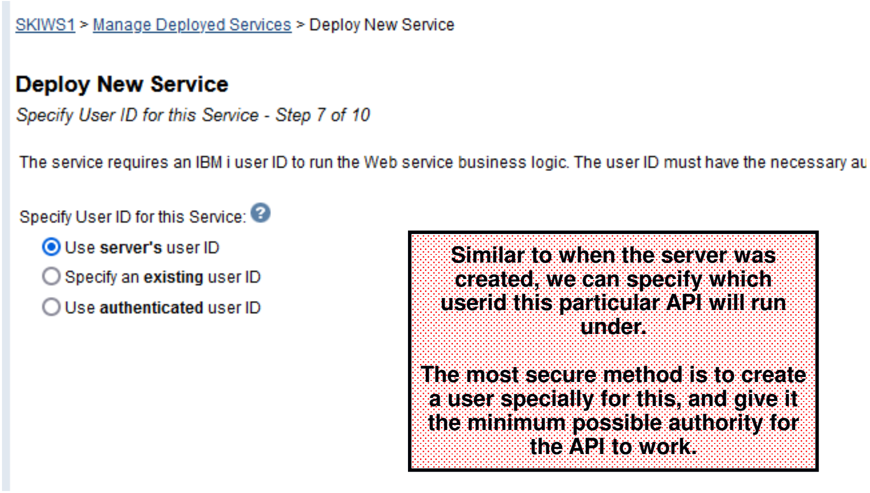
Parameter name	Data type	Input source	Identifier	Default Value
CUSTNO	zoned	*PATH_PARAM	custno	*NONE or...

Since this example just retrieves a customer, I used the "GET" method.

The output document will be JSON.

The input parameter comes from the "Path" portion of the URL.

REST Wizard (7 of 10)



SKIWS1 > Manage Deployed Services > Deploy New Service

Deploy New Service

Specify User ID for this Service - Step 7 of 10

The service requires an IBM i user ID to run the Web service business logic. The user ID must have the necessary au

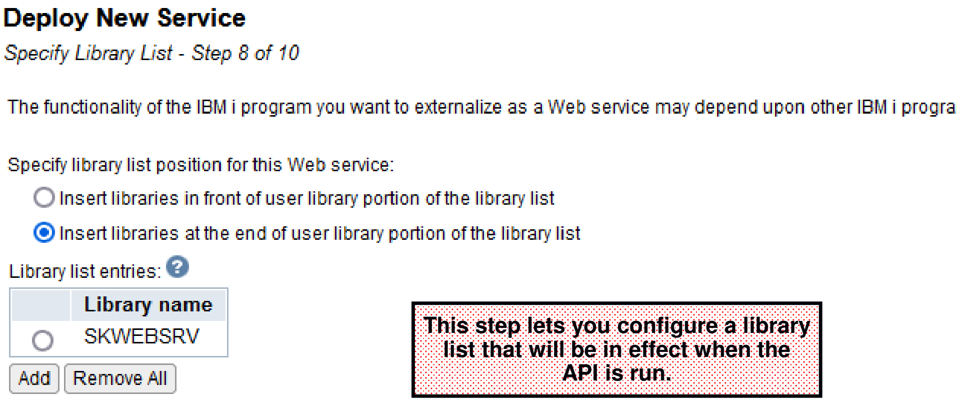
Specify User ID for this Service:

- ● Use **server's** user ID
- ○ Specify an **existing** user ID
- ○ Use **authenticated** user ID

Similar to when the server was created, we can specify which userid this particular API will run under.

The most secure method is to create a user specially for this, and give it the minimum possible authority for the API to work.

REST Wizard (8 of 10)



Deploy New Service

Specify Library List - Step 8 of 10

The functionality of the IBM i program you want to externalize as a Web service may depend upon other IBM i progra

Specify library list position for this Web service:

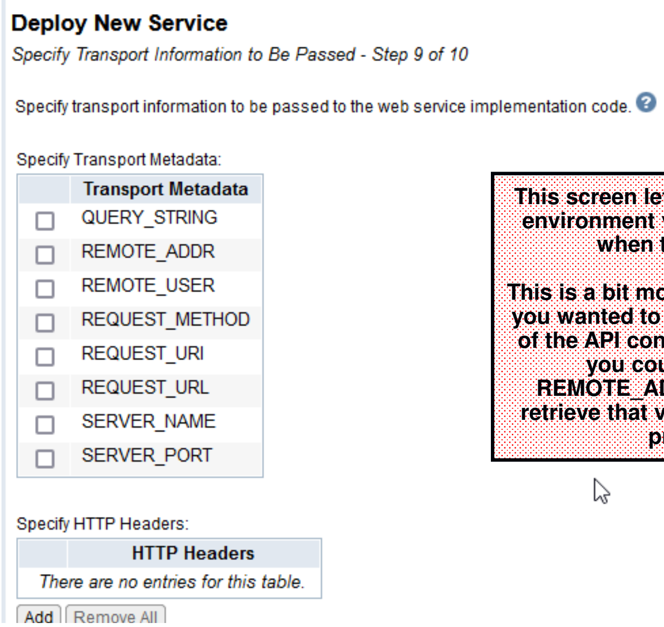
- ○ Insert libraries in front of user library portion of the library list
- ● Insert libraries at the end of user library portion of the library list

Library list entries:

	Library name
○	SKWEBSRV

This step lets you configure a library list that will be in effect when the API is run.

REST Wizard (9 of 10)



Deploy New Service

Specify Transport Information to Be Passed - Step 9 of 10

Specify transport information to be passed to the web service implementation code.

Specify Transport Metadata:

	Transport Metadata
☐	QUERY_STRING
☐	REMOTE_ADDR
☐	REMOTE_USER
☐	REQUEST_METHOD
☐	REQUEST_URI
☐	REQUEST_URL
☐	SERVER_NAME
☐	SERVER_PORT

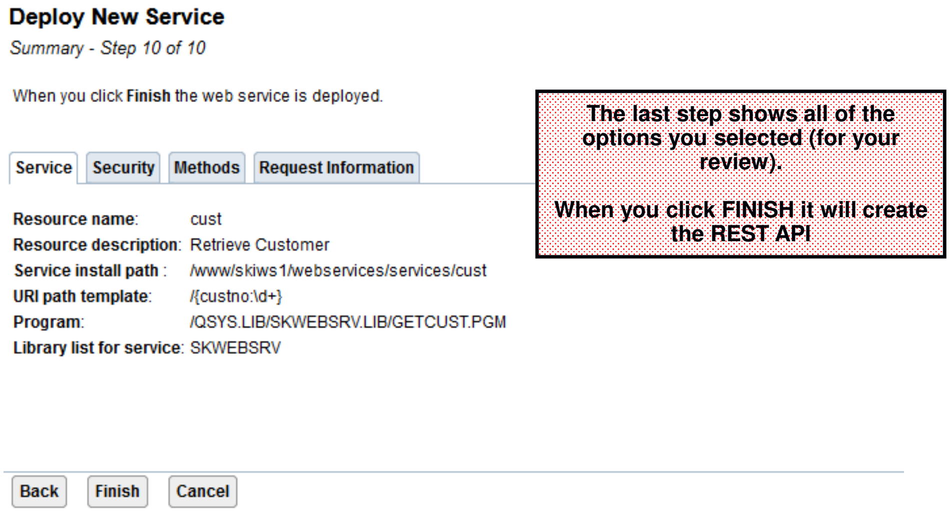
Specify HTTP Headers:

	HTTP Headers
	There are no entries for this table.

This screen lets you control which environment variables will be set when the API runs.

This is a bit more "advanced", but if you wanted to know the IP address of the API consumer, for example, you could enable the REMOTE_ADDR variable, then retrieve that variable in your RPG program.

