REST Developer's Guide

Version 10.3

October 2018

WEBMETHODS

This document applies to webMethods Integration Server Version 10.3 and to all subsequent releases.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Document ID: IS-RS-DG-103-20181015

Table of Contents

About this Guide....5
Document Conventions....5
Online Information and Support....6
Data Protection....7

About Integration Server REST Processing....9
Overview....10
About REST Request Messages....11
Sending Responses to the REST Client....11
Status Line....11
Header Fields....12
Message Body....12
Setting Responses Using pub.flow:HTTPResponse....13
How webMethods Integration Server Processes REST Requests....13

Configuring a REST Resource Using the Legacy Approach....15
Processing Requests Using Partial Matching of URL Aliases....18

Configuring a REST V2 Resource....21
Considerations for Specifying the URL template in a REST V2 Resource Operation....22
Examples of Configuring REST Resources Using the URL Template-Based Approach....23
Configuring a REST V2 Resource Based on JSON API....25
Examples of Configuring REST Resources Based on JSON API....25

Setting Up a REST Application Using the Legacy REST Approach....39
Setting Up a REST Application on Integration Server....40
Setting Up a REST Application Using the Legacy REST Approach....40
Configuration....42
Converting an Existing Application to a REST Application....42

Setting Up a REST Application Using REST API Descriptor....45
Using REST API Descriptors for Your REST Application....46
Services for REST Resources Configured Using the URL Template-Based Approach....47
Configuration....48
Converting an Existing Application to a REST Application....49

About this Guide

This guide is for developers using webMethods Integration Server to create REST applications. This guide assumes basic knowledge of REST concepts and HTTP request processing and familiarity with Software AG Designer and webMethods Integration Server.

Document Conventions

Convention	Description
Bold	Identifies elements on a screen.
Narrowfont	Identifies service names and locations in the format *folder.subfolder.service* , APIs, Java classes, methods, properties.
Italic	Identifies: Variables for which you must supply values specific to your own situation or environment. New terms the first time they occur in the text. References to other documentation sources.
`Monospace font`	Identifies: Text you must type in. Messages displayed by the system. Program code.
{ }	Indicates a set of choices from which you must choose one. Type only the information inside the curly braces. Do not type the { } symbols.
|	Separates two mutually exclusive choices in a syntax line. Type one of these choices. Do not type the | symbol.
[ ]	Indicates one or more options. Type only the information inside the square brackets. Do not type the [ ] symbols.

Convention	Description
...	Indicates that you can type multiple options of the same type. Type only the information. Do not type the ellipsis (...).

Online Information and Support

Software AG Documentation Website

You can find documentation on the Software AG Documentation website at "http://documentation.softwareag.com". The site requires credentials for Software AG's Product Support site Empower. If you do not have Empower credentials, you must use the TECHcommunity website.

Software AG Empower Product Support Website

If you do not yet have an account for Empower, send an email to "empower@softwareag.com" with your name, company, and company email address and request an account.

Once you have an account, you can open Support Incidents online via the eService section of Empower at "https://empower.softwareag.com/".

You can find product information on the Software AG Empower Product Support website at "https://empower.softwareag.com".

To submit feature/enhancement requests, get information about product availability, and download products, go to "Products".

To get information about fixes and to read early warnings, technical papers, and knowledge base articles, go to the "Knowledge Center".

If you have any questions, you can find a local or toll-free number for your country in our Global Support Contact Directory at "https://empower.softwareag.com/public_directory.asp" and give us a call.

Software AG TECHcommunity

You can find documentation and other technical information on the Software AG TECHcommunity website at "http://techcommunity.softwareag.com". You can:

- Access product documentation, if you have TECHcommunity credentials. If you do not, you will need to register and specify "Documentation" as an area of interest.
- Access articles, code samples, demos, and tutorials.
- Use the online discussion forums, moderated by Software AG professionals, to ask questions, discuss best practices, and learn how other customers are using Software AG technology.
- Link to external websites that discuss open standards and web technology.

Data Protection

Software AG products provide functionality with respect to processing of personal data according to the EU General Data Protection Regulation (GDPR). Where applicable, appropriate steps are documented in the respective administration documentation.

1 About Integration Server REST Processing

- Overview 10
- About REST Request Messages 11
- Sending Responses to the REST Client 11
- How webMethods Integration Server Processes REST Requests 13

Overview

Representational State Transfer (REST) is an architectural style used to build distributed hypermedia systems. The World Wide Web is the best known example of such a system.

The focus of REST is on resources rather than services. A resource is a representation of an object or information. A resource can represent:

- A single entity, like a coffee pot you want to purchase from an online shopping site.
- A collection of entities, like records from a database.
- Dynamic information, like real-time status updates from a monitoring site.

That is, resources are the entities or collections of entities in a distributed system that you want to post or retrieve or take action on. In a REST style system, each resource is identified by a universal resource identifier (URI).

Development of REST systems is defined by a series of constraints:

- Clients and servers are separate.
- Communication between clients and servers is stateless.
- Clients can cache responses returned from servers.
- There may be intermediate layers between the client and server.
- Servers can supply code for the clients to execute.
- Clients and servers remain loosely coupled by communicating through a uniform interface.

The uniform interface is the key constraint that differentiates REST from other architectural approaches. The characteristics of this interface are:

- Requests identify resources.
- Responses contain representations of those resources.
- Clients manipulate resources through their representations.
- Messages are self-descriptive.
- The interface employs Hypermedia as the engine of application state (HATEOAS), which enables the client to find other resources referenced in the response.

One strength of REST is that it leverages the well understood methods supported by HTTP to describe what actions should be taken on a resource. To be REST-compliant, an application must support the HTTP GET, POST, PUT, PATCH, and DELETE methods. Many applications use web browsers to interact with resources on the Internet. Web browsers, however, typically support only the HTTP GET and HTTP POST methods. To get around this restriction, you can use Integration Server to build REST-compliant applications that support all five methods.

Integration Server can be a REST server or a REST client. When Integration Server acts as a REST server, it hosts an application that you write. The application includes services that you write that instruct Integration Server to process some or all of the HTTP GET, POST, PUT, PATCH, and DELETE methods in request messages against resources. When Integration Server acts as a REST client, it sends specially formatted requests to the REST server.

