# Building a REST service with integrated web services server for IBM i: Part 1

## The basics

By Nadir Amra, IBM Software Engineer
Published 08 May 2015 (updated 05 May 2024)

**Abstract:** Rapidly changing application environments require a flexible mechanism to exchange data between different application tiers. Representational State Transfer (REST) has gained widespread acceptance across the Web as the interface of choice for mobile and interactive applications.

You may already be using integrated web services server to expose ILE programs and service programs as SOAP-based web services. This series of articles introduces a powerful new feature of the integrated web services server – the ability to deploy ILE programs and services programs as RESTful web services. In this first installment, you will learn basic REST concepts and how integrated web services server supports REST services.

# Introduction

There is a new style of applications evolving, driven by many forces converging simultaneously. The rapid rise and ubiquity of mobile and social applications are stimulating increased levels of interaction between humans through electronic means. Cloud, mobile and social are also fueling the hyper-growth of API-centric, business as-a-service economies, where data has considerable value and can be monetized given an easy-to-consume API.

Typically, the APIs (or web services) are either SOAP-based or Representational State Transfer (REST) -based web services. However, REST has gained widespread acceptance across the Web as a simpler alternative to SOAP. The key motivator of choosing REST-based web services is its simplicity and its ubiquity:

- It's about delivering content in the simplest possible way
- HTTP is available everywhere; it's like the air around us

For several years now IBM i users have had the ability to deploy ILE programs and services programs as web services based on the SOAP protocol using the integrated web services server support that is part of the operating system. REST web services was not supported by the integrated web services server, until now.

This article is the first in a series of articles about the integrated web services server REST support. This first part starts out by explaining the basic concepts behind REST web services and how the integrated web services server supports REST services. Future installments in this series will build upon the basic concepts:

- Part two will take you through the steps of deploying a simple ILE application as a RESTful web service.
- Part three will take you through the steps of deploying a more complex ILE application that uses more of the REST features.

# Prerequisites

## *Software*

To get all the PTFs required by the integrated web services server in support of REST, you will need to load the latest HTTP Group PTF. The IBM Support web page *IBM i Group PTFs with level* lists the HTTP group PTFs for each of the supported releases of the IBM i operating system.

## *Assumptions*

Before reading this article, you should be familiar with the fundamental concepts of JavaScript Object Notation (JSON) and XML.

# The basics of RESTful web services

A REST API (also called a RESTful API or RESTful web API) is an application programming interface (API) that conforms to the design principles of the

*representational state transfer* (REST) architectural style, where you design web services that focus on a system's resources, including how resource states are addressed and transferred over HTTP by a wide range of clients written in different languages and support for a variety of data formats.

In its purest form, a concrete implementation of a REST web service follows the following basic design principles:

- **Uniform interface**: All API requests for the same resource should look the same, no matter where the request comes from. The REST API should ensure that the same piece of data, such as the name or email address of a user, belongs to only one uniform resource identifier (URI). The URI should be exposed as directory structure-like:
    - Use nouns, not verbs (`/accounts/{id}` not `/getaccount?id=nn}`
    - Predictable
- **Client-server decoupling**: In REST API design, client and server applications must be completely independent of each other. The only information that the client application should know is the URI of the requested resource; it can't interact with the server application in any other ways. Similarly, a server application shouldn't modify the client application other than passing it to the requested data via HTTP.
- **Statelessness**: REST APIs are stateless, meaning that each request needs to include all the information necessary for processing it. In other words, REST APIs do not require any server-side sessions. Server applications aren't allowed to store any data related to a client request.
- **Use HTTP methods**: REST APIs communicate through HTTP requests to perform standard functions like creating, reading, updating and deleting records (also known as CRUD) within a resource:
    - To create a resource on the server, use POST.
    - To retrieve a resource, use GET.
    - To change the state of a resource or to update it, use PUT.
    - To remove or delete a resource, use DELETE.

An example makes this clearer. Assume that you have an application designed for tracking software defects and you want to enable easy reuse and manipulation of the data in the application database. You'd expose a URL endpoint — let's call it `/defects` — and then allow developers to access this endpoint using different HTTP methods and content. Based on the HTTP method and content, it's possible to deduce the operation being requested and take appropriate action on the data. Here are some examples:

- `GET /defects`: list all bugs.
- `GET /defects/123`: Retrieve bug #123.
- `POST /defects/123`: Create a new defect #123 with the POST request body.
- `PUT /defects/123`: Update defect #123 with the PUT request body.
- `DELETE /defects/123`: delete defect #123

The state of a resource at any instant, or timestamp, is known as the resource representation. This information can be delivered to a client in virtually any format including JavaScript Object Notation (JSON), HTML, XLT, Python, PHP or plain text. JSON is popular because it's readable by both humans and machines—and it is programming language-agnostic.