REST Wizard (10 of 10)



Deploy New Service

Summary - Step 10 of 10

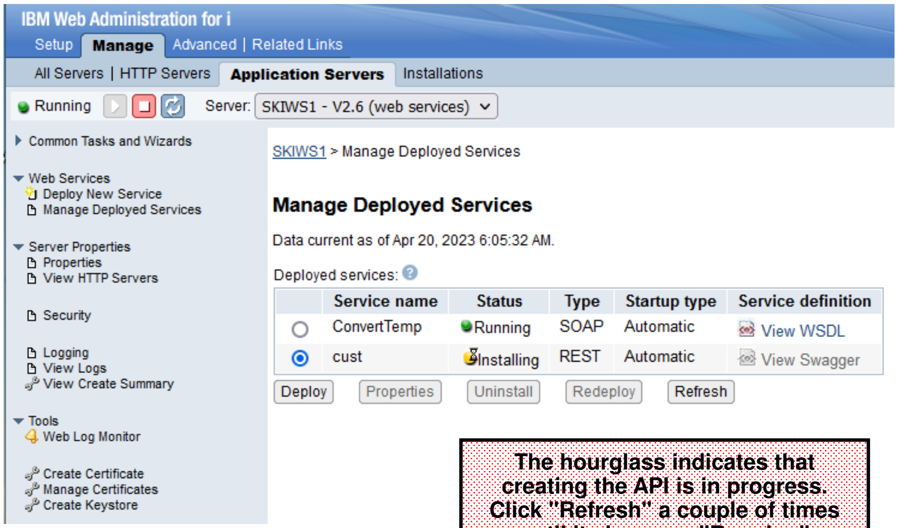
When you click **Finish** the web service is deployed.

Resource name: cust
Resource description: Retrieve Customer
Service install path : /www/skiws1/webservices/services/cust
URI path template: /{custno:\d+}
Program: /QSYS.LIB/SKWEBSRV.LIB/GETCUST.PGM
Library list for service: SKWEBSRV

The last step shows all of the options you selected (for your review).

When you click FINISH it will create the REST API

Wait For the API to Install



Manage Deployed Services

Data current as of Apr 20, 2023 6:05:32 AM.

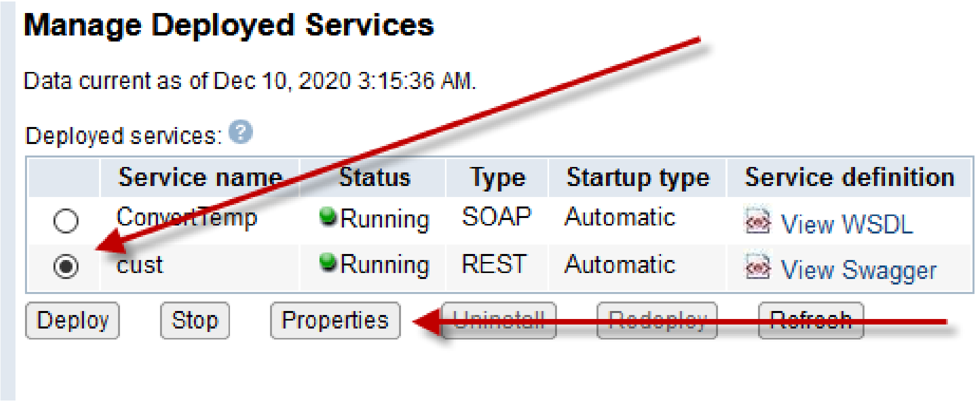
Deployed services:

	Service name	Status	Type	Startup type	Service definition
○	ConvertTemp	● Running	SOAP	Automatic	View WSDL
●	cust	Installing	REST	Automatic	View Swagger

The hourglass indicates that creating the API is in progress. Click "Refresh" a couple of times until it shows as "Running"

Looking Up the URI of Your API (1 of 2)

To determine the URI needed to call your new API, select your service, and click "Properties"



Manage Deployed Services

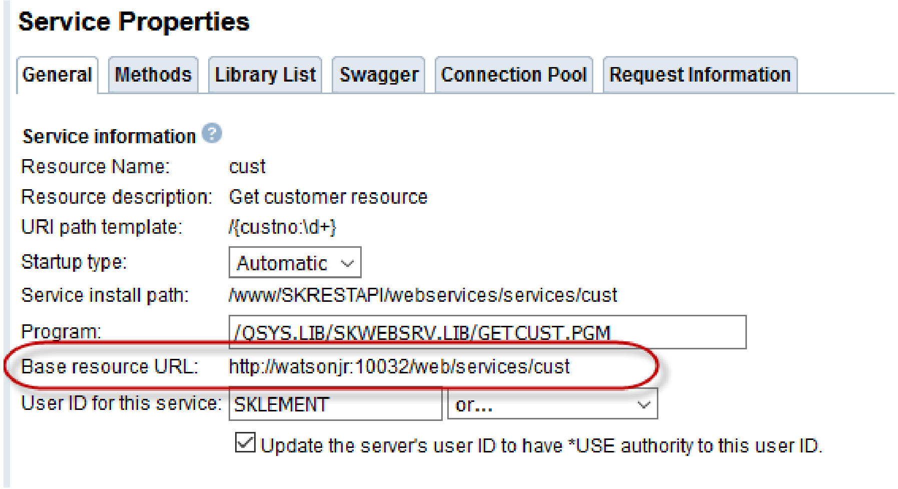
Data current as of Dec 10, 2020 3:15:36 AM.

Deployed services:

	Service name	Status	Type	Startup type	Service definition
○	ConvertTemp	● Running	SOAP	Automatic	View WSDL
●	cust	● Running	REST	Automatic	View Swagger

Looking Up the URI of Your API (2 of 2)

The base resource URL is the URI (base resource name) of the API you created. It does not contain any of the variable parts of the URI such as customer number, however.



Service Properties

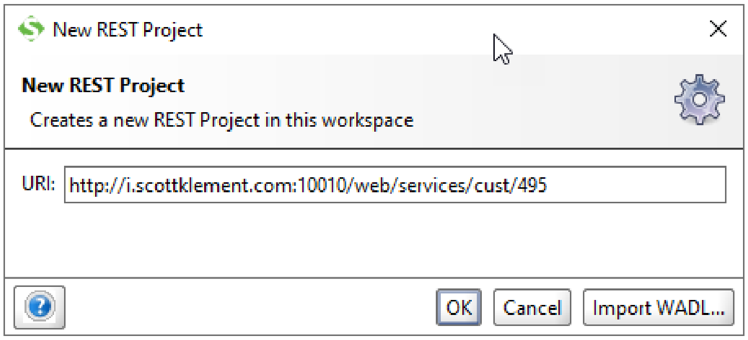
Service information

Resource Name: cust
Resource description: Get customer resource
URI path template: /{custno:\d+}
Startup type: Automatic
Service install path: /www/SKRESTAPI/webservices/services/cust
Program: /QSYS.LIB/SKWEBSRV.LIB/GETCUST.PGM
Base resource URL: http://watsonjr:10032/web/services/cust
User ID for this service: SKLEMENT or...
☑ Update the server's user ID to have *USE authority to this user ID.

SOAPUI REST Testing (1 of 2)

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

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



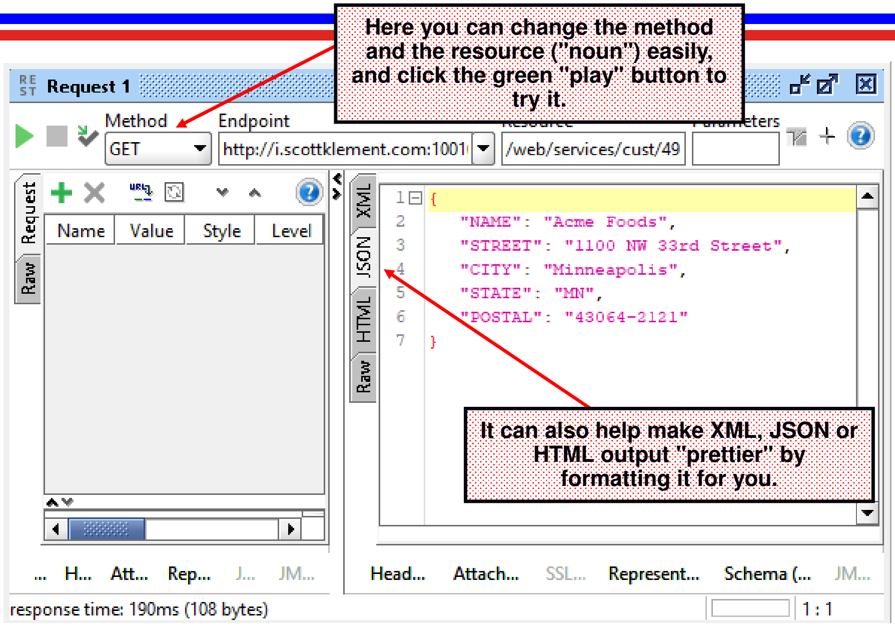
New REST Project

New REST Project

Creates a new REST Project in this workspace

URI: http://i.scottklement.com:10010/web/services/cust/495

SOAPUI REST Testing (2 of 2)



Here you can change the method and the resource ("noun") easily, and click the green "play" button to try it.