About REST Request Messages

REST clients send specially formatted requests to your REST application. The format of REST requests is determined by the webMethods Integration Server REST implementation and your specific application, but essentially it conveys the following information, or tokens, to the REST server:

- The HTTP method to execute
- The directive
- The name of the resource

A simple REST request looks like this:

```
METHOD /directive/resource_type/resource_id HTTP/1.1
```

Where...	Is the...
METHOD	HTTP request method.
directive	The type of processing to perform.
resource_type/ resource_id	Resource to act upon.

More complex request messages can contain more explicit information about the resource.

Sending Responses to the REST Client

When Integration Server responds to an HTTP request, the response contains a status line, header fields, and a message body.

Status Line

The status line consists of the HTTP version followed by a numeric status code and a reason phrase. The reason phrase is a brief textual description of the status code.

Integration Server will always set the HTTP version to match the version of the client that issued the request. You cannot change the HTTP version.

You can use the pub.flow:setResponseCode service to set the status code and reason phrase. You can also set the status code and reason phrase of an HTTP request by adding a variable named *$httpResponse* that references the pub.flow:httpResponse document type to the flow service pipeline. For more information on this document type, see the section *pub.flow:HTTPResponse* in the *webMethods Integration Server Built-In Services Reference*. If you do not explicitly set the status code, Integration Server will set it to 200 for successfully completed requests and an appropriate error code for unsuccessful requests.

HTTP/1.1 defines all the legal status codes in Section "http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10". Examine these codes to determine which are appropriate for your application.

Header Fields

You can communicate information about the request and the response through header fields in the HTTP response. Integration Server will generate some header fields, such as Set-Cookie, WWW-Authenticate, Content-Type, Content-Length, and Connection. You can use the pub.flow:setResponseHeader to set Content-Type and other header fields. You can also set the header fields of an HTTP request by adding a variable named *$httpResponse* that references the pub.flow:httpResponse document type to the flow service pipeline. For more information on this document type, see the section *pub.flow:HTTPResponse* in the *webMethods Integration Server Built-In Services Reference*..

HTTP/1.1 defines the header fields that can appear in a response in three sections of RFC 2616: "4.5", "6.2", and "7.1". Examine these codes to determine which are appropriate for your application.

Message Body

The message body usually contains a representation of the requested resource, one or more URLs that satisfy the request, or both. In some cases, the message body should be empty, as specified in "RFC 2616, Section 4.3"

You can use the pub.flow:setResponse service to explicitly set the message body. You can also set the message body of an HTTP request by adding a variable named *$httpResponse* that references the pub.flow:httpResponse document type to the flow service pipeline. For more information on this document type, see the section *pub.flow:HTTPResponse* in the *webMethods Integration Server Built-In Services Reference*.. If you do not explicitly set the message body, the output pipeline of the top-level service will be returned to the client in the message body.

In URL template-based approach, Integration Server returns the output defined in the output of the service as a response to the client.

For more information about how Integration Server builds HTTP responses, see the section *About Content Handlers for HTTP Responses* in the *webMethods Integration Server Administrator's Guide*.

Setting Responses Using pub.flow:HTTPResponse

The pub.flow:HTTPResponse document type helps you to set the response headers. You can add a reference of pub.flow:HTTPResponse document type with the name *$httpResponse* to the pipeline and use this pipeline variable instead of invoking the pub.flow:setResponseCode, pub.flow:setResponseHeader, and pub.flow:setResponse services to set the response headers.

For more information, see the section *pub.flow:setResponse* in the *webMethods Integration Server Built-In Services Reference*.

How webMethods Integration Server Processes REST Requests

When Integration Server processes a REST request, it parses the tokens and identifies the HTTP method to execute, locates the resource to act upon, and passes additional information as input parameters to the services you wrote for your application. The configuration of the REST resources determines how Integration Server handles the requests from REST clients. Integration Server provides the following two approaches for configuring REST resources:

- Legacy approach, in which creating a REST resource includes creating the resource folder and flow services that correspond to supported HTTP methods.
- URL template-based approach, in which a URL template serves as a template for client requests to invoke a REST V2 resource.

The following sections explain the approaches in greater detail.

- "Configuring a REST Resource Using the Legacy Approach" on page 15
- "Configuring a REST V2 Resource" on page 21

2 Configuring a REST Resource Using the Legacy Approach

■ Processing Requests Using Partial Matching of URL Aliases 18

You can use the legacy approach to create a new REST resource that include the REST resource folder and the flow services that correspond to HTTP methods. REST resources generated using the legacy approach are invoked with the `rest` directive. For information about the procedure to configure REST resources, see the *webMethods Service Development Help*.

On Integration Server the resources of your application are represented as folders within a package. For each resource, you will write individual services for the HTTP methods that you want Integration Server to execute against the resource. Those services must be named _get, _post, _put, _patch, and _delete, and they are stored in the folder for the resource. For more information, see "Setting Up a REST Application Using the Legacy REST Approach" on page 40.

Consider a Discussion application that maintains a database of discussions about different topics. The following examples show how Integration Server would parse these REST requests.

Example 1

Here is a request to obtain a list of all topics contained in the database, and how Integration Server parses the request:

```
GET /rest/discussion/topic HTTP/1.1
```

Where...	Is the...
GET	Type of HTTP method to perform. Integration Server maps this value to the corresponding service on Integration Server, in this case, the _get service.
rest	Type of processing to perform, in this case, Integration Server REST processing. **Note:** For more information about directives, see the section *Controlling the Use of Directives* in the *webMethods Integration Server Administrator's Guide*.
discussion/topic	Location of the _get service for this resource on Integration Server. In this example, the _get service resides in the topic folder in the discussion folder (discussion.topic).

