Implementing REST Methods

The Bridge helps you with implementing REST methods. This page explains how REST methods have to be implemented so they can be called via REST requests and how different kinds of REST parameters have to be defined so they are provided to the service automatically by the Bridge.

Example File (Builder project Add-ons/REST):

<your example path>\Add-ons\REST\uml\supportManager.xml

<your example path>\Add-ons\REST\uml\supportManager_auth.xml

Example **Add-ons/REST/supportManager** shows the REST implementation of a simple support case manager. With this example, you can

- create a new support case using a POST request
- get information on support cases in general, on specific support cases, and on customers using GET requests
- mark a support case as resolved using a PUT request
- mark a support case as closed using a DELETE request

For simplification reasons, the example uses persistent states to persist the support cases. Normally, you would probably implement this use case using a database instead.

REST Methods

With REST methods, the Bridge distinct between **verb** methods and **named** methods (as described in REST Service). REST methods have to be static and must have stereotype <<REST>>.

Implementing Verb Methods

Verb methods intercept requests issued directly to the resource. Verb methods can only path parameters that are defined on the parent resource(s).
With the Bridge, you can implement all available HTTP methods as REST methods, as there are GET, POST, PUT, DELETE, PATCH, HEAD, and OPTIONS.

The Trailing "/"

Verb methods can be in form of `GET` or `GET/`. The difference is subtle but significant. It is important to understand the difference.

Have a look at the support manager example.

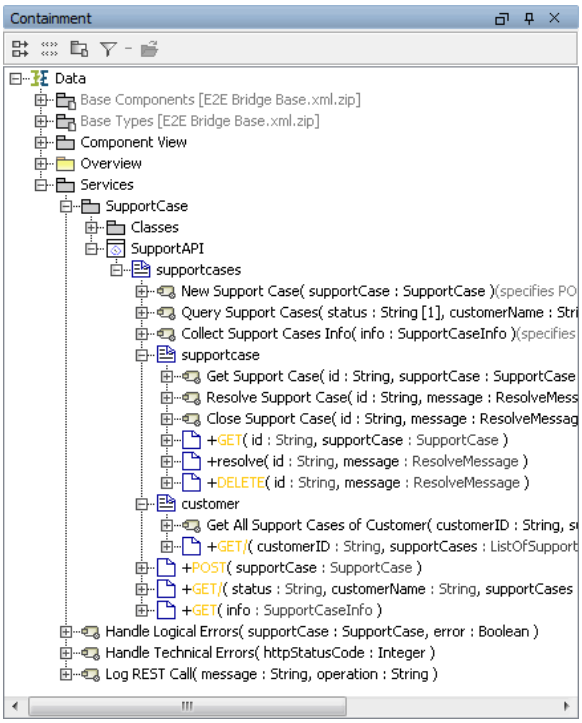
- A `GET` with a trailing **/** on the URL should always return a list of resources. Issuing a `GET` on `/support/supportcases/` is expected to return a list of existing support cases.
- A `GET` without trainling **/** o the URL should always return more general information on the resources. A `GET` on `/support/supportcases` is expected to return information on the support cases in general, e.g number of support cases, list of customers afflicted, ...

Look at the verb methods implemented in the example:

On this Page:

Related Pages:

- Defining a REST Service Interface
- REST Service Reference
- RESTful HTTP Service



Just assign the name of an HTTP method to the REST method and the Bridge call it on the corresponding request. Append a trailing **/** to the method name to make the distinction described in the note above (see `GET` and `GET/` on resources **supportcases** and **customer** in the figure above).

Find below an overview on the REST methods provided by the example:

Resource	REST Method	Description	Tag httpMethod	Input	In via	Output (body)
supportcases	GET	get a support case overview	GET or blank	none		**SupportCaseInfo**, a complex object containing some general information on the resource. In the example, these are number of support requests and a list of customers afflicted by support cases.
	GET/	query support cases	GET or blank	none		**ListOfSupportCases** Output parameters should be of complex type. Arrays are allowed but not accepted by all clients, so the compiler will throw a warning. Better wrap the array in a complex type: 
				status	query	
				customerName	query	
	POST	create a new support request	POST	**SupportCase**	body	**SupportCase**
supportcase	GET	get a specific support case	GET or blank	**id**	path	**SupportCase**
	DELETE	close support case	DELETE	**id**	path	**ResolveMessage** (Please also see note above.)
customer	GET/	get all support cases of a specific customer	GET or blank	**customerId**	path	**ListOfSupportCases** (Please also see note above.)