Method: GET

Endpoint: http://i.scottklement.com:1001

Resource: /web/services/cust/49

```
{
   "NAME": "Acme Foods",
   "STREET": "1100 NW 33rd Street",
   "CITY": "Minneapolis",
   "STATE": "MN",
   "POSTAL": "43064-2121"
}
```

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

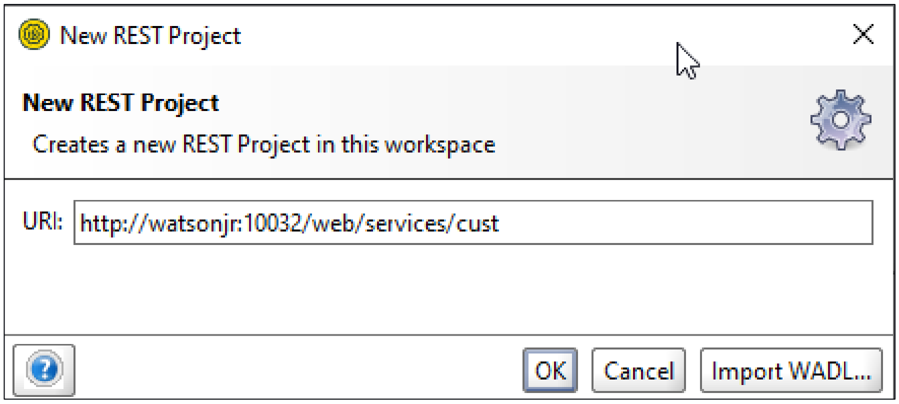
response time: 190ms (108 bytes)

SOAPUI REST Testing (1 of 3)

Once the API has finished creating, you can test it out in SoapUI

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

If you don't know the URL, you can get it (as "Base Resource URL") from the properties of your service in IWS.



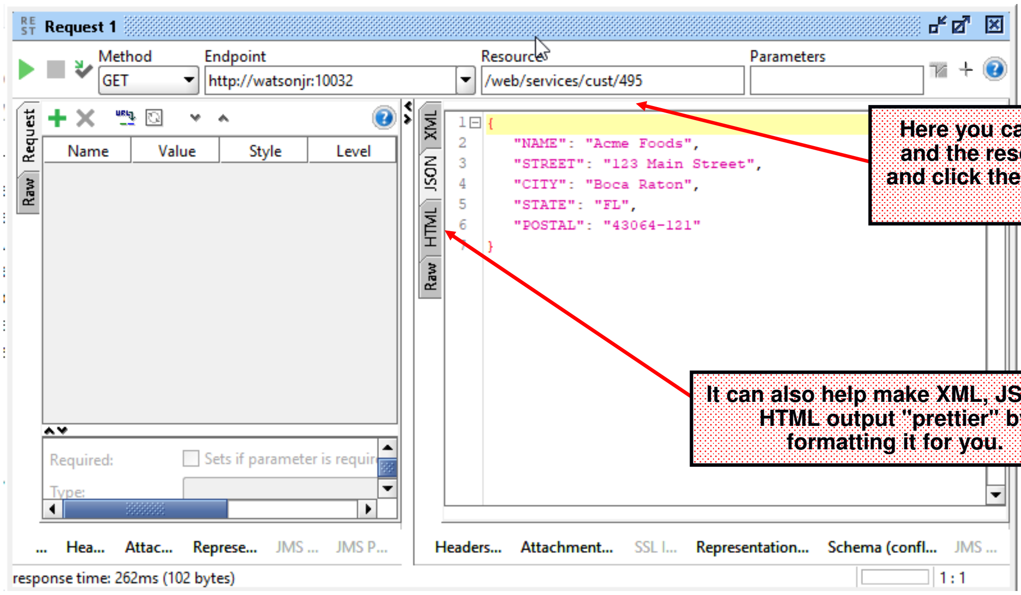
New REST Project

New REST Project

Creates a new REST Project in this workspace

URI: http://watsonjr:10032/web/services/cust

SOAPUI REST Testing (2 of 3)



Request 1

Method: GET

Endpoint: http://watsonjr:10032

Resource: /web/services/cust/495

Parameters:

Name	Value	Style	Level

```
{
   "NAME": "Acme Foods",
   "STREET": "123 Main Street",
   "CITY": "Boca Raton",
   "STATE": "FL",
   "POSTAL": "43064-121"
}
```

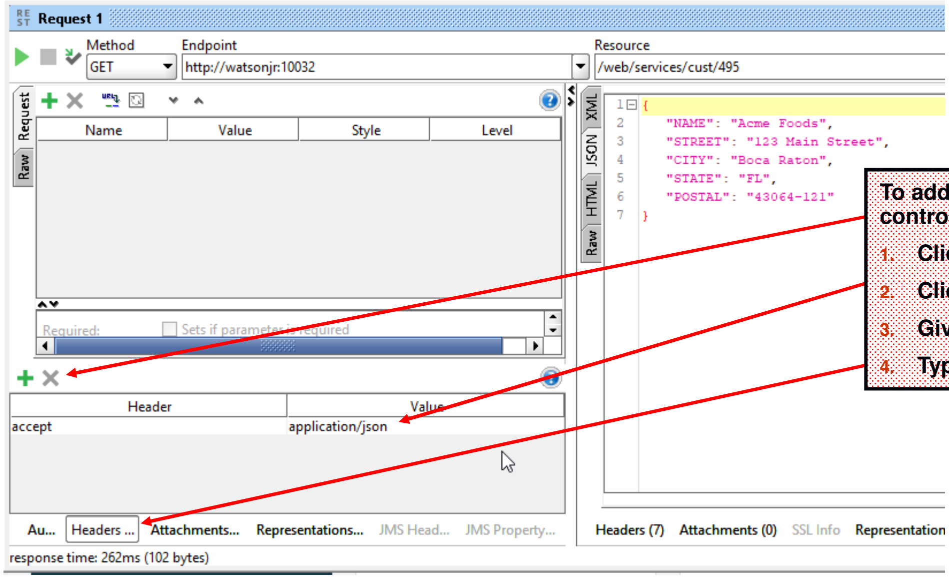
response time: 262ms (102 bytes)

Request
Raw
XML
JSON
HTML
Raw
Required:
Sets if parameter is requir
Type:
... Hea... Attac... Represe... JMS ... JMS P...
Headers... Attachment... SSL I... Representation... Schema (confl... JMS ...
1 : 1

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.

SOAPUI REST Testing (3 of 3)



Request 1

Method: GET

Endpoint: http://watsonjr:10032

Resource: /web/services/cust/495

Name	Value	Style	Level

```
{
   "NAME": "Acme Foods",
   "STREET": "123 Main Street",
   "CITY": "Boca Raton",
   "STATE": "FL",
   "POSTAL": "43064-121"
}
```

Header	Value
accept	application/json

response time: 262ms (102 bytes)

Request
Raw
XML
JSON
HTML
Raw
Required:
Sets if parameter is required
Au... Headers ... Attachments... Representations... JMS Head... JMS Property...
Headers (7) Attachments (0) SSL Info Representation

To add the "accept" header (to control the output document type)

1. Click "Headers" at the bottom
2. Click the green + symbol
3. Give it the name "accept"
4. Type the media type under value

Do It Yourself

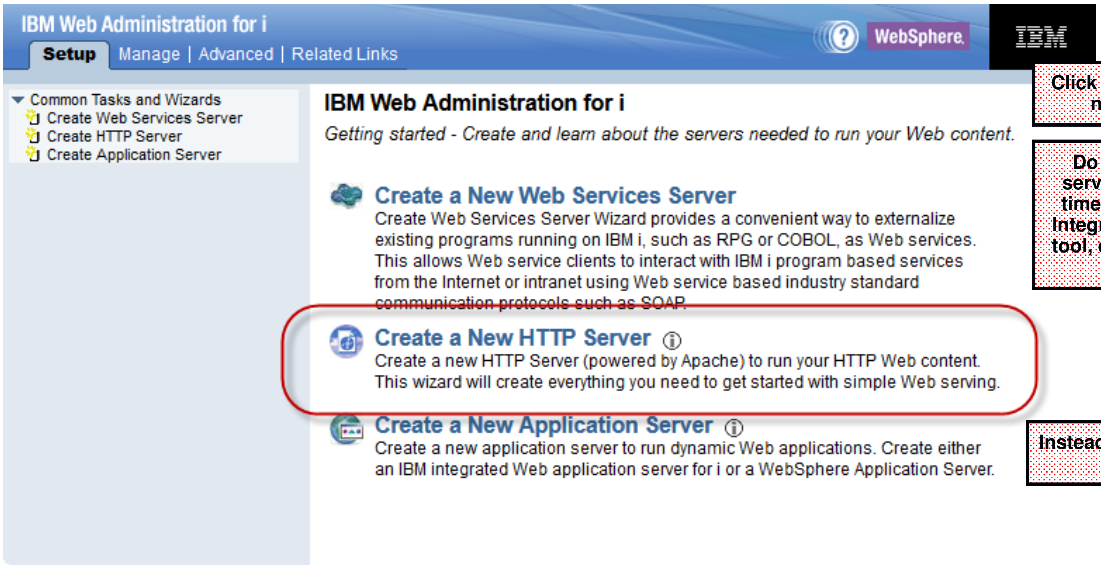
IWS is a neat tool, but:

- 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
- limited authentication types
- limited to only XML or JSON, no other options
- etc.

