REST Service

With the Builder, you can model REST services and develop software in the form of resources with RESTful interfaces. For more information on the concepts of REST refer to the Wikipedia pages of Representational State Transfer.

These pages explain, how REST concepts are implemented to the Bridge. The implementation supports JSON and XML content types and provides an OpenAPI 2.0 Specification for a REST service documentation and test.
You could also use plain HTTP to build RESTful services yourself - how to do this is described on RESTful HTTP Service. However, this approach is recommended only for content types divergent to JSON and XML.

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

The xUML REST service supports IPv6. For more information on how to **access** a REST service via the Bridge, refer to REST Adapter.

On this Page:

Related Pages:

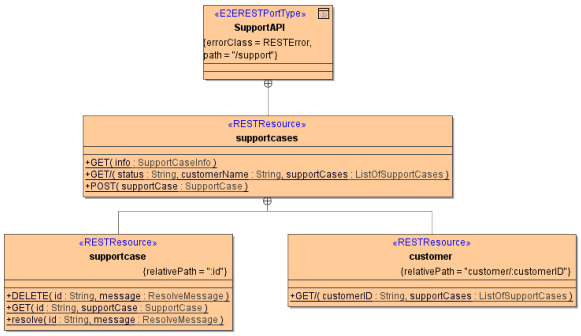
- Defining a REST Service Interface
- Handling Blobs in the REST Interface
- Documenting a REST Service
- Implementing REST Methods
- REST Service Error Handling
- Calling REST Services
- Testing REST Services
- REST Service Authentication
- REST Service Reference

- Representational State Transfer
- RESTful HTTP Service
- OpenAPI 2.0 Specification
- REST Adapter

Concepts and Conventions

The concepts of REST are based on Resource-Oriented Architecture (ROA) design principles. All elements accessible via the REST interface are REST resources, that can be accessed via their path.

Figure: Example Class Diagram of a REST Interface



The REST Resource Path

Resources can be accessed via their resource path. Together with protocol, server name/IP, and port, the paths form URLs. Unless proxied, the protocol is always HTTP in the Bridge.

The resource paths are calculated from the underlying model: the resulting path is a concatenation of all relative paths in the model hierarchy. The path starts with '/' - denoting server root - followed by the path definitions of the REST port type. Next, the path definition of each nested resource is appended, finalized by the path definition of the method.

Parts of the resource path can be dynamic, a so called path variable. This is a part of the path that can be given any value at runtime. Each dynamic path segment within a path has to have a unique name as it will be turned into an method parameter. Have a look at the figure above and the following examples:

Resource	Class	Resource Path	Example Resource Path
support API	SupportAPI	/support	`/support`
support cases	supportcases	/support/supportcases	`/support/supportcases`
a specific support case	supportcase	/support/supportcases/:id	`/support/supportcases/1234`

a specific method on a specific support case	supportcase	/support/supportcases/:id/resolve	`/support/supportcases/1234/resolve`
support cases of a customer	customer	/support/customer/:customerID	`/support/supportcases/customer/0815`

REST Parameters

Output

There can be exactly one output parameter that has to be of complex type. It will be written to response **body**. The format of the response depends on the definitions in the REST headers **Accept** and **Content-Type**. The Bridge supports JSON and XML.

Input

Input parameters can be provided via path, query, body, or header of the HTTP request.

Parameter	Description			Example
path	Path parameters are part of the path and, in consequence, the URL. They are all required.			**id** on rest resource **supportcase**
query	Query parameters are appended to the path after the '?' delimiter in standard query-string. Query parameters are optional.			**status** and **customerName** on resource **supportcases**, method **GET/**
header	Header parameters are transferred through request headers.			
body	Body parameters are transferred within the request body. Since there is only one body, only one body-parameter can be defined for a method. The form of the body should correspond to the **Content-Type** header, or **Accept** header if the former is not present. Bridge REST services accept both, JSON and XML content types.			**supportcase** on resource **supportcases**, method **POST**

REST Methods

REST Methods are methods that can be called on REST resources via HTTP requests. The Bridge supports the following HTTP methods for REST methods: GET, POST, PUT, PATCH, DELETE, OPTIONS and HEAD. REST methods have their own path relative to the parent resource they are defined on. Refer to The REST Resource Path further above for an example and to Defining REST Operations for more details.

REST Documentation

Analogously to the WSDL file of SOAP services, the Builder generates an OpenAPI 2.0 Specification service descriptor encoded in YAML (Swagger), that describes the REST interface.
The Bridge comes with a REST service documentation and test tool that reads the service descriptor and shows an overview of the capabilities of the REST service. Furthermore, it enables you to interact with the service and perform HTTP calls. Refer to Testing REST Services for more details.

REST Service and HTTP Protocol Support

Runtime 2019.9 Bridge xUML services read the following incoming HTTP headers containing correlation information:

- **X-Transaction-Id** or **xTransactionId** (in JMS context)
  This header identifies the transaction the call belongs to. You can set the transaction id manually with setTransactionID. If not set, the Runtime will generate one.
  This header will be passed through the callstack to identify all service calls that belong to a transaction.
- **X-Request-Id**
  This header should identify the unique request.
- **X-Sender-Host** and **X-Sender-Service**
  These headers should contain the sender host resp. the sender service.

These headers will be all logged to the transaction log. Having this information, you can use this for error analysis or usage metrics.

HTTP Header Roles

Runtime 2020.12 If the standard HTTP header handling does not meet your needs, you can take control of the header handling by defining your own header roles.
Refer to HTTP Header Support > Overwriting the Standard HTTP Headers for a detailed explanation of how to use this feature.