Request headers and parameters are also important in REST API calls because they include important identifier information such as metadata, authorizations, uniform resource identifiers (URIs), caching, cookies and more. Request headers and response headers, along with conventional HTTP status codes, are used within well-designed REST APIs.

# The basics of integrated web services support of REST

At this point we should have a basic idea of what RESTful services are all about. Now let us discuss some new terminology and how it applies to the deployment of a RESTful web service. You will encounter the concepts discussed in this section as you deploy a RESTful web service.

## *The root resource*

In the context of integrated web services server, the *root resource* is the ILE program object (i.e. ILE program or service program) that will be deployed as a RESTful web service.

## *Resource methods*

*Resource methods* are methods in a root resource that are bound to HTTP methods. In the context of the integrated web services server, resource methods correspond to the main entry point in an ILE program, or exported procedures in an ILE service program.

The integrated web services server supports the binding of a resource method to any of the following HTTP methods: GET, PUT, POST and DELETE.

In addition to binding a resource method to an HTTP method, you will also need to indicate what content type the resource method will accept from a client. The integrated web services support allows *ALL (content-type of "`*/*`"), *XML (content type of "`application/xml`", *JSON (content type of "`application/json`") and *XML_AND_JSON (content type of "`application/xml,application/json`").

If a client request comes in with an HTTP method that cannot be handled since a resource method was not found to handle the HTTP method, or the content type of the HTTP request cannot be consumed by a resource method, an error HTTP status code will be returned to the client.

You will also need to indicate the content type of the data the resource method produces. Allowed values are *XML, *JSON, and *XML_AND_JSON. At this point you may be wondering how a resource method can produce both XML and JSON? When you specify

*XML_AND_JSON, what you are indicating is that the content type of the response is dependent on what the client is willing to accept via HTTP content negotiation. If a resource method specifies a content type that the client will not accept, the client will get an error HTTP response code.

## Resource method input parameters

For each resource method, you will need to decide on the following:

- Whether input parameters should be wrapped or unwrapped.
- If input parameters are unwrapped, whether you want to inject a value in the parameters from various input sources.

When you deploy an ILE program object, you are given the opportunity to indicate which parameters are input parameters and which parameters are output parameters. Depending on the data type of the parameters, you may then have to decide whether the input parameters should be wrapped or unwrapped.

If you indicate that input parameters are to be wrapped, then all the parameters are encapsulated within an outer structure. Thus, client HTTP requests to a REST service residing in the integrated web services server must have a payload in the request, and the content type of the request must be XML (content type of `application/xml`) or JSON (content type of `application/json`). If the content type is not JSON or XML, the client will get an HTTP error response.

If you indicate that the parameters are to be unwrapped, this means that the input parameters will not be wrapped within an outer structure. It also means that you must inject all supported primitive parameters (parameters that are of type `char`, `int`, `float`, `packed`, `zoned`, and `byte`) with values from any of the following input sources:

- Variables in the URI path template (more on this later)
- Form variables (`<input type="text" name="**lastname**">`)
- Query string variables ( `/cars/new?**color**=blue` )
- Matrix parameters ( `/cars;**color**=blue/new` )
- HTTP headers
- HTTP cookie variables

Parameters cannot be unwrapped under the following conditions:

- If there is more than one parameter that is a structure
- There are arrays in the parameter list
- The type of the parameter is primitive but is not eligible to be unwrapped

## Resource method output parameters

All output parameters will be wrapped in an outer structure. The exception to this is if the output parameter is designated as one that will contain HTTP headers or the HTTP response (i.e. status) code.

The integrated web services REST support allows you to designate an output parameter that is an array of type `char` to be a parameter that will contain HTTP headers. The HTTP headers will be returned to the client as an HTTP header in the response (not in the message body). Each element in the array must be in the following format:

```
<header-name>: <header-value>
```

In the REST world, one of the use cases for setting HTTP headers is to set HTTP headers relating to HTTP caching.

The integrated web services REST support also allows you to designate an output parameter that is of type `int` to be a parameter that will contain the HTTP response code that is to be returned to the client. As with HTTP headers, the parameter will not be returned in the message body.