Example 2

Here is a request to display information about topic number 3419, and how Integration Server parses the request:

```
GET /rest/discussion/topic/3419 HTTP/1.1
```

Where...	Is...
`3419`	An instance of a resource passed into a service as the *$resourceID* variable. In the example, the *$resourceID* variable narrows the focus of the GET request to topic 3419. **Note:** Integration Server assigns the first token after the folder(s) to the *$resourceID* parameter. To determine whether a token represents a folder or the *$resourceID* , Integration Server looks in the current namespace for a folder that has the same name as the token. If it does not find a folder with this name, Integration Server assigns the token to the *$resourceID* variable. In other words, the first token (after the directive) that does not correspond to a folder becomes the *$resourceID* .

Example 3

Here is a request to display information about a particular comment, 17 for example, and how Integration Server parses the request:

```
GET /rest/discussion/topic/3419/comment/17 HTTP/1.1
```

Where...	Is...
`comment/17`	Additional information that further narrows the information about the resource. This information is passed into a service as the *$path* variable. In the example, `comment/17` further narrows the focus of the GET request to comment 17.

Example 4

Here is a request to display information contributed by participant Robertson in 2009 about topic 17, and how Integration Server parses the request:

```
GET /rest/discussion/topic/3419/comment/17?year=2009&name=Robertson
HTTP/1.1
```

Where...	Are...
`year` and `name`	Input variables that are specific to your application. Tokens specified after the ? must be entered as name/value pairs. In this example, `year=2009` and `name=Robertson` narrow the focus of the GET request to entries that participant Robertson added to comment 17 in 2009.

Processing Requests Using Partial Matching of URL Aliases

REST URL requests usually include the identifier for a particular resource. However, because the identifier varies for each instance of a resource, REST requests often do not exactly match any of the defined URL aliases for a particular resource. To enable you to define URL aliases for REST resources, Integration Server can use partial matching to process REST requests. A *partial match* occurs when a REST request includes only part of a URL alias. For more information about URL aliases, see the section *Creating an HTTP URL Alias* in the *webMethods Integration Server Administrator's Guide*.

Note: You can configure URL aliases *only* for REST resources configured using the legacy approach.

When partial matching is enabled and Integration Server receives a REST request URL, an alias is considered a match if the entire alias matches all or part of the request URL, starting with the first character of the request URL's path.

For example, assume the following URL aliases are defined:

URL Alias	URL Path
a1	rest/purchasing/order
a2	rest/purchasing/invoice
a22	rest/purchasing/admin
a3	invoke/pub.flow/debugLog

When partial matching is enabled, the following request URLs would get different results:

- A request URL of http://MyHost:5555/a1 matches URL alias a1 exactly. The resulting URL is http://MyHost:5555/rest/purchasing/order.
- A request URL of http://MyHost:5555/a2/75909 matches alias a2 because the request URL's path begins with "a2". The trailing characters of the request URL are retained and the resulting URL is http://MyHost:5555/rest/purchasing/invoice/75909.
- A request URL of http://MyHost:5555/a1/75909/customer/0122?terms=net7 matches alias a1 because the request URL's path begins with "a1". The trailing characters of the request URL are retained and the resulting URL is http://MyHost:5555/rest/purchasing/order/75909/customer/0122?terms=net7.

In some cases, a partial match can result in an invalid request. For example, a request URL of http://host:5555/a3456 matches alias a3 because the request URL's path begins with "a3". The trailing characters of the request URL are retained and the

resulting URL is http://host:5555/invoke/pub.flow/debugLog456. Since there is no pub.flow:debugLog456 service, this would be an invalid request.

For instructions on enabling partial matching, see the section *Enabling Partial Matching of URL Aliases* in the *webMethods Integration Server Administrator's Guide*.

3 Configuring a REST V2 Resource

- Considerations for Specifying the URL template in a REST V2 Resource Operation 22
- Examples of Configuring REST Resources Using the URL Template-Based Approach 23
- Configuring a REST V2 Resource Based on JSON API 25
- Examples of Configuring REST Resources Based on JSON API 25

You can use the URL template-based approach to configure REST resources. In this approach, you define a URL template for client requests to use and invoke the resources.

REST resources configured using this approach are also known as *REST V2 resources*.

For each REST V2 resource, you must define operations that include the following:

- The format of the URL that REST clients must follow when sending requests to Integration Server acting as the REST server. Integration Server attempts to match a request URL received from any application against the URL template defined for a REST V2 resource operation and determines whether the request URL is valid.
- The HTTP methods supported by the resource operation.
- The flow service associated with a resource operation. You can either associate an existing service with a resource operation or create a new service and associate it with the resource operation.

The URL template-based approach provides you with greater flexibility than the legacy approach in defining REST resources. For a REST V2 resource, you can define multiple operations and associate each operation with a URL template, HTTP methods, and a flow service. In addition, you can edit these details based on your requirements.

Important: You *cannot* configure REST V2 resources when Integration Server is deployed in a multitenanted environment.

Considerations for Specifying the URL template in a REST V2 Resource Operation

Consider the following while defining the URL template in a REST V2 resource operation:

- A URL template definition is a combination of static path segments and dynamic parameters. For example, in the URL template `/restv2/cust:customerNode/customer/{id}/order/{orderID}`, `customer` and `order` are static path segment while `{id}` and `{orderID}` are dynamic.
- Enclose dynamic parameters in the URL template within braces (`{}`). For example, in the URL template `/restv2/customer/{id}`, the `{id}` parameter is dynamic and represents an attribute of the customer resource.
- Any dynamic parameter that you specify in a URL template must be available as a variable of type String in the input signature of the flow service associated with the resource operation. If you specify the option of creating a new flow service when defining the resource operation, a new service with the specified name is automatically created with the dynamic parameter in the URL template added to the input signature.

  Note: For information about creating REST V2 resources and defining resource operations, see *webMethods Service Development Help*.

- While a URL template definition can include multiple dynamic parameters, each dynamic parameter can appear only once in the URL template.
- A URL template cannot include the following characters: & ; ? @ # | [ ]
- Query parameters are not supported in the definition of a URL template. However, the request URL from the client application to Integration Server can include query parameters at run time.
- Ensure that the resource operations in a REST V2 resource are unique, which is a combination of URL template and the HTTP method.
- You can define a logical representation of a resource that is associated with the REST V2 resource.
- You can define attributes for the REST V2 resource.
- You can define relationships between REST V2 resources so that using a single resource all related resources can be accessed.

