REST Developer's Guide

Version 10.1

October 2017

WEBMETHODS

This document applies to webMethods Integration Server Version 10.1 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-101-20171017

Table of Contents

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 storage locations for services on webMethods Integration Server, using the convention *folder.subfolder:service* .
UPPERCASE	Identifies keyboard keys. Keys you must press simultaneously are joined with a plus sign (+).
Italic	Identifies variables for which you must supply values specific to your own situation or environment. Identifies new terms the first time they occur in the text.
`Monospace font`	Identifies text you must type or messages displayed by the system.
{ }	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.
...	Indicates that you can type multiple options of the same type. Type only the information. Do not type the ellipsis (...).

Online Information

Software AG Documentation Website

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

Software AG Empower Product Support Website

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.

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.

1 About Integration Server REST Processing

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.

2 How REST Processing Works

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.

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 format 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

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 "Services" on page 24.

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 *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

Where...	Are...
	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 *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 *webMethods Integration Server Administrator's Guide*.

Configuring a REST Resource Using the URL Template-Based Approach

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

REST resources configured using this approach, also known as *REST V2 resources*, are invoked with the `restv2` directive. For each 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 format 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 format, 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.
- You *cannot* migrate the REST resources created using the URL template-based approach in version 10.0 of Integration Server to a later version.

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

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

- A URL format definition can either include only static parameters or a combination of both and static and dynamic parameters. The definition cannot include only

dynamic parameters. For example, in the URL format `/restv2/customer/{id}/order/{orderID}`, the parameters `customer` and `order` are static while `{id}` and `{eventID}` are dynamic.

- Enclose dynamic parameters in the URL format within braces (`{}`). For example, in the URL format `/restv2/customer/{id}`, the `{id}` parameter is dynamic and represents an attribute of the customer resource.
- Two or more dynamic parameters cannot be combined to form a separate parameter. For example, a parameter `{topic}-{id}` cannot be formed as a combination of `{topic}` and `{id}`.
- Any dynamic parameter that you specify in a URL format 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 format 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 format definition can include multiple dynamic parameters, each dynamic parameter can appear only once in the URL format.
- A URL format cannot include the following characters: & ; = ? @ # | [ ]
- Query parameters are not supported in the definition of a URL format. However, the request URL from the client application to Integration Server can include query parameters at run time.
- Ensure that multiple resource operations for a REST V2 resource do not include similarly defined URL formats and are associated with the same flow service. This is because when a client application sends a request URL that matches a defined URL format, Integration Server cannot invoke the required resource because of the availability of multiple resource operations defined in a similar manner. In such a situation, Integration Server issues an error message when saving the REST V2 resource.

  For example, a client request `GET /restv2/customer/23` issued to an Integration Server will not be able to invoke the correct resource operation if the Integration Server has two operations defined as follows:

  Resource Operation 1

  - URL Format: `/restv2/customer/{custid}`
  - Supported HTTP Method: `GET`
  - Associated Service: `custdetails:custorders`

  Resource Operation 2

  - URL Format: `/restv2/customer/{order}`

- Supported HTTP Method: `GET`
- Associated Service: `custdetails:custorders`

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 13. Using the URL template-based approach, you can create a REST V2 resource named `discussion` 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 format:

```
/restv2/discussion/topic/{id}
```

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

```
GET /restv2/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 configuring supported HTTP methods for services, see *webMethods Service Development Help*.
restv2	Type of processing to perform, in this case, Integration Server REST processing. **Note:** For more information about directives, see *webMethods Integration Server Administrator's Guide*.
discussion	Name of the resource on Integration Server.
topic	Name of the static parameter in the URL format.
`236`	Identifier for a topic. Integration Server matches this value against the dynamic parameter `{id}` specified in the URL format. **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 format. For more information about

Where...	Is the...
	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 format:

```
/restv2/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/discussion/topic/t-1591 HTTP/1.1
```

Where...	Is the...
`t-1591`	Identifier for a topic. In the URL format 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 format (`t-{id}`). **Note:** In this example, Integration Server treats `t-1591` as a valid value considering the URL format specified for 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 format:

```
/restv2/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/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}`.

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 *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 *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 *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.

For more information about how Integration Server builds HTTP responses, see *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 *webMethods Integration Server Built-In Services Reference*.