Depending on the status code, the following actions will be taken:

- If the category of the status code is successful (200 range), or redirection (300 range), and the status code value is not 204 (no content), then the message body will be processed and returned to the client.
- If the category of the status code is client error (400 range) or server error (500 range) or if the status code value is 204, no message body will be returned to the client.

The supported status code values are listed in Table 2. If a status code that is not recognized is encountered, the client will receive a 500 (internal server error) status code.

### Table 2. Supported HTTP status codes

| | | |
|---|---|---|
| 200 OK | 307 Temporary Redirect | 410 Gone |
| 201 Created | 400 Bad Request | 412 Precondition Failed |
| 202 Accepted | 401 Unauthorized | 415 Unsupported Media Type |
| 204 No Content | 403 Forbidden | 500 Internal Server Error |
| 301 Moved Permanently | 404 Not Found | 503 Service Unavailable |
| 303 See Other | 406 Not Acceptable | |
| 304 Not Modified | 409 Conflict | |

## *REST URLs and URI path templates*

URLs are used to specify the location of a resource. Interaction between the server and client is based on issuing HTTP operations to URLs. Defining URL patterns is important

because URLs often have a long lifetime so that clients can directly address a resource long after the resource is initially discovered.

The web services engine will pass HTTP requests to the root resource based on the incoming URL. The URL has the following form:

```
http://<host>:<port>/<context-root>/<root-resource>/<uri-path-template>
```

When an integrated web services server is created, the default context root for all services deployed in the integrated web services server is `/web/services`. The context root for the server may be changed by modifying the server's properties.

The root resource is the name you choose for your RESTful web application.

The *URI path template* is a partial URI that further qualifies how requests are mapped to resources and resource methods. URI path templates may contain variables embedded within the URI syntax. These variables are substituted at runtime in order for a resource to respond to a request based on the substituted URI. Variables are denoted by braces (`{` and `}`). For example, if the URI path template `/defects/{id}` is associated with a root resource, then an HTTP request with the URI of `/defects/123` would match the URI path template and route the request to the resource.

By default, the URI variable must match the regular expression "`[^/]+?`" (one or more characters up to the first forward slash). You may choose to customize the regular expression by specifying a different regular expression after the variable name. For example, if we wanted the `id` to be numeric, you can override the default regular expression in the variable definition by specifying: `/defects/{id: \d+}`. If a URL came in from a client that contained a non-numeric value for the last segment of the path, an error HTTP status code will be returned to the client.

As indicated, URI path templates may be specified at the root resource level. URI path templates may also be specified for resource methods. Such methods are often referred to as *sub-resource* methods. This enables you to differentiate resource methods and group together those that are common without having to create many root resources (i.e. deploy more RESTful services). For example, assume that we have an ILE service program named CONVERTTEMP that has two procedures, FTOC and CTOF. If we associate the URI path template of "`/ctof`" with procedure CTOF, then an HTTP request to `/CONVERTTEMP` will be mapped to the procedure FTOC, but an HTTP request for `/CONVERTTEMP/ctof` will be mapped to the procedure CTOF.

## *The format of structured messages*

You may be wondering how one would figure out what the format of an XML or JSON message body should look like for a client request to be successful. The format of the message is given by the Swagger document, which is the REST equivalent of the SOAP WSDL document. But one can deduce what format of the XML or JSON message should be by understanding how integrated web services REST support handles input and output parameters.

In the following discussion, assume that we have an exported procedure named CONVERTTEMP that has one input parameter named TEMPIN of type `char`. It also has one output parameter named TEMPOUT of type `char`.

### The format of XML messages

Only one input parameter can accept a message content of XML and it must be a structure.

For an input parameter to accept a message body that is XML, the format of the XML document that needs to be sent by the client is dependent on whether the input parameters have been wrapped or unwrapped.
For the CONVERTTEMP procedure, if we chose to unwrap the parameters, then the resource method will have an input parameter that is of a primitive type, and thus it can never accept an XML document as input. On the other hand, if we chose to wrap the input parameters, then a wrapper structure named CONVERTTEMPInput is generated that will contain all the input parameters. In our case there is only one, and that is TEMPIN. Thus, the XML document that a client would need to send would have the following format:

#### Identifier Generation

Identifiers for XML element names and JSON field names are generated based on the procedure name, the structure name and the field names within a structure. The first character in the identifier is changed from uppercase to lowercase unless the first two characters are in uppercase, in which case the identifier is left alone.