Important: To invoke REST APIs, Software AG recommends that you create REST API descriptors.

Examples of Configuring REST Resources Using the URL Template-Based Approach

Consider the Discussion application described earlier in "Configuring a REST Resource Using the Legacy Approach" on page 15. Using the URL template-based approach, you can create a REST V2 resource named `discussion` in a REST resource element called `app` under the `discussionNode` folder and define resource operations. The following examples show resource operations for the created resource and how Integration Server parses these requests:

Example 1

Consider a REST V2 resource operation configured with the following URL template:

```
/restv2/app:discussionNode/discussion/topic/{id}
```

Here is an example request to display information about a specific topic:

```
GET /restv2/app:discussionNode/discussion/topic/236 HTTP/1.1
```

Where...	Is the...
GET	HTTP method supported by the resource operation. **Note:** Integration Server treats this method as valid only if the resource and the underlying service are configured to support the GET method. For more information about

Where...	Is the...
	configuring supported HTTP methods for services, see *webMethods Service Development Help*.
restv2/ app:discussionNode	Template prefix. This informs the type of processing to perform, in this case, Integration Server REST processing, and fully qualified name of the REST v2 element. **Note:** For more information about directives, see the section *Controlling the Use of Directives* in the *webMethods Integration Server Administrator's Guide*.
discussion	Name of the resource on Integration Server.
topic	Name of the static part in the URL template.
`236`	Identifier for a topic. Integration Server matches this value against the dynamic parameter `{id}` specified in the URL template. **Note:** The `id` parameter must be available as a variable of type String in the input signature of the flow service associated with the resource operation for which you are defining the URL template. For more information about configuring a resource operation for a REST V2 resource, see *webMethods Service Development Help*.

Example 2

Consider a REST V2 resource operation configured with the following URL template:

```
/restv2/app:discussionNode/discussion/topic/t-{id}
```

Here is a request to display information about a topic based on its identifier and how Integration Server parses the request:

```
GET /restv2/app:discussionNode/discussion/topic/t-1591 HTTP/1.1
```

Where...	Is the...
`t-1591`	Identifier for a topic. In the URL template specified for this example, `t` is a static parameter while `{id}` is a dynamic parameter. Integration Server matches the value `t-1591` against the topic identifier parameters specified in the URL template (`t-{id}`). **Note:** In this example, Integration Server treats `t-1591` as a valid value considering the URL template specified for

Where...	Is the...
	the REST V2 resource operation. However, an identifier that does not follow the specified format, for example, `236` would be considered invalid.

Example 3

Consider a REST V2 resource operation configured with the following URL template:

```
/restv2/app:discussionNode/discussion/topic/t-{id}/comment/{cid}
```

Here is a request to display information about a particular comment related to a topic, and how Integration Server parses the request:

```
GET /restv2/app:discussionNode/discussion/topic/t-1591/comment/4
HTTP/1.1
```

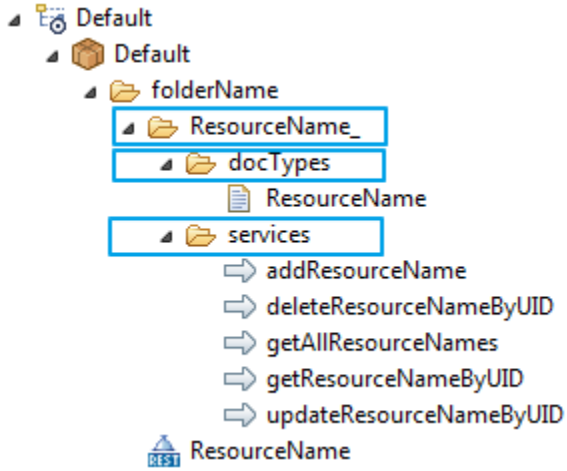
Where...	Is the...
`comment/4`	Additional information for the topic with the identifier `t-1591`. Integration Server matches this value with the portion of the request URL after the topic identifier. The value `4` is matched against the dynamic parameter `{cid}`.

Configuring a REST V2 Resource Based on JSON API

You can configure a REST V2 resource based on JSON API to enhance the usage capabilities of the resource. In this approach, you can define a JSON API based REST V2 resource and JSON API specific URL templates are automatically generated. You can use these URLs to invoke and use the resources. For more details on JSON API specification, see "http://jsonapi.org/format/".

Examples of Configuring REST Resources Based on JSON API

You can create a REST V2 resource and make the resource as JSON API compliant. Integration Server automatically creates all the services, document types, and JSON API based URL templates for the REST V2 resource along with the applicable input and output signatures.. The following figure provides an overview,



- ResourceName_: Contains the relevant documents under docTypes folder and services under services folder for the JSON API based REST V2 resource.
- docTypes: Contains all the automatically generated document types corresponding to the resource.
- services: Contains automatically generated services for the resource. Each service corresponds to the automatically created URL templates.

This following example creates two REST V2 resources as *Article* and *Author* and both the resources are JSON API compliant. You can create the attributes for *Article* ensuring that one attribute, for example, *id* is set as unique identifier. Integration Server generates the REST URL templates based on this unique identifier. For more information about creating a JSON API compliant REST V2 resource and adding new attributes, see *webMethods Service Development Help*.

Integration Server generates the following URL templates according to the JSON API specifications. You can check these URLs and modify them under the **Resource Configuration** tab. For example,

```
/Article GET

/Article/{id} GET

/Article POST

/Article/{id} DELETE

/Article/{id} PATCH
```

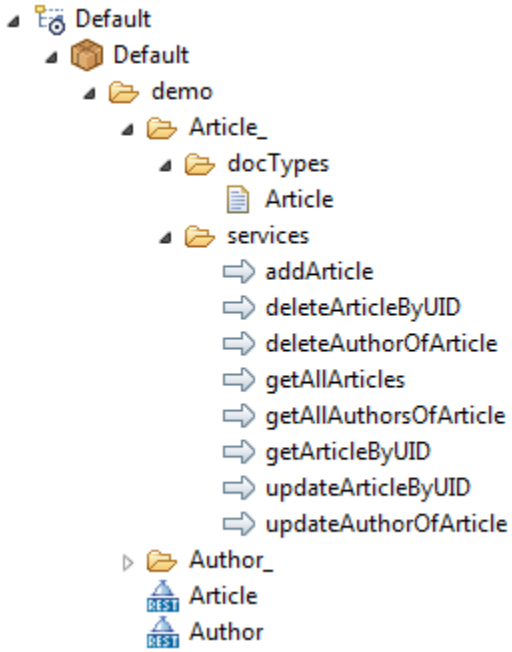
You can assign a relationship between *Article* and *Author* by setting the attribute type as `Resource`. Integration Server generates the following relationship URL templates according to the JSON API specification:

```
/Article/{id}/Author GET

/Article/{id}/Author PATCH

/Article/{id}/Author DELETE
```

Once you create the assets, the package viewer might have the following folder structure as viewed from Software AG Designer.