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


IBM Web Administration for i
Setup | Manage | Advanced | Related Links
WebSphere
IBM
Common Tasks and Wizards
Create Web Services Server
Create HTTP Server
Create Application Server

IBM Web Administration for i

Getting started - Create and learn about the servers needed to run your Web content.

Create a New Web Services Server
Create Web Services Server Wizard provides a convenient way to externalize existing programs running on IBM i, such as RPG or COBOL, as Web services. This allows Web service clients to interact with IBM i program based services from the Internet or intranet using Web service based industry standard communication protocols such as SOAP.

Create a New HTTP Server ⓘ
Create a new HTTP Server (powered by Apache) to run your HTTP Web content. This wizard will create everything you need to get started with simple Web serving.

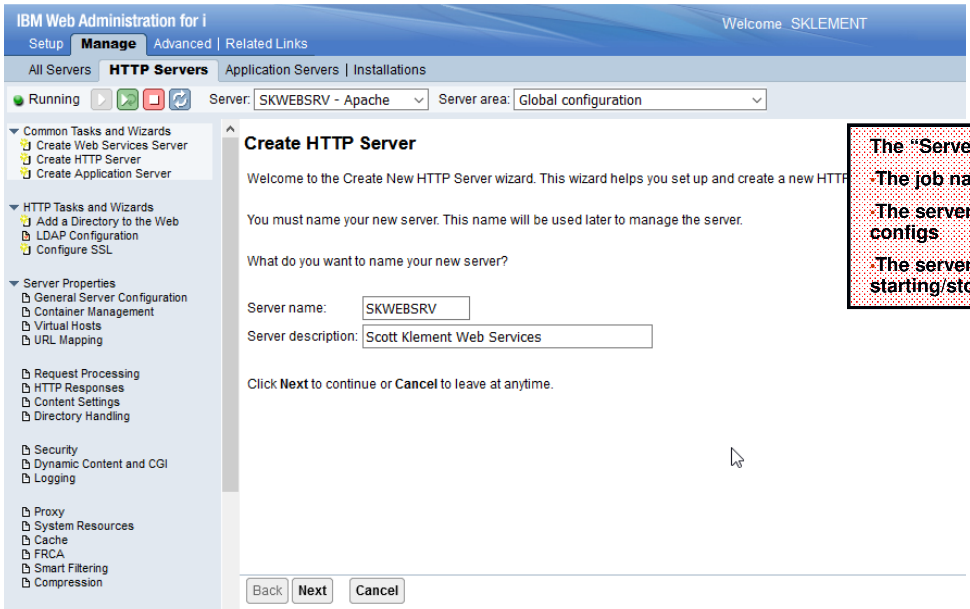
Create a New Application Server ⓘ
Create a new application server to run dynamic Web applications. Create either an IBM integrated Web application server for i or a WebSphere Application Server.

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”



Create HTTP Server

Welcome to the Create New HTTP Server wizard. This wizard helps you set up and create a new HTTP

You must name your new server. This name will be used later to manage the server.

What do you want to name your new server?

Server name: SKWEBSRV

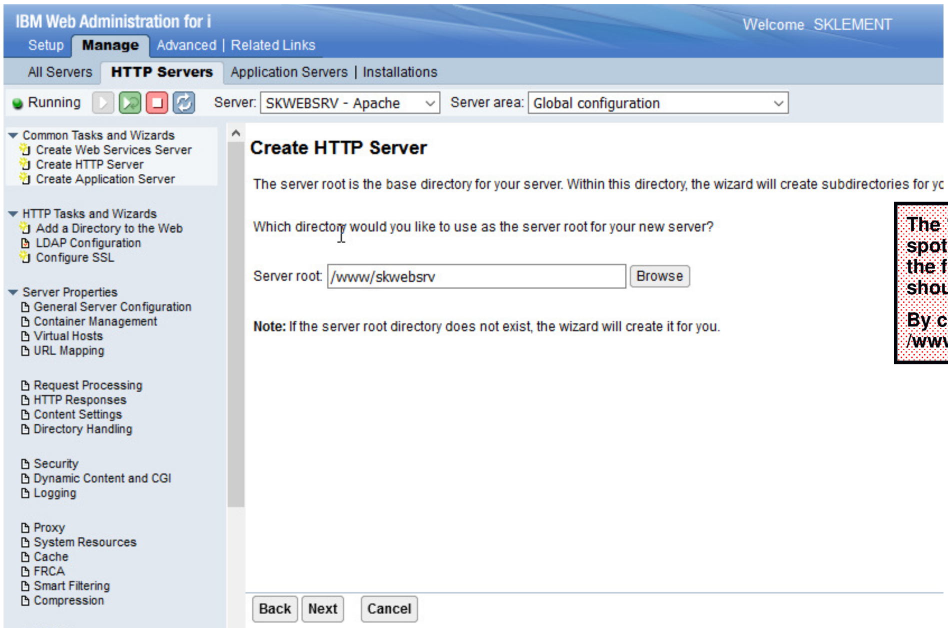
Server description: Scott Klement Web Services

Click **Next** to continue or **Cancel** to leave at anytime.

The “Server Name” controls:

- **The job name of the server jobs**
- **The server name you select when editing configs**
- **The server name you select when starting/stopping the server.**

Server Root



Create HTTP Server

The server root is the base directory for your server. Within this directory, the wizard will create subdirectories for yc

Which directory would you like to use as the server root for your new server?

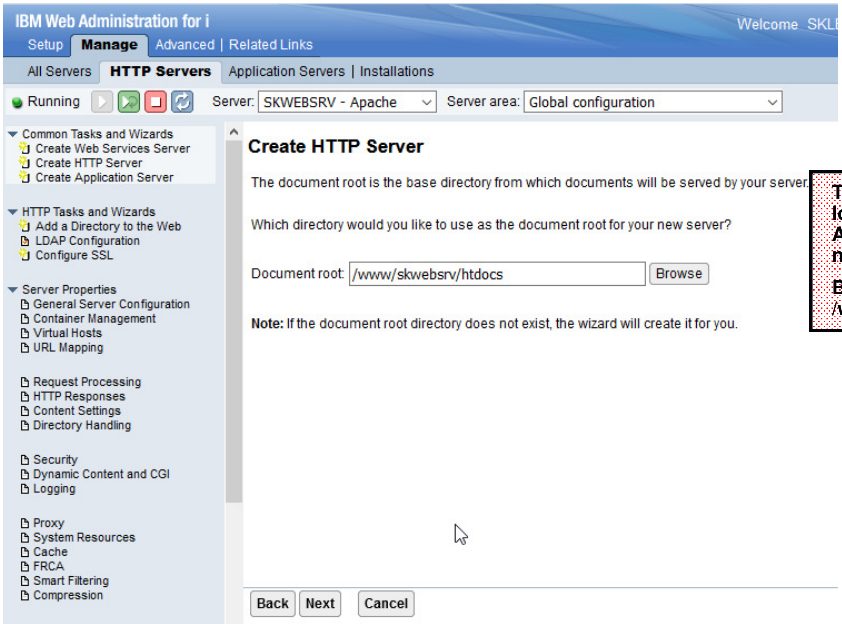
Server root: /www/skwebsrv

Note: If the server root directory does not exist, the wizard will create it for you.

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.

Document Root



Server: SKWEBSRV - Apache   Server area: Global configuration

Create HTTP Server

The document root is the base directory from which documents will be served by your server.

Which directory would you like to use as the document root for your new server?