For more information on REST parameters, e.g. input methods, see section REST Parameters further below.

Implementing Named Methods

Named methods are methods that refer to a HTTP method, but have a divergent name, e.g. **resolve** in the support case manager example.
To call such an method, append its name (or **relativePath**) to the parent resource.

Resource	REST Method	Description	Tag httpMethod	Input	In via	Output (body)
supportcase	resolve	set the status of the support case to resolve	PUT	**id**	path	**ResolveMessage**

Error Handling

Each REST port type should have a <<RESTError>> class assigned. The Bridge will use this class as output in case of error (see Defining a REST Service Interface).

In the support manager example, REST port type SupportAPI has class **RESTError** assigned as error class.

Using function `getRestHttpResponse()`, you get access to the HTTP response object and can set the error details:

```
local response = getRestHttpResponse();
set response.responseObject = <my error object>;
set response.httpStatus = <HTTP status code>;
```

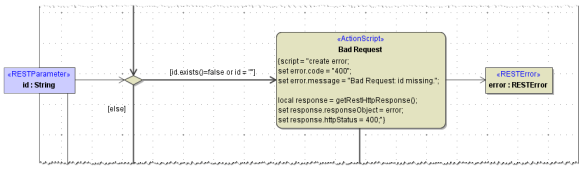
Assign the error object and a HTTP status code that corresponds to the error. This information will be returned via the HTTP response.

The xUML Runtime will recognize attributes as error code and/or error message under the following conditions:

- if you applied the names **code** and/or **message** to these attribute(s)
- if you applied the stereotypes <<RESTErrorCode>> and/or <<RESTErrorMessage>> to these attribute(s)

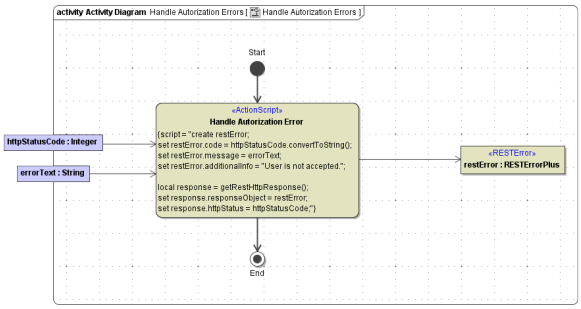
In this case, Runtime error codes and/or messages will automatically by assigned to these attributes in case of error.

In the support manager example, if user does not provide a support case id with the REST call, the implementation is as follows:



Error Handling for Specific Error Classes

If you defined a specific error class or **Blob** for a specific HTTP status code as described on Defining a REST Service Interface, you can use the same way as described above to provide the error details to the response. Provide the error details to an instance of the specific error class or **Blob** and provide the error class to the response object.
Look at activity **Handle Authorization Errors** of the support manager example:



REST Parameters

REST methods do not need to have parameters, but they can have. Typically, they will at least have an output parameter giving back a status or the resource content.
Parameters that are coming with the REST call are automatically provided to service parameters. Output parameters are provided to the request body.

If a parameter is not provided by the caller, it will come in as NULL or have the default value, if there is a default specified. For more information on how to specify a default, see Attribute Specification.

- **Input :** Input parameters can come via path, body, query, or header of the HTTP request. Principally and as a default, all defined input parameters are mandatory, only query and header parameters can be c hanged to be optional (see further below).
- **Output:** There can be no or exactly one output parameter via the response body. It has to be of complex type or of type **Blob**. More output parameters can be specified via the HTTP headers.

The Bridge supports JSON and XML content types, whereas JSON is the default if no divergent content type is specified. For more information on the supported content types, refer to Calling REST Services.