A new RPG enhancement has been released for IBM i releases 7.1 and 7.2 that allows you to control the identifier case of parameter names. See the following PTFs for details:

SI55531 7.2
SI55442 7.2
SI55340 7.1

### Listing 1. Sample format of XML input - wrapped

```
<CONVERTTEMPInput>
   <TEMPIN>23</TEMPIN>
</CONVERTTEMPInput>
```

Suppose CONVERTTEMP has two parameters, TEMPIN of type `char`, and a structure, STRUCT1, that has two fields of type `int` named FLD1 and FLD2. If we choose to wrap the parameters, then as before the CONVERTTEMPInput wrapper structure will get generated with all the input parameters, and the format of XML document would look like the following:

### Listing 2. Sample format of XML input – wrapped with structure

```
<CONVERTTEMPInput>
   <TEMPIN>23</TEMPIN>
   <STRUCT1>
      <FLD1>00</FLD1>
```

```
      <FLD2>01</FLD2>
   </STRUCT1>
</CONVERTTEMPInput>
```

If we chose to unwrap the input parameters, then the TEMPIN parameter must be injected with a value from a source input (e.g. a cookie or path variable) and the incoming XML document would have the following format:

### Listing 3. Sample format of XML input – unwrapped with structure

```
<STRUCT1>
   <FLD1>00</FLD1>
   <FLD2>01</FLD2>
</STRUCT1>
```

For output parameters, a wrapper structure is always generated, and in our example the XML document that would be returned is as follows:

### Listing 4. Sample format of XML output

```
<CONVERTTEMPResult>
   <TEMPOUT>23</TEMPOUT>
</CONVERTTEMPResult>
```

### The format of JSON messages

Similar to XML, only one input parameter can accept a message content of JSON and it must be a structure.

For an input parameter to accept a message body that is JSON, the format of the JSON document that needs to be sent by the client is dependent on whether the input parameters have been wrapped or unwrapped.

For the CONVERTTEMP procedure, if we chose to unwrap the parameters, then the resource method will have an input parameter that is of a primitive type, and thus it can never accept JSON data as input. On the other hand, if we chose to wrap the input parameters, then a wrapper structure named CONVERTTEMPInput is generated that will contain all the input parameters. In our case there is only one, and that is TEMPIN. Thus, the JSON document that a client would need to send would have the following format:

### Listing 5. Sample format of JSON input - wrapped

```
{"TEMPIN": "2337"}
```

Notice that the wrapper structure name is not included in the JSON data.

Now suppose CONVERTTEMP has two parameters, TEMPIN of type `char`, and a structure, STRUCT1, that has two fields of type `int` named FLD1 and FLD2. If we choose to wrap the parameters, then as before the CONVERTTEMPInput wrapper structure will get generated with all the input parameters, and the format of the JSON document would look like the following:

### Listing 6. Sample format of input JSON data – with structure

```
{
   "TEMPIN": "23",
   "STRUCT1": {
      "FLD1": "00",
      "FLD2": "01"
   }
}
```

If we chose to unwrap the input parameters, then the TEMPIN parameter must be injected with a value from a source input (e.g. a cookie or path variable) and the incoming JSON document would have the following format:

### Listing 7. Sample format of JSON input – unwrapped with structure

```
{
   "FLD1": "00",
   "FLD2": "01"
}
```

For output parameters, a wrapper structure is always generated, and in our example the JSON document that would be returned is as follows:

### Listing 8. Sample format of JSON output

```
{"TEMPOUT": "1280.55"}
```

## Summary

This article described the basic concepts of REST and the integrated web services server REST support. For several years now the integrated web services server allowed you to expose ILE programs and service programs as web services based on the SOAP protocol. The unmatched simplicity of exposing assets (e.g. data or services) as SOAP-based web services has now been extended to REST-based web services.

Developing REST APIs is a great way of promoting data sharing and reuse, but more importantly, it enables enterprises to engage customers in new markets. By allowing external developers to access your data via REST, you make it easy for them to build cool new applications and come up with interesting ways to piggyback on your product or data. So, unleash your assets!

Do not forget to look at parts two and three of the series that will take you through the steps of deploying an ILE application as a RESTful web service.

## Resources

- For everything about the integrated Web services support on IBM i see the product web page.
- Parts two and three of the series can be found on the integrated web services web site.