For more information on defining REST V2 resource attributes and assigning relationships, see *Defining a REST V2 Resource Attribute* topic under *webMethods Service Development Help*.

For more information about checking these REST V2 URLs and modifying them, see *webMethods Service Development Help*.

Request and Response Formats for JSON API Resource

This section lists sample request and response formats for two JSON API resources, *Article* and *Author* .

Get all *Article* records

- Request format: `http://host:port/rad/nameSapce/Article` with `Accept=application/vnd.api+json`
- Response format:

```
{
    "data": [
        {
            "id": "1",
            "type": "Article",
            "attributes": {
                "blogname": "facebook"
            },
            "relationships": {
                "Author": {
                    "data": {
                        "id": "2",
                        "type": "Author"
```

```
                    },
                    "links": {
                        "self": "http://host
                    :port/rad/nameSapce/Article/1/
                      relationships/Author",
                    "related": "http://host:port/
                      rad/nameSapce/Article/1/Author"
                    }
                }
            },
            "links": {
                "self": "http://host:port/rad/nameSapce/Article/1"
            }
        },
        {
            "id": "3",
            "type": "Article",
            "attributes": {
                "blogname": "gmail"
            },
            "relationships": {
                "Author": {
                    "data": {
                        "id": "4",
                        "type": "Author"
                    },
                    "links": {
                        "self": "http://host:port/rad/nameSapce/
                                 Article/3/relationships/Author",
                        "related": "http://host:port/rad/nameSapce/
                                 Article/3/Author"
                    }
                }
            },
            "links": {
                "self": "http://host:port/rad/nameSapce/Article/3"
            }
        }
    ],
    "links": {
        "self": "http://host:port/rad/nameSapce/Article"
    },
    "meta": {
        "total-records": "2"
    }
}
```

Get single *Article* record

- Request format: `http://host:port/rad/nameSapce/Article/1` with `Accept=application/vnd.api+json`
- Response format:

```
{
    "data": {
        "id": "1",
        "type": "Article",
        "attributes": {
            "blogname": "facebook"
        },
        "relationships": {
            "Author": {
```

```
                "data": {
                    "id": "2",
                    "type": "Author"
                },
                "links": {
                     "self": "http://host:port/rad/nameSapce/Article/1/
                              relationships/Author",
                     "related": "http://host:port/rad/nameSapce/Article/1/
                                 Author"
                }
            }
        },
        "links": {
            "self": "http://host:port/rad/nameSapce/Article/1"
        }
    },
    "links": {
        "self": "http://host:port/rad/nameSapce/Article/1"
    },
    "meta": {
        "total-records": "1"
    }
}
```

Post an *Article* record

- Request format: `http://host:port/rad/nameSapce/Article` with `Content-Type=application/vnd.api+json` and `Accept=application/vnd.api+json` and the body as

```
{
    "data": {
        "id": "1",
        "type": "Article",
        "attributes": {
            "blogname": "facebook"
        },
        "relationships": {
            "Author": {
                "data": {
                    "id": "2",
                    "type": "Author"
                }
            }
        }
    }
}
```

- Response format: Response status is 204. If UID is set or any response code is set then the response is,

```
{
    "data": {
        "id": "1",
        "type": "Article",
        "attributes": {
            "blogname": "facebook"
        },
        "relationships": {
            "Author": {
                "data": {
                    "id": "2",
                    "type": "Author"
```

```
                },
                "links": {
                     "self": "http://host:port/rad/nameSapce/Article/1/
                              relationships/Author",
                     "related": "http://host:port/rad/nameSapce/Article/1/
                                 Author"
                }
            }
        },
        "links": {
            "self": "http://host:port/rad/nameSapce/Article/1"
        }
    },
    "links": {
        "self": "http://host:port/rad/nameSapce/Article"
    },
    "meta": {
        "total-records": "1"
    }
}
```

Delete an *Article* record

- Request format: `http://host:port/rad/nameSapce/Article/1` with `Accept=application/vnd.api+json`
- Response format:

```
{
    "data": {
        "id": "1",
        "type": "Article",
        "attributes": {
            "blogname": "facebook"
        },
        "relationships": {
            "Author": {
                "data": {
                    "id": "2",
                    "type": "Author"
                },
                "links": {
                     "self": "http://host:port/rad/nameSapce/Article/1/
                         relationships/Author",
                     "related": "http://host:port/rad/nameSapce/Article/1/
                          Author"
                }
            }
        },
        "links": {
            "self": "http://host:port/rad/nameSapce/Article/1"
        }
    },
    "links": {
        "self": "http://host:port/rad/nameSapce/Article/1"
    },
    "meta": {
        "total-records": "1"
    }
}
```

Patch a record

- Request format: `http://host:port/rad/nameSapce/Article/1` with `Content-Type=application/vnd.api+json` and `Accept=application/vnd.api+json` with body as

```
{
    "data": {
        "id": "1",
        "type": "Article",
        "attributes": {
            "blogname": "facebook",
            "id": "1"
        },
        "relationships": {
            "Author": {
                "data": {
                    "id": "2",
                    "type": "Author"
                }
           }
        }
    }
}
```