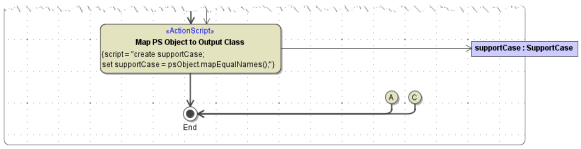
Output: Body Parameter

The REST method can provide no or exactly one output parameter via the response body. If an output is provided via the body, this has to be of complex type or of type **Blob**.

Depending on the **Content-Type** and **Accept** headers, the Bridge will provide the response as JSON or XML.
For more information on the supported content types, refer to Calling REST Services.

Example

Activity diagram **Get Support Case** shows the implementation of a body output parameter. It implements a GET request on a specific support case and the support case data is provided via a complex parameter.



Parameter **supportCase** is a complex structure and will written to the response body. The output format can be JSON or XML depending on the accept header specified in the request.
For more information on the supported content types, refer to Calling REST Services.

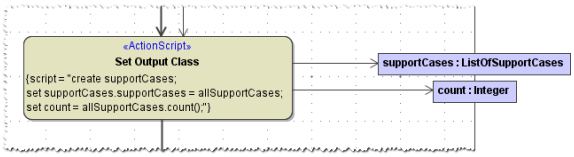
Output: Header Parameter

The REST method can provide multiple output parameters via the response headers. These parameters can be of simple type or array of simple type.

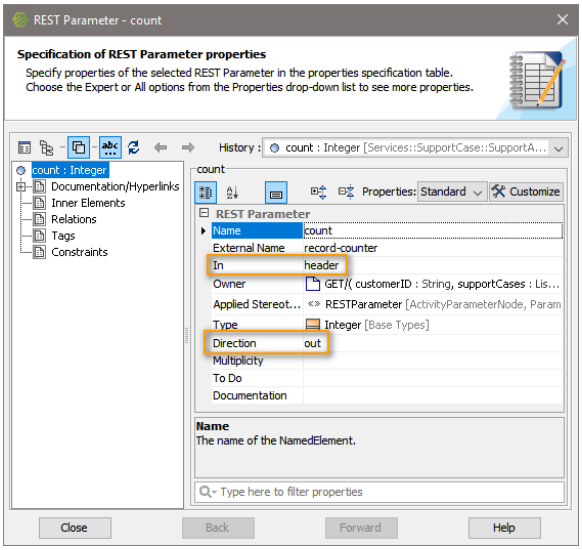
> As of Runtime 2019.1,we changed how the Runtime handles header parameters that are not set. If you do not set a header parameter that is defined for an operation, this parameter is omitted and not provided in the response headers.
> In older Runtime versions, such parameters were provided with `NULL`.

Example

Activity diagram **Get All Support Cases of Customer** shows the implementation of a header output parameter. It implements a GET request support cases of a specific customer. The support case data is provided via the response body as an array of complex type, the record count is provided via the response headers.



Parameter **count** is transferred via the response headers:



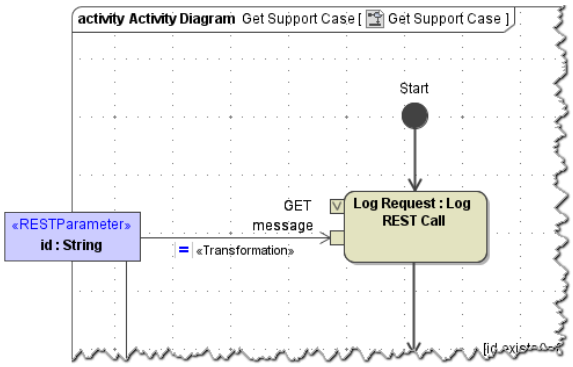
Input: Path Parameter

Path parameters are part of the path and, thus, part of the URL, e.g. `/support/supportcases/1234`. They are all required. All path parameters are automatically provided to method parameters of the service having the same name. They must be consumed by the called method and must have the same name as the path segment identifiers (without colon, though).

They can be of type **Integer**, **Float**, **String**, **Boolean**, and **DateTime**. During parsing the request, these are treated as **Strings** and then will be converted to the types defined in the model. Conversion errors will be handled by the Bridge by sending a 400 response ("Bad Request.").