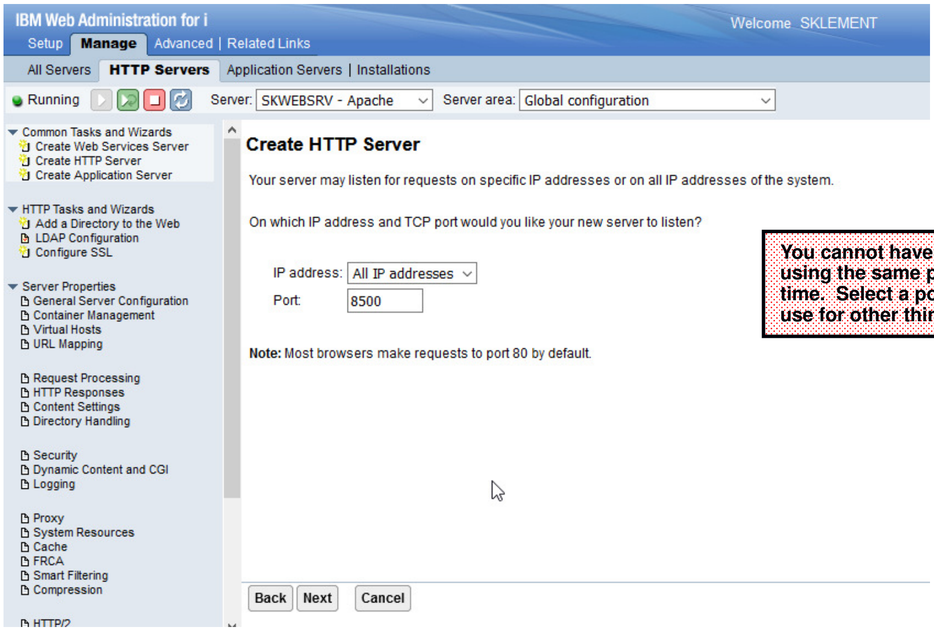
Document root: /www/skwebsrv/htdocs

Note: If the document root directory does not exist, the wizard will create it for you.

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



Server: SKWEBSRV - Apache   Server area: Global configuration

Create HTTP Server

Your server may listen for requests on specific IP addresses or on all IP addresses of the system.

On which IP address and TCP port would you like your new server to listen?

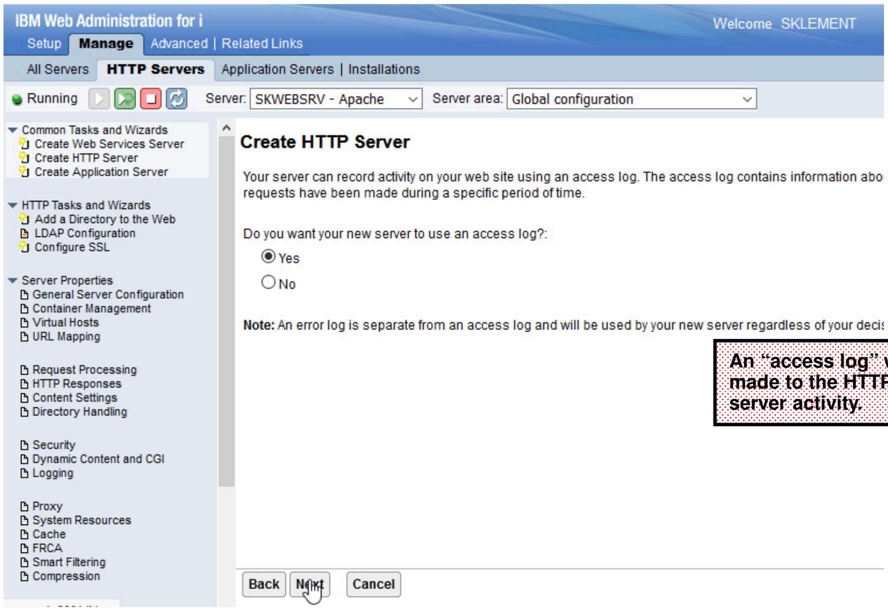
IP address: All IP addresses

Port: 8500

Note: Most browsers make requests to port 80 by default.

You cannot have two different servers using the same port number at the same time. Select a port number that's not in use for other things.

Access Log



Server: SKWEBSRV - Apache Server area: Global configuration

Create HTTP Server

Your server can record activity on your web site using an access log. The access log contains information abo requests have been made during a specific period of time.

Do you want your new server to use an access log?:

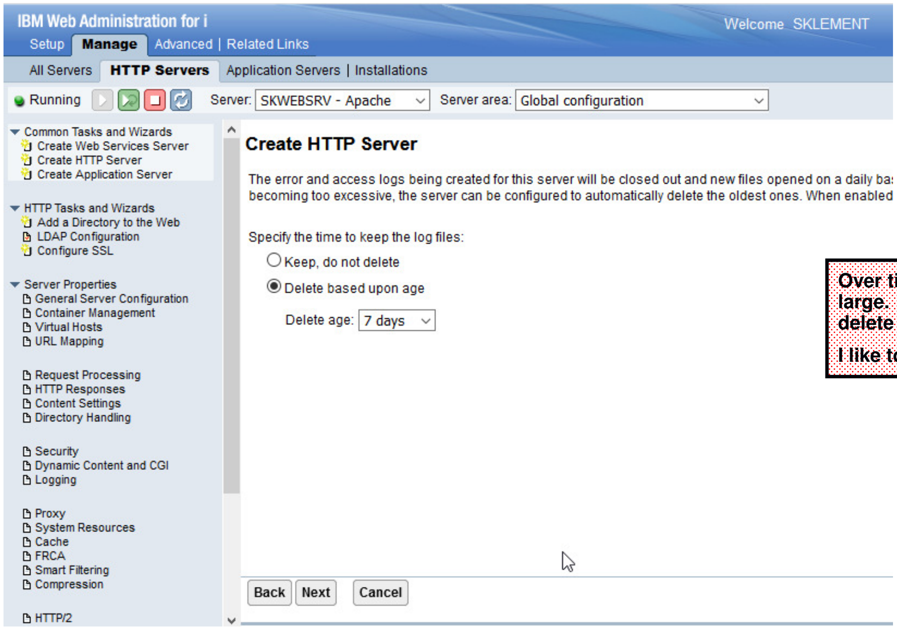
● Yes

○ No

Note: An error log is separate from an access log and will be used by your new server regardless of your decis

An "access log" will log all accesses made to the HTTP server. Useful to track server activity.

Access Log Retension



Server: SKWEBSRV - Apache Server area: Global configuration

Create HTTP Server

The error and access logs being created for this server will be closed out and new files opened on a daily ba: becoming too excessive, the server can be configured to automatically delete the oldest ones. When enabled

Specify the time to keep the log files:

○ Keep, do not delete

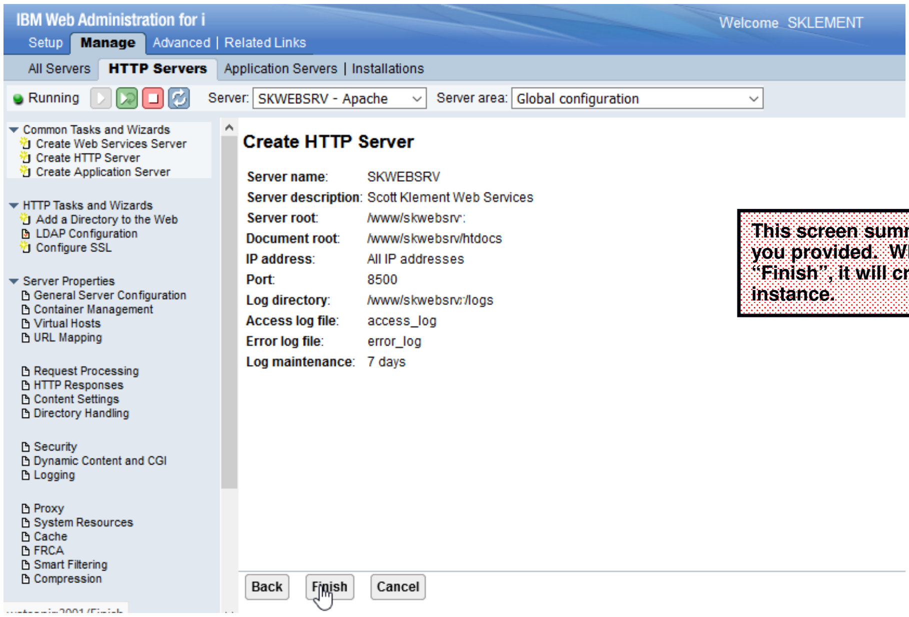
● Delete based upon age

Delete age: 7 days

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



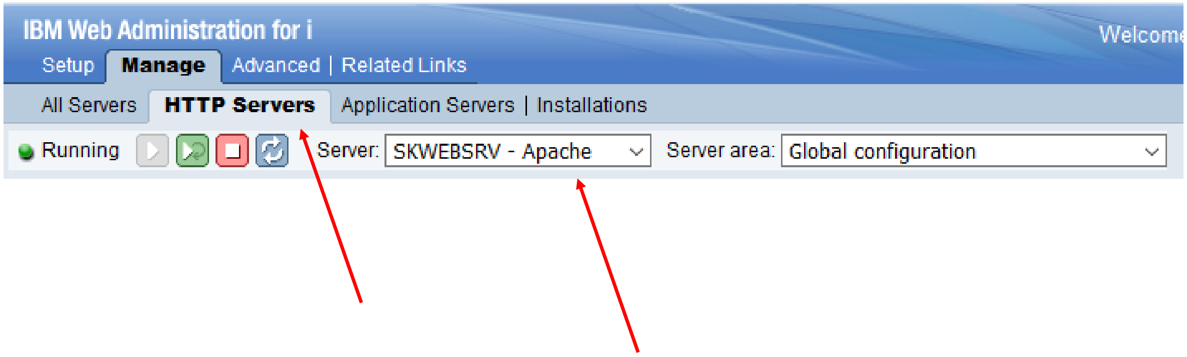
Create HTTP Server

Server name: SKWEBSRV
Server description: Scott Klement Web Services
Server root: /www/skwebsrv
Document root: /www/skwebsrv/htdocs
IP address: All IP addresses
Port: 8500
Log directory: /www/skwebsrv/logs
Access log file: access_log
Error log file: error_log
Log maintenance: 7 days

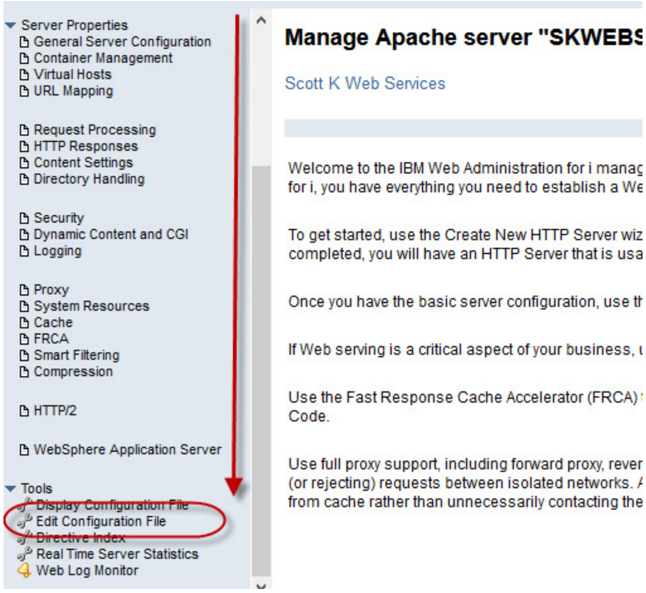
This screen summarizes the settings you provided. When you click "Finish", it will create the server instance.

Getting to the Server

- You should now be on the settings page for your new HTTP server. However, if you navigate away and need to get back you can:
- Return to the *Web Administration for i* page
- Click the HTTP Servers tab
- Select your server from the "Server" drop-down



Edit Configuration File



Manage Apache server "SKWEBS

Scott K Web Services

Welcome to the IBM Web Administration for i manag
for i, you have everything you need to establish a We

To get started, use the Create New HTTP Server wiz
completed, you will have an HTTP Server that is usa

Once you have the basic server configuration, use th

If Web serving is a critical aspect of your business, u

Use the Fast Response Cache Accelerator (FRCA) t
Code.

Use full proxy support, including forward proxy, rever
(or rejecting) requests between isolated networks. A
from cache rather than unnecessarily contacting the

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.

Add Apache Options For Your Server

I recommend adding the following options to your configuration file

These should be customized for your environment and are described on the next slide.

```
DefaultFsCCSID 37
DefaultNetCCSID 1208
CgiConvMode %%MIXED/MIXED%%

ScriptAlias /api/customers /qsys.lib/skwebsrv.lib/cust001r.pgm

<Directory /qsys.lib/skwebsrv.lib>
   SetEnv QIBM_CGI_LIBRARY_LIST "QTEMP;QGPL;SKLEMENT;SKWEBSRV;YAJL"
   require valid-user
   AuthType basic
   AuthName "SK REST APIs"
   PasswdFile %%SYSTEM%%
   UserId %%CLIENT%%
</Directory>
```

Character Set Options

These options control how Apache will translate data between character encodings.

```
DefaultFsCCSID 37
DefaultNetCCSID 1208
CgiConvMode %%MIXED/MIXED%%
```

- DefaultFsCCSID = should be set to your normal EBCDIC CCSID.
  - 37 = The normal EBCDIC for the USA where I live. Replace with the best one for where you live. *Never use 65535*.
  - Jobs will run under this CCSID.
  - This is important if you plan to use the SQL HTTP, JSON or XML functions in your API
- DefaultNetCCSID = should be the CCSID of the data as you want it sent over the network
  - I always recommend UTF-8 (CCSID 1208) for this. UTF-8 is the character set of the web. It is what you should always use when working with XML and JSON documents.
- CgiConvMode = controls what/how Apache translates with the above CCSIDs. I've found %%MIXED/MIXED%% works nicely for APIs.

URI to Object Mapping

Aliases tell Apache how to map from a path in the URI to an object on disk

- Regular `Alias` keyword will download the object from disk
- The `ScriptAlias` keyword denotes that you should run it as a program and download its output rather than downloading the object itself.

```
ScriptAlias /api/customers /qsys.lib/skwebsrv.lib/cust001r.pgm
```

- If URL starts with **`/api/customers`**, Apache will **`CALL PGM(SKWEBSRV/CUST001R)`**

```
http://ibmi.example.com/api/customers/495
```

- Consumer connects to: **`ibmi.example.com`**
- Apache sees the `/api/customers` and calls `SKWEBSRV/CUST001R`
- Our program can read the 495 (customer number) from the URL itself.

Library Options

The <Directory> section specifies options used whenever accessing the given path /qsys.lib/skwebsrv.lib -- i.e. anytime it uses something in the SKWEBSRV library.

```
<Directory /qsys.lib/skwebsrv.lib>
   SetEnv QIBM_CGI_LIBRARY_LIST "QTEMP;QGPL;SKLEMENT;SKWEBSRV;YAJL"
   require valid-user
   AuthType basic
   AuthName "SK REST APIs"
   PasswdFile %%SYSTEM%%
   UserId %%CLIENT%%
</Directory>
```

- QIBM_CGI_LIBRARY_LIST is how we can control the library list when our API is called.
- Require valid-user means that Apache will only allow access for authenticated users
- AuthType specifies the authentication type -- basic is a plaintext userid/password
- AuthName is a string sent to the user to tell him/her what they are signing in to
- PasswdFile %%SYSTEM%% means you will sign on with a standard IBM i user profile and password. It's also possible to set up other methods such as LDAP, Kerberos, or your own file containing users/passwords
- UserId is which user profile the API is run under. %%CLIENT%% means it will use the profile that you signed into the PasswdFile with.

Using RegExp For Program Names

People often ask me how to avoid the need for editing the Apache configuration each time you want to add a new API.

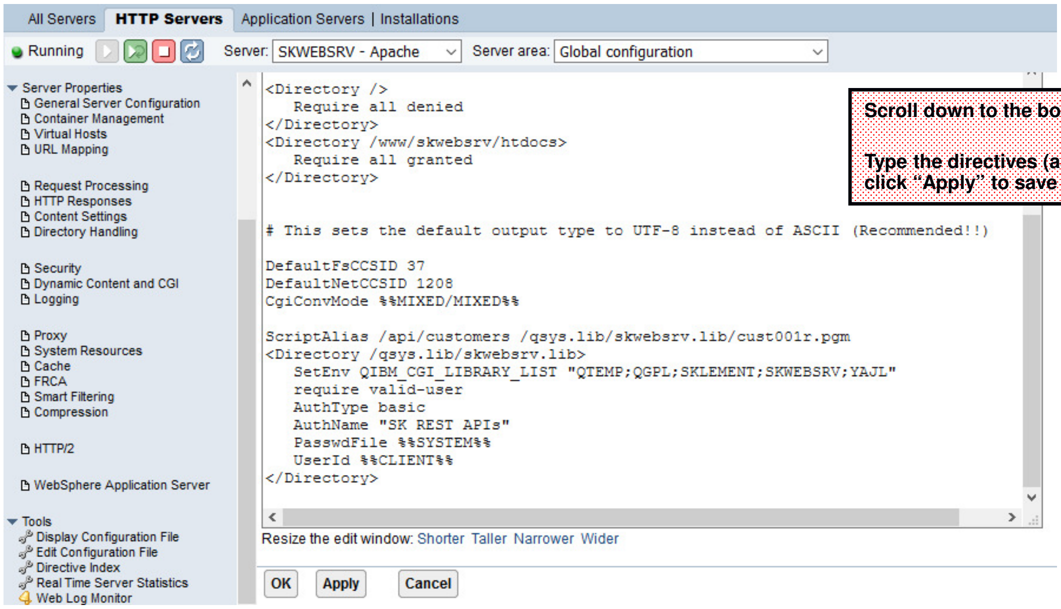
Here's an alternative way to do ScriptAlias that might help.

```
ScriptAliasMatch /api/([a-z0-9]+)/.* /qsys.lib/skwebsrv.lib/$1.pgm
ScriptAliasMatch /api/([a-z0-9]+)$ /qsys.lib/skwebsrv.lib/$1.pgm
```