- Response format:

```
{
    "data": {
        "id": "1",
        "type": "Article",
        "attributes": {
            "blogname": "facebook"
        },
        "relationships": {
            "Author": {
                "data": {
                    "id": "2",
                    "type": "Author"
                },
                "links": {
                   "self": "http://host:port/rad/nameSapce/Article/1/
                        relationships/Author",
                   "related": "http://host:port/rad/nameSapce/Article/1/
                       Author"
                }
            }
        },
        "links": {
            "self": "http://host:port/rad/nameSapce/Article/1"
        }
    },
    "links": {
        "self": "http://host:port/rad/nameSapce/Article/1"
    },
    "meta": {
        "total-records": "1"
    }
}
```

Get all related data

- Request format: `http://host:port/rad/nameSapce/Article/1/Author`

- Response format:

```
{
    "data": [
        {
            "id": "2",
            "type": "Author",
            "attributes": {
                "name": "social"
            },
            "relationships": {},
            "links": {
                "self": "http://host:port/rad/nameSapce/Author/2"
            }
        }
    ],
    "links": {
        "self": "http://host:port/rad/nameSapce/Article/1/Author"
    },
    "meta": {
        "total-records": "1"
    }
}
```

Update an *Author* record

- Request format: `http://host:port/rad/nameSapce/Article/1/Author` with body as

```
{
    "data":
        {
            "id": "2",
            "type": "Author",
            "attributes": {
                "name": "social"
            }
        }
}
```

- Response format:

```
{
    "data": {
        "id": "2",
        "type": "Author",
        "attributes": {
            "name": "social"
        },
        "relationships": {},
        "links": {
            "self": "http://host:port/rad/nameSapce/Author/2"
        }
    },
    "links": {
        "self": "http://host:port/rad/nameSapce/Article/1/Author"
    },
    "meta": {
        "total-records": "1"
    }
}
```

Delete an *Author* record

- Request format: `http://host:port/rad/nameSapce/Article/1/Author` with `Accept=application/vnd.api+json`
- Response format:

```
{
    "data": {
        "id": "2",
        "type": "Author",
        "attributes": {
            "name": "social"
        },
        "relationships": {},
        "links": {
            "self": "http://host:port/rad/nameSapce/Author/2"
        }
    },
    "links": {
        "self": "http://host:port/rad/nameSapce/Article/1/Author"
    },
    "meta": {
        "total-records": "1"
    }
}
```

Include an *Author* record

- Request format: `http://host:port/rad/nameSapce/Article?include=Author`
- Response format:

```
{
    "data": [
        {
            "id": "1",
            "type": "Article",
            "attributes": {
                "blogname": "facebook"
            },
            "relationships": {
                "Author": {
                    "data": {
                        "id": "2",
                        "type": "Author"
                    },
                    "links": {
                       "self": "http://host:port/rad/nameSapce/Article/1/
                              relationships/Author",
                       "related": "http://host:port/rad/nameSapce/Article/1/
                                Author"
                    }
                }
            },
            "links": {
                "self": "http://host:port/rad/nameSapce/Article/1"
            }
        },
        {
            "id": "3",
            "type": "Article",
            "attributes": {
                "blogname": "gmail"
            },
```

```
            "relationships": {
                "Author": {
                    "data": {
                        "id": "4",
                        "type": "Author"
                    },
                    "links": {
                        "self": "http://host:port/rad/nameSapce/Article/3/
                               relationships/Author",
                        "related": "http://host:port/rad/nameSapce/Article/3/
                                  Author"
                    }
                }
            },
            "links": {
                "self": "http://host:port/rad/nameSapce/Article/3"
            }
        }
    ],
    "included": [
        {
            "id": "2",
            "type": "Author",
            "attributes": {
                "name": "social"
            },
            "relationships": {},
            "links": {
                "self": "http://host:port/rad/nameSapce/Author/2"
            }
        },
        {
            "id": "4",
            "type": "Author",
            "attributes": {
                "name": "mail"
            },
            "relationships": {},
            "links": {
                "self": "http://host:port/rad/nameSapce/Author/4"
            }
        }
    ],
    "links": {
        "self": "http://host:port/rad/nameSapce/Article"
    },
    "meta": {
        "total-records": "2"
    }
}
```

Sparse field sets

- Request format: `http://host:port/rad/nameSapce/Article/1?fields=blogname,id`
- Response format:

```
{
    "data": {
        "id": "1",
        "type": "Article",
        "attributes": {
```

```
            "blogname": "facebook",
            "id": "1"
        },
        "relationships": {
            "Author": {
                "data": {
                    "id": "2",
                    "type": "Author"
                },
                "links": {
                   "self": "http://host:port/rad/nameSapce/Article/1/
                          relationships/Author",
                   "related": "http://host:port/rad/nameSapce/Article/1/
                          Author"
                }
            }
        },
        "links": {
            "self": "http://host:port/rad/nameSapce/Article/1"
        }
    },
    "links": {
        "self": "http://host:port/rad/nameSapce/Article/1"
    },
    "meta": {
        "total-records": "1"
    }
}
```

Sort a record

- Request format: `http://host:port/rad/nameSapce/Article?fields=id,blogname&sort=-id`
- Response format:

```
{
    "data": [
        {
            "id": "3",
            "type": "Article",
            "attributes": {
                "blogname": "gmail",
                "id": "3"
            },
            "relationships": {
                "Author": {
                    "data": {
                        "id": "4",
                        "type": "Author"
                    }
                }
            }
        },
                        {
            "id": "1",
            "type": "Article",
            "attributes": {
                "blogname": "facebook",
                "id": "1"
            },
            "relationships": {
                "Author": {
```

```
                    "data": {
                        "id": "2",
                        "type": "Author"
                    }
                }
            }
        }
    ],
    "links": {
        "self": "http://host:port/rad/nameSapce/Article"
    },
    "meta": {
        "total-records": "2"
    }
}
```

Filter a record

- Request format: `http://host:port/rad/nameSapce/Article?filter[id][EQ]='xyz'`
- Response format: In pipeline $filter value is set so user needs to implement the service according to the filter.