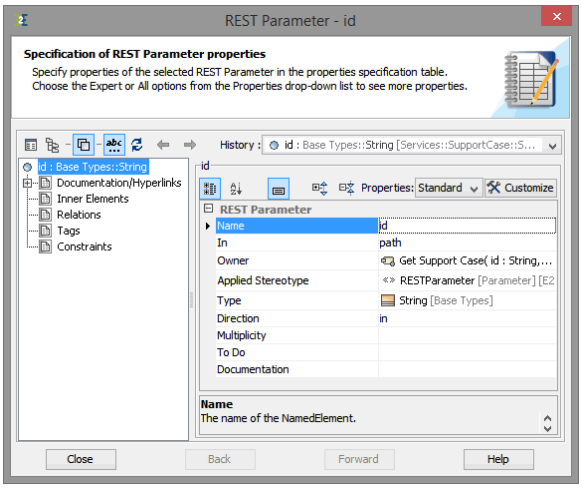
Example

Activity diagram **Get Support Case** shows the implementation of a path parameter. It implements a GET request on a specific support case and the support case id comes via the request URL.

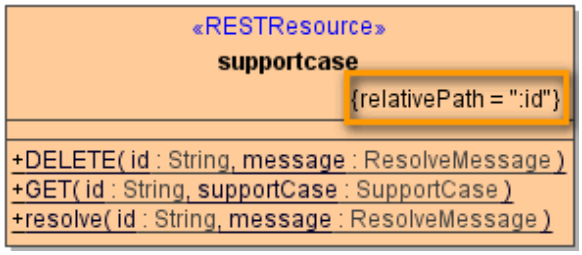


Parameter **id** is a rest parameter that comes with the resource path. Its value is transferred automatically to the method parameter having the same name. Type conversion to types divergent to **String** is applied automatically by the Bridge.

Figure: Specification of REST Parameter id - Input Method



As you can see from the screenshot above, parameter **id** has a defined input method **path**. That implies, that the corresponding resource has this parameter defined on its relative path.



Input: Body Parameter

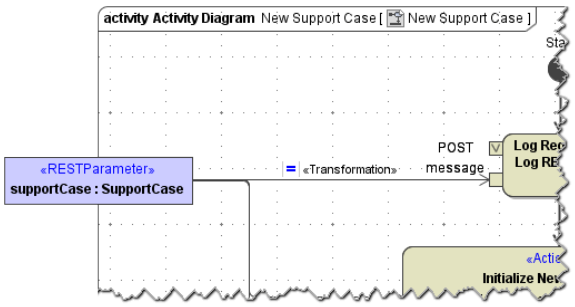
Body parameters are transferred in the request body. Since there is only one body in a HTTP request, only one body-parameter can be defined for an method. Body parameters have to be of complex type.

> To be in line with the HTTP specification, body parameters are allowed for PUT, POST and PATCH requests only.

The form of the body should correspond to the content type coming with the request. At the moment, the Bridge supports **application/json** and **text/xml** content types.
For more information on the supported content types, refer to Calling REST Services.

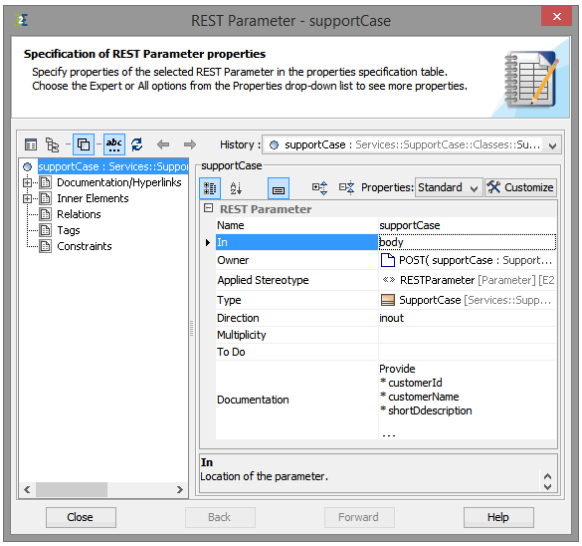
Example

Activity diagram **NewSupport Case** shows the implementation of a body parameter. It implements a POST request to create a new support case and the support case data comes via the request body.