- ScriptAliasMatch lets you do a ScriptAlias using a regular expression
- ( ) allows a matching string to be stored in a variable. The first parenthesis are stored in variable 1, if there's a second (only one is shown in this example) it'd be stored in variable 2, etc.
- $1 returns the value of variable 1. (use $2 for variable 2, $3 for variable 3, etc.)
- In this example a URI such as `/api/cust001r` would store the string cust001r into variable 1
- Since `$1` is cust001r, it would `CALL SKWEBSRV/CUST001R`
- If the URL contained a different string after /api/ then that would be the program called.

I prefer not to use this method because I like my API names to be friendly like "/api/customers", rather than follow an object naming convention like "/api/cust001r"

Add Custom Directives

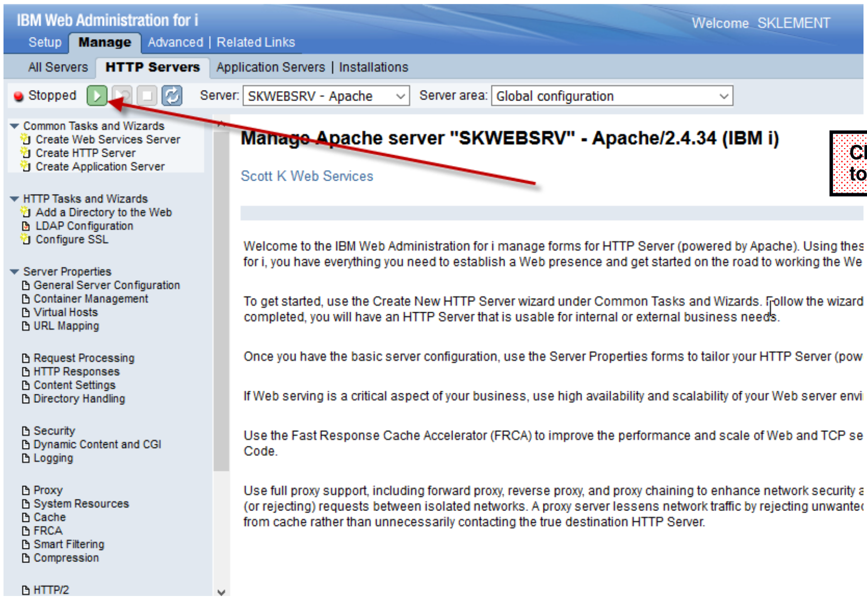


```
<Directory />
   Require all denied
</Directory>
<Directory /www/skwebsrv/htdocs>
   Require all granted
</Directory>

# This sets the default output type to UTF-8 instead of ASCII (Recommended!!)

DefaultFsCCSID 37
DefaultNetCCSID 1208
CgiConvMode %%MIXED/MIXED%%

ScriptAlias /api/customers /qsys.lib/skwebsrv.lib/cust001r.pgm
<Directory /qsys.lib/skwebsrv.lib>
   SetEnv QIBM_CGI_LIBRARY_LIST "QTEMP;QGPL;SKLEMENT;SKWEBSRV;YAJL"
   require valid-user
   AuthType basic
   AuthName "SK REST APIs"
   PasswdFile %%SYSTEM%%
   UserId %%CLIENT%%
</Directory>
```

Scroll down to the bottom of the file.

Type the directives (as shown) and click "Apply" to save your changes.

Start New Apache Server



Manage Apache server "SKWEBSRV" - Apache/2.4.34 (IBM i)

Scott K Web Services

Welcome to the IBM Web Administration for i manage forms for HTTP Server (powered by Apache). Using thes
for i, you have everything you need to establish a Web presence and get started on the road to working the We

To get started, use the Create New HTTP Server wizard under Common Tasks and Wizards. Follow the wizard
completed, you will have an HTTP Server that is usable for internal or external business needs.

Once you have the basic server configuration, use the Server Properties forms to tailor your HTTP Server (pow

If Web serving is a critical aspect of your business, use high availability and scalability of your Web server envi

Use the Fast Response Cache Accelerator (FRCA) to improve the performance and scale of Web and TCP se
Code.

Use full proxy support, including forward proxy, reverse proxy, and proxy chaining to enhance network security a
(or rejecting) requests between isolated networks. A proxy server lessens network traffic by rejecting unwante
from cache rather than unnecessarily contacting the true destination HTTP Server.

Click the green "start" button at the top to start your new server.

You can also start from 5250 with:

`STRTCPSVR *HTTP HTTPSVR(SKWEBSRV)`

CUST001R Example

CUST001R is the provider that we were calling with HTTPAPI earlier. (The "more sophisticated" Customer Maintenance CRUD API.)

- There is quite a lot to it -- it does not make sense to post the entire program here
- Instead, please download the source from my web site
- But, I will go over some of the important highlights in the following slides.

Think about what we need to do!

- Apache will call us
- It will provide the JSON or XML document sent from the consumer via "standard input"
- We can send back a JSON or XML document via "standard output"
- We'll need to know the URI to determine the customer number
- We'll need to know the content-type and accept headers so we know which data format to read and/or send back.

IBM Routines You'll Need

```
ctl-opt option(*srcstmt: *nodebugio: *noshowcpy);

dcl-pr QtmhWrStout extproc(*dclcase);
   DtaVar    pointer value;
   DtaVarLen int(10) const;
   ErrorCode char(32767) options(*varsize);
end-pr;

dcl-pr QtmhRdStin extproc(*dclcase);
   DtaVar     pointer value;
   DtaVarSize int(10) const;
   DtaLen     int(10);
   ErrorCod4  char(32767) options(*varsize);
end-pr;

dcl-pr getenv pointer extproc(*dclcase);
   var pointer value options(*string);
end-pr;
```

These definitions allow you to call IBM-provided subprocedures for

- **QtmhRdStin** reads standard input (message sent to provider)
- **getenv** etrieves an environment variable.
- **QtmhWrStout** writes data to standard output. (message sent back to consumer)

The Qtmh procedures are in service program QHTTPSVR/QZHBCGI, so you will need to bind to that service program when you create your RPG program.

Important Environment Variables

```
env = getenv('REQUEST_METHOD');
if env <> *null;
   method = %str(env);
endif;

env = getenv('REQUEST_URI');
if env <> *null;
   url = %str(env);
endif;

env = getenv('CONTENT_TYPE');
if env <> *null;
   inputType = %str(env);
endif;

env = getenv('HTTP_ACCEPT');
if env <> *null;
   outputType = %str(env);
endif;
```

The getenv() API can be used to retrieve some important information.

- **REQUEST_METHOD** the HTTP method used to call your API
- **REQUEST_URI** the URI used to call your API
- **CONTENT_TYPE** the content-type header (media type of data sent from consumer) **HTTP_ACCEPT** the accept header (media type of data to send back to the consumer)

Extracting the Customer Number from the URI

```
dcl-c REQUIRED_PART const('/api/customers/');

dcl-s pos      int(10);
dcl-s custpart varchar(50);
dcl-s url      varchar(1000);
dcl-s custid   packed(5: 0);

monitor;
   pos = %scan(REQUIRED_PART:url) + %len(REQUIRED_PART);
   custpart = %subst(url: pos);
   custid = %int(custpart);
on-error;
   custid = 0;
endmon;

if custid = 0 and method <> 'GET' and method <> 'POST';
   errMsg = 'You must supply a customer ID!';
   httpstatus = 404;
   // send back error
endif;
```

To extract the customer number from the URI, simply use %SCAN to find the spot after /api/customers, and substring it out.