Page number and page limit of a record

- Request format: `http://host:port/rad/nameSapce/Article?page[number]=0&page[limit]=1`
- Response format:

```
{
    "data": [
        {
            "id": "1",
            "type": "Article",
            "attributes": {
                "blogname": "facebook"
            },
            "relationships": {
                "Author": {
                    "data": {
                        "id": "2",
                        "type": "Author"
                    },
                    "links": {
                      "self": "http://host:port/rad/nameSapce/Article/1/
                           relationships/Author",
                      "related": "http://host:port/rad/nameSapce/Article/1/
                         Author"
                    }
                }
            },
            "links": {
                "self": "http://host:port/rad/nameSapce/Article/1"
            }
        }
    ],
    "links": {
        "self": "http://host:port/rad/nameSapce/Article",
        "last": "http://host:port/rad/nameSapce/
                   Article?page[offset]=1&page[limit]=1"
    },
```

```
    "meta": {
        "total-records": "2"
    }
}
```

Processing the Errors

The output signature of a service might have an error structure associated with it. The error object contains multiple entities that provide additional information about problems encountered while performing an operation. An error object contains the following entities:

- id: a unique identifier for this particular occurrence of the problem.
- links: a link object containing the following members:
  - about: a link that leads to further details about this particular occurrence of the problem.
- status: the HTTP status code applicable to this problem, expressed as a string value.
- code: an application-specific error code, expressed as a string value.
- title: a short summary of the problem that should not change from occurrence to occurrence of the problem.
- detail: an explanation specific to this occurrence of the problem.
- source: an object containing references to the source of the error, optionally including any of the following members:
  - pointer: a JSON Pointer to the associated entity in the request document.
  - parameter: a string indicating which URL query parameter caused the error.
- meta: a meta object containing non-standard meta-information about the error.

4 Setting Up a REST Application Using the Legacy REST Approach

- Setting Up a REST Application on Integration Server 40
- Converting an Existing Application to a REST Application 42

Setting Up a REST Application on Integration Server

Integration Server can act as a REST server or REST client. Integration Server acts as a REST server in the following ways based on the approach used:

Setting Up a REST Application Using the Legacy REST Approach

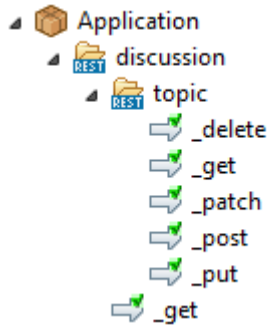
In legacy approach, Integration Server hosts services that perform the GET, PUT, POST, PATCH, and DELETE methods. These services, which you provide, perform processing that is specific to your application.

Services for REST Resources Configured Using the Legacy Approach

When you build a REST application on your Integration Server by configuring resources using the legacy approach, you must include services that correspond to the HTTP methods you want to provide for each resource. These services must be named as follows:

Service	Description
_get	Performs the GET method.
_put	Performs the PUT method.
_post	Performs the POST method.
_patch	Performs the PATCH method.
_delete	Performs the DELETE method.

These services reside in folders on your Integration Server in a directory structure that is specific to your application. For example, the discussion application described in "Configuring a REST Resource Using the Legacy Approach" on page 15 might have the following structure as viewed from Software AG Designer:



In addition to the _get, _put, _post, _patch, and _delete services, you can also place a special service named _default in one or more of the application folders. Integration Server executes this service if a REST request specifies an HTTP method that is not represented by a service in the folder. For example, suppose the folder contains the _get, _put, and _post services, but no _patch or _delete service. If the client issues a DELETE request, Integration Server will execute the _default service, and pass "DELETE" to it in the *$httpMethod* variable.

If a request specifies an HTTP request method that is not represented by a service in the folder and there is no _default service in the folder, the request fails with the "404 Not Found" or "405 Method Not Allowed error." Integration Server issues 404 if the first token in the URI does not exist in the namespace, or 405 if one or more tokens in the URI identify elements in the namespace but the URI does not correctly identify a REST resource folder and a service to execute.

Example 1

A REST resource's folder contains the _get, _post, and _default services:

If the client sends a...	Integration Server responds by...
GET request	Executing the _get service
POST request	Executing the _post service
DELETE request	Executing the _default service

Example 2

A REST resource's folder contains the _get, _put, and _delete services:

If the client sends a...	Integration Server responds by...
GET request	Executing the _get service
PUT request	Executing the _put service
POST request	Issuing error "405 Method Not Allowed"

Additional possible uses for the _default service are:

- Direct all REST requests through common code before branching off to individual GET, PUT, POST, PATCH, or DELETE methods.
- Make PUT and POST processing the same by directing PUT and POST requests to the same code.

Configuration

There are a few things you can configure with respect to REST processing:

- Name of the REST directive

  Note: You can configure the name of the REST directive *only* for resources that use the `rest` directive, that is, the REST resources configured using the legacy approach.

  If you want to allow clients to specify a name other than "rest" for the REST directive, you can do so with the watt.server.RESTDirective configuration parameter. For example, to allow clients to specify "process" for the REST directive, you would change the property to the following:

  ```
  watt.server.RESTDirective=process
  ```

  With this setting, clients can specify "rest" or "process" for the REST directive. In the following example, the two requests are equivalent:

  ```
  METHOD /process/discussion/topic/9876 HTTP/1.1
  ```

  ```
  METHOD /rest/discussion/topic/9876 HTTP/1.1
  ```

  For more information about the watt.server.RESTDirective property, refer to *webMethods Integration Server Administrator's Guide*.

- Which ports will accept the rest directive

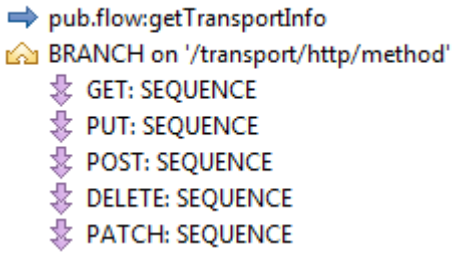
  By default, all Integration Server ports except the proxy port allow use of the rest directive. You can limit which ports will allow this directive by specifying them on the watt.server.allowDirective configuration parameter. For more information about this property, refer to the *webMethods Integration Server Administrator's Guide*.

