Working with APIs

This page describes how you can work with APIs that allows messages to be sent directly into the ESB. It contains the following sections:

- Introduction
  - Defining an API
  - Specifying the context
  - Using patterns with resources
    - URL mappings
    - URI templates
  - Examples
  - Configuring non-HTTP endpoints
- Basic REST architectural principles
  - Uniform interface
  - Stateless
  - Cacheable
  - Client–server
  - Layered system
- Best practices for designing APIs for use with REST

Introduction

An API allows you to send a message directly into the ESB and perform specific logic on it based on the instructions in the HTTP call. This section introduces the fundamentals of creating APIs in the ESB. It illustrates the concepts through the XML configuration.

For information on adding an API via the ESB tooling plug-in, see Working with APIs via WSO2 ESB Tooling.

For information on adding an API through the Management Console UI, see Working with APIs via the Management Console.

Defining an API

The syntax of a REST API is as follows.

```
<api name="API_NAME" context="URI_PATH_OF_API"
[hostname="HOST_NAME_OF_SERVER"] [port="PORTNUMBER"]>
 <resource [methods="GET|POST|PUT|DELETE|OPTIONS|HEAD|PATCH"]
 [uri-template="URI_TEMPLATE"|url-mapping="URL_MAPPING"]>
 <inSequence>
 ... 
 </inSequence>?
 <outSequence>
 ... 
 </outSequence>?
 <faultSequence>
 ... 
 </faultSequence>?
 </resource>
</api>
```

An API definition is identified by the `<api>` tag. Each API must specify a unique name and a unique URL context.
An API is made of one or more **resources**, which is a logical component of an API that can be accessed by making a particular type of HTTP call. For example:

```xml
<api name="API_1" context="/order">
   <resource url-mapping="/list" inSequence="seq1" outSequence="seq2"/>
</api>
```

Once a request is dispatched to a resource it will be mediated through the in-sequence of the resource. At the end of the in-sequence the request can be forwarded to a back-end application for further processing. Any responses coming from the back-end system are mediated through the out-sequence of the resource. You can also define a fault-sequence to handle any errors that may occur while mediating a message through a resource.

### Specifying the context

An API in WSO2 ESB is analogous to a web application deployed in the ESB runtime. Each API is anchored at a user-defined URL context, much like how a web application deployed in a servlet container is anchored at a fixed URL context. An API will only process requests that fall under its URL context. For example, if a particular API is anchored at the context “/test”, only HTTP requests whose URL path starts with “/test” will be handled by that API.

If your ESB is deployed in **multi-tenancy mode**, you must add the tenant domain to the API context when defining the API in the Source view instead of the Design view in the Management Console. For example, if you are defining the API in