What Do We Do With All Of This?

I will not show every detail, but consider what we can do with the information we have:

- With the customer number, we can retrieve the existing database record (if any)
- With the HTTP method, we know whether we want to read, update, write or delete the record.
- We can check the content-type for 'application/json' or 'text/xml' to determine if the input data is JSON or XML
- We can check the accept header for 'application/json' or 'text/xml' to determine which data type to send back.

At this point, the program will read the existing database record into the 'cust' data structure. I won't show that logic, since you probably already know how to work with databases in RPG.

Next, we'll need to read the input message (if doing a PUT or POST) and update the database. (I won't show the database logic.)

And we'll need to create output messages containing the customer information and send them back.

Reading a JSON Input Message

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

```
dcl-proc loadInputJson;

   dcl-pi *n ind;
      cust likeds(cust_t);
   end-pi;

   dcl-s loaded ind inz(*off);

   monitor;
      data-into cust %DATA( '*STDIN'
                          : 'case=convert +
                             allowmissing=yes')
                     %PARSER('YAJLINTO');
      loaded = *on;
   on-error;
      httpstatus = 400;
      loaded = *off;
   endmon;

   return loaded;

end-proc;
```

YAJLINTO allows the special value of *STDIN to read the "standard input" (data sent from the consumer).

Writing a JSON Output Message

```
dcl-proc sendResponseJson;

   dcl-pi *n ind;
      cust likeds(cust_t) const;
      httpStatus packed(3: 0) value;
   end-pi;

   dcl-s success ind inz(*on);
   dcl-s responseJson varchar(100000);

   monitor;
      data-gen cust
               %data(responseJson)
               %gen( 'YAJLDTAGEN'
                   : '{ +
                        "write to stdout": true, +
                        "http status": ' + %char(httpstatus) +
                     '}' );
   on-error;
     httpstatus = 500;
     success = *off;
   endmon;

   return success;
end-proc;
```

YAJLDTAGEN provides options:

- write to stdout = automatically send JSON document back to consumer
- http status option = set the HTTP status code

Because of these options provided by YAJLINTO and YAJLDTAGEN, you do not need to manually call the IBM-provided QtmhRdStin and QtmhWrStout procedures if you use YAJL.

What if XML is Required?

The YAJLINTO and YAJLDTAGEN have built-in features for writing APIs that made reading and writing the JSON fairly simple. For the most part, DATA-INTO and DATA-GEN do all of the work!

However, that is not the case when you want to use SQL. For examples of reading and writing XML messages, I will show you the process you need to use when SQL is used to interpret/format the message.

Note that even though this example is for XML -- the same technique could've been used for JSON, too. We'd simply use the JSON_TABLE, JSON_OBJECT, et al functions instead of the XML ones.

Reading an XML Input Message (1 of 3)

```
dcl-proc loadInputXml;

   dcl-pi *n ind;
      cust likeds(cust_t);
   end-pi;

   dcl-s myXml sqltype(CLOB: 100000);
   dcl-s success varchar(5) inz('true');
   dcl-s errMsg  varchar(500);
   dcl-s RcvLen int(10);
   dcl-c MISSING -1;
   dcl-s start  int(10);

   dcl-ds Result qualified;
     custno like(CUSTFILE.custno);
     name   like(CUSTFILE.name);
     street like(CUSTFILE.street);
     city   like(CUSTFILE.city);
     state  like(CUSTFILE.state);
     postal like(CUSTFILE.postal);
   end-ds;
```

```
   dcl-ds Status qualified inz;
     custno int(5);
     name   int(5);
     street int(5);
     city   int(5);
     state  int(5);
     postal int(5);
     NullInds int(5) dim(6) pos(1);
   end-ds;

   dcl-s myXml sqltype(CLOB: 100000);

   QtmhRdStin( %addr(myXml_data)
             : %size(myXml_data)
             : RcvLen
             : ignore );

   myXml_len = RcvLen;
```

To use SQL, I must read standard input myself. By calling QtmhRdStin(). Here it is loaded straight into myXML, which is a CLOB field.

Reading an XML Input Message (2 of 3)

```
<cust success="false" errorMsg="some message here">
   … more data here …
</cust>
```

XMLPARSE interprets a character string representing an XML document and returns a corresponding SQL XML type column.

XMLTABLE converts the XML column into a (virtual) XML table that you can query with a select statement.

- passing specifies the input XML document
- '$doc/cust' is the XPath that determines each row in the output table
- columns specifies the columns in the output table
- Each column listed has a path option with an XPath relative to the row

```
   exec SQL
     select ifnull(success, 'null'), ifnull(errorMsg, '')
       into :success, :errMsg
       from xmltable(
         '$doc/cust'
         passing xmlparse( DOCUMENT :myXml ) as "doc"
         columns
           success  varchar(5)   path '@success',
           errorMsg varchar(500) path '@errorMsg'
       ) as X1;
```

In this case, $doc/cust/@success means

- $doc = the document (from "passing")
- /cust = the <cust> XML tag
- @success = the success attribute within that tag

Reading an XML Input Message (3 of 3)

```
<?xml version="1.0" encoding="UTF-8"?>
<cust success="true" errorMsg="">
   <data custno="495">
      <name>Acme Foods</name>
      <address>
         <street>123 Main Street</street>
         <city>Boca Raton</city>
         <state>FL</state>
         <postal>43064-2121</postal>
      </address>
   </data>
</cust>
```

```
    exec SQL
      select *
        into :Result:Status.NullInds
        from xmltable(
          '$doc/cust/data'
          passing xmlparse( DOCUMENT :myXml ) as "doc"
          columns
            custno decimal(5, 0) path '@custno',
            name   varchar(30)   path 'name',
            street varchar(30)   path 'address/street',
            city   varchar(20)   path 'address/city',
            state  char(2)       path 'address/state',
            postal varchar(10)   path 'address/postal'
        ) as X2;
```

- One row per `/cust/data` tag within the document
- Observe how each column is extracted from within that data tag by its own path.
- If any columns are missing, they will be set to null, so can be checked via the Status data structure.

- As you may be able to see… processing XML with SQL is significantly more complex than reading/writing JSON with DATA-INTO/GEN

Writing the XML Output Message (1 of 2)

```
dcl-s data sqltype(CLOB : 100000);

exec sql
   select
     XMLSERIALIZE(
       XMLELEMENT( name "cust",
         XMLATTRIBUTES(:success as "success",
                       :errMsg  as "errorMsg"),
         XMLAGG(
           XMLELEMENT(name "data",
             XMLATTRIBUTES(T2.custno as "custno"),
             XMLELEMENT(name "name",     trim(T2.name)),
             XMLELEMENT(name "address",
               XMLELEMENT(name "street", trim(T2.street)),
               XMLELEMENT(name "city",   trim(T2.city  )),
               XMLELEMENT(name "state",  trim(T2.state )),
               XMLELEMENT(name "postal", trim(T2.postal))
             )
           )
         )
       )
     AS CLOB(100000) CCSID 1208
     VERSION '1.0' INCLUDING XMLDECLARATION)
   into :data
   from CUSTFILE T2;
```

Writing the list of all customers is somewhat easier because we can use XMLAGG to read directly from the database table (CUSTFILE) and build the whole XML message at once.

Writing the XML Output Message (2 of 2)

```
dcl-s hdr     varchar(500);
dcl-s utfdata varchar(200000) ccsid(*utf8);

if success = 'true';
  hdr = 'Status: 200' + CRLF
      + 'Content-type: application/xml; charset=UTF-8' + CRLF
      + CRLF;
else;
  hdr = 'Status: 500' + CRLF
      + 'Content-type: application/xml; charset=UTF-8' + CRLF
      + CRLF;
endif;

if data_len = 0;
   utfdata = '';
else;
   utfdata = %subst(data_data:1:data_len);
endif;

QtmhWrStout( %addr(hdr:*data)    : %len(hdr)    : ignore );
QtmhWrStout( %addr(utfdata:*data): %len(utfdata): ignore );
```

To write the output, I create a list of HTTP headers separated by CRLF characters.

Sending CRLF on a line by itself means that I'm done with the headers. Everything after that will be considered the document itself.

Notice that I'm telling Apache that my document is already in a UTF-8 character set. I am converting it using RPG's built-in CCSID(*UTF8) support.

The QtmhWrStout() API sends the data back to Apache, who will send it to the consumer.

You can call QtmhWrStout() as many times as you wish, the data will be appended to create a single return document.

This Presentation

You can download a PDF copy of this presentation and its code samples from

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

Thank you!