Converting an Existing Application to a REST Application

When using the the legacy approach, consider one of the following approaches to transform the existing application to a REST application:

- Refactor your existing services into _get, _put, _post, _patch and _delete services.
- Use the invoke directive, as shown in the following example:

  For existing applications that use the invoke directive, you can update a service to call the pub.flow:getTransportInfo service and then perform a branch on `/transport/http/method` to execute the appropriate portions of your existing code, as in the following example:



Note: If you use the invoke directive, you cannot use the *$resourceID* and *$path* pipeline variables. In addition, you cannot use the _default service.

5 Setting Up a REST Application Using REST API Descriptor

- Using REST API Descriptors for Your REST Application 46
- Services for REST Resources Configured Using the URL Template-Based Approach 47
- Configuration 48
- Converting an Existing Application to a REST Application 49

Using REST API Descriptors for Your REST Application

A REST API descriptor provides a way of describing the operations provided by one or more REST resources together with information about how to access those operations, the MIME types the resources consume and produce, and the expected input and output for the operations. Fundamentally, a REST API descriptor is composed of REST resources and information about how to access those resources. Using this information, Integration Server creates and maintains a Swagger document for the REST API descriptor. Integration Server generates the Swagger document based on version 2.0 of the Swagger specification.

Note: You can create REST API descriptors using the legacy approach to expose the legacy REST APIs as Swagger document.

Using Designer you can create a REST API descriptor using resource first or Swagger first approach.

- Resource first refers to REST API descriptors created using the REST v2 resource. For more information about creating a REST API descriptor using the resource first approach, refer to the *webMethods Service Development Help*.

  Note: The base path of the created REST API descriptor is set as `/rad/<namespaceName of the REST API descriptor>` or path in the Swagger document. You can change the base path. To avoid exposing the server namespace name to the end user, set the base path to an application specific path. Also, when you change the base path, you must create a URL alias with the defined base path. For more information about creating a URL alias, refer to the *webMethods Service Development Help*.

- Swagger first refers to REST API descriptors created by importing a Swagger document. During this process, Integration Server creates services, doctypes, and REST V2 resources based on the contents of the Swagger document. The application developer then provides the service implementation of the generated services. You can then invoke the REST application through a client. Because the Swagger specification does not include resource definitions, Integration Server allows you to create the REST V2 resources based on tags or based on the Swagger paths. Tags defined in a Swagger document are used for grouping of operations. If you choose to create a REST V2 resource based on tags, all the operations associated to a tag are grouped together. Otherwise, the resources are created based on the Swagger path. For more information about creating a REST API descriptor using a Swagger document, refer to the *webMethods Service Development Help*.

  Note: In the Swagger first approach to create a REST API descriptor, Integration Server automatically creates a URL alias for the base path of the created REST API descriptor.

You can use REST API descriptors to access a Swagger document through a URL in JSON and YAML format.

The standard URLs to access the Swagger document are:

- JSON Format: *http://host:port/<base_path>?swagger.json*
- YAML Format: *http://host:port/<base_Path>?swagger.yaml*

The following table lists the standard URLs under appropriate conditions:

If the URL includes..	URL in JSON format is..	URL in YAML format is..
A default base path.	*http://host:port/rad/<namespace_of_rad>?swagger.json* . For example, *http://host:port/rad/cc:rad?swagger.json* .	*http://host:port/rad/<namespace_of_rad>>?swagger.yaml* . For example, *http://host:port/rad/cc:rad?swagger.yaml* .
A application specific base path.	*http://host:port/<alias_name>?swagger.json* . For example, *http://host:port/api?swagger.json* .	*http://host:port/<alias_name>?swagger.yaml* . For example, *http://host:port/api?swagger.yaml* .

The server response type for a JSON based application is, *application/json* and for a YAML based application is, *application/x-yaml* .

You can also access the Swagger document for REST API descriptor that is created using the legacy REST resources using the above mentioned URLs.

Note: If the REST API Descriptor does not exist in the Integration Server namespace or if the format is not *swagger.json* or *swagger.yaml* , Integration Server returns an error message. For example, if the URL is *http://host:port/api?swagger.xml* , then the Integration Server throws an error.

Services for REST Resources Configured Using the URL Template-Based Approach

The URL template-based approach helps you configure REST resources for an existing Integration Server service. The HTTP methods that you can configure for a REST resource are restricted only by the methods that you configure as allowed for the underlying service. The methods supported by a REST resource must be a subset of the methods allowed for the service corresponding to the REST resource. For information about configuring the supported methods for a REST resource and its corresponding Integration Server service, see the *webMethods Service Development Help*.

If a REST request specifies an HTTP method that is not allowed for its service, the request fails with a "405 Method Not Allowed error.

Example 1

A REST service and its corresponding resource support the GET, PUT, and DELETE services:

If the client sends a...	Integration Server responds by...
GET request	Executing the GET method
PUT request	Executing the PUT method
POST request	Issuing error "405 Method Not Allowed"

Note: This example assumes that the request URL is in a format supported by the REST resource.

Configuration

There are a few things you can configure with respect to REST V2 processing:

- Name of the REST V2 directive

  If you want to allow clients to specify a name other than "restv2" for the REST V2 directive, you can do so with the watt.server.RESTDirective.V2 configuration parameter. For example, to allow clients to specify "process" for the REST V2 directive, you would change the property to the following:

  ```
  watt.server.RESTDirective.V2=process
  ```

  With this setting, clients can specify "restv2" or "process" for the REST directive. In the following example, the two requests are equivalent:

  ```
  METHOD /process/discussion/topic/9876 HTTP/1.1
  ```

  ```
  METHOD /restv2/discussion/topic/9876 HTTP/1.1
  ```

  For more information about the watt.server.RESTDirective.V2 property, refer to *webMethods Integration Server Administrator's Guide*.

- Which ports will accept the restv2 directive

  By default, all Integration Server ports except the proxy port allow use of the rest directive. You can limit which ports will allow this directive by specifying them on the watt.server.allowDirective.V2 configuration parameter. For more information about this property, refer to the *webMethods Integration Server Administrator's Guide*.

Converting an Existing Application to a REST Application

If you have an existing application that you want to transform into a REST application, consider using the URL template-based approach and configure REST resources for the application. This is the most straightforward approach you can use to transform the application.