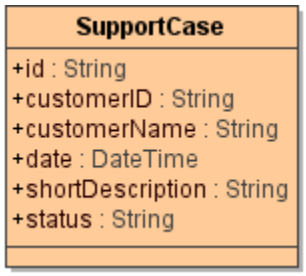


Parameter **supportCase** is a rest parameter of complex type that comes via the request body. Its value is transferred automatically to the method parameter. Type conversion is applied automatically by the Bridge.

Figure: Specification of REST Parameter supportCase - Input Method



The content that comes within the body (whether JSON or XML) must represent the structure given by complex type **SupportCase**.



Input: Query Parameter

Query parameters are provided to the service via the standard query-string. It is appended to the path after the **?** delimiter and key-value pairs are delimited by **&**, e.g. `/support/supportcases/?status=in%20progress&customerName=Wishes%20unltd` .
The Bridge will ignore unknown parameters, known parameters will be decoded and passed to the method. It is not necessary to provide all possible parameters with the query-string - omitted parameters will be NULL or have the specified default value (see Attribute Specification).

Query parameters can be of any simple type (**Integer**, **Float**, **String**, **Boolean**, **DateTime**, and **Blob**) or **A rray**. During parsing the request, these are treated as **Strings** and will be converted to the types defined in the model. Conversion errors will be handled by the Bridge by sending a 400 response ("Bad Request.").

Arrays

Arrays are accepted, but they must be continous arrays. A request like `/my_resource?a=1&b=2&a=3&a=4` is valid and produces two parameters:

- an array of **Strings** a = ['1', '3', '4']
- a **String** b = '2'

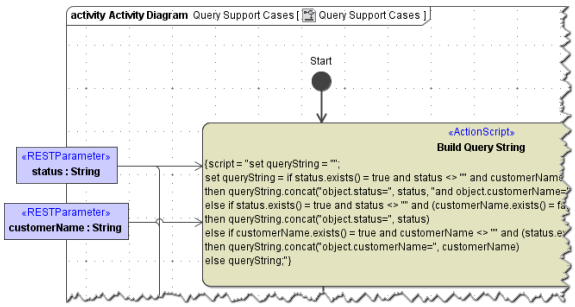
A request like `/my_resource? a[0]=1&b=2&a[5]=3&a[6]=4` is not valid.

Optional Parameters

As per default, all REST parameters are mandatory. As it concerns query parameters, this setting can be changed by the modeler using the multiplicity tag of the parameter. Set the multiplicity from **undefined** to **0..1** to make the parameter optional.
This setting will be written to the service descriptor file that describes the interface of the REST service.

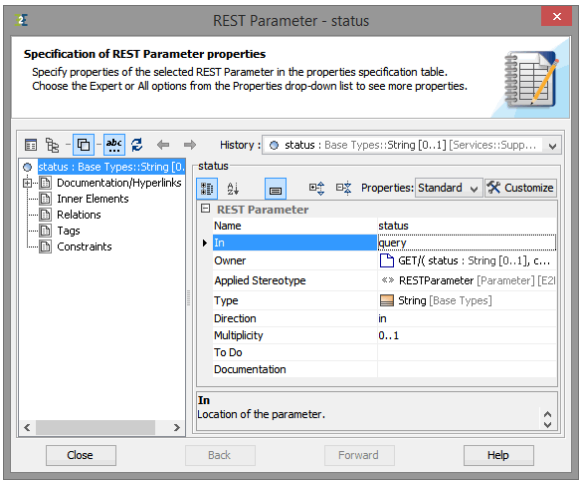
Example

Activity diagram **Query Support Cases** shows the implementation of a query parameter. It implements a GET request on support cases by status and/or customer.



Parameters **status** and **customerName** are rest parameters that come via the query-string. Their value is transferred automatically to the corresponding method parameters of the same name. Type conversion is applied automatically by the Bridge.

Figure: Specification of REST Parameter status - Input Method



Input: Header Parameter

Header parameters are transferred through request headers. Similarly like query parameters, unrecognised items are ignored.

Optional Parameters

As per default, all REST parameters are mandatory. As it concerns header parameters, this setting can be changed by the modeler using the multiplicity tag of the parameter. Set the multiplicity from **undefined** to **0..1** to make the parameter optional.
This setting will be written to the service descriptor file that describes the interface of the REST service.