3 Setting Up Your REST Application

Setting Up a REST Application on Integration Server

Integration Server can act as a REST server or REST client. For Integration Server to act as a REST server, it must host services that perform the GET, PUT, POST, PATCH, and DELETE methods. These services, which you provide, perform processing that is specific to your application.

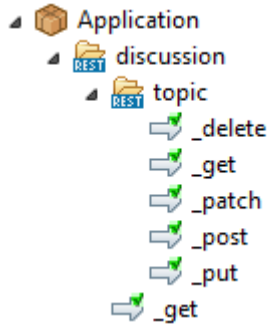
Services

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 13 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.

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 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*.

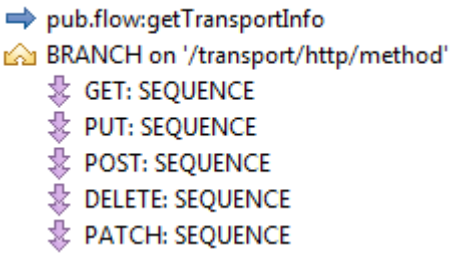
Converting an Existing Application

If you have an existing application that you want to transform into a REST application, you can 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.

If you want to use the legacy approach, then you can consider either of the following approaches to transform the existing 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.

4 Documenting Your Rest Application

Providing Information About Your Application

It is important to document your REST application so that your customers and partners will be able to build clients that interact with it correctly. Your documentation should cover how to:

- Send requests to your application
- Handle responses from your application

The following sections describe the different areas your documentation should cover.

General Information

Include the following general information about your application:

- A list of resource types

  In the sample Discussion application described above, resource types would be discussion and topic.

- The HTTP methods your application supports for each resource

  In the sample Discussion application, the discussion resource supports GET, but the topics resource supports DELETE, GET, POST, PATCH, and PUT.

Information About Each Request

Include the following information about each request:

- The format of the request URL

  For example, documentation for the Discussion application could provide a list of possible client requests:

  - Return general information about the Discussion application:

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

  - Return a list of all topics contained in the database:

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

  - Display entries made by participant Robertson in 2009 to topic 3419:

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

- Which request header fields are required or optional and how your application responds to them. For example, the Discussion application might specify the following information to explain which header fields it accepts and how it responds to them:

- **Authorization.** The Discussion application accepts BASIC and DIGEST authorization. All requests must include an Authorization header.
- **Content-Type.** Clients should include a Content-Type header with all requests. Acceptable Content-Type values for requests that contain a body are application/json, application/xml, text/xml, text/html, and text/plain. For more information about Content-Types, see *webMethods Integration Server Administrator's Guide*.
- **Accept.** Clients can optionally supply an Accept header to indicate the Content-Type they want the response to use. When you specify the Content-Type for the Accept header, Integration Server uses the content handler registered to that Content-Type to respond to the request. For example, if the content handler is application/json, Integration Server responds to the request with JSON content. Acceptable values are application/json, application/xml, text/xml, and text/html. If no Accept header is specified in the request, the response will use text/xml. For more information about the Accept header, see *webMethods Integration Server Administrator's Guide*.

- Whether a body is required and what structure the body should have.

  Documentation for the Discussion example might provide the following examples to illustrate body structure:

  Example 1: Creating a new topic

  Request:

```
POST  /discussion/topic HTTP/1.1
Host: IS_server:5555
Authorization: BASIC <your-credentials>
Content-Length: <request-body-length>
Content-Type: text/xml; charset=utf-8
```

  Response: If the request was valid, the Discussion application will respond with the following:

```
HTTP/1.1 201 Created
Content-Length: 0
ETag: 32619
Location: http://host/discussion/topic/32619
```

  Example 2: Adding an entry to an existing topic

  Request:

```
PUT /discussion/topic=36219 HTTP/1.1
Host: IS_server:5555
Authorization: BASIC <your-credentials>
Content-Length: 17
Content-Type: text/xml; charset=utf-8
comment=I+agree
```

  Response: If the request was valid, the Discussion application will respond with the following:

```
HTTP/1.1 200 OK
Content-Length: 0
Location: http://host/discussion/topic/36219?comment=2
```

Information About Responses

Your documentation should include the following to describe the response that corresponds to each request:

- A list of HTTP Status-Codes and Reason-Phrases the application returns and the circumstances under which it returns them. For a list of possible responses that you can code your application to return, refer to http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.2.
- A list of the response header fields you return and what they mean in the context of your application.
- A description of what will appear in the body of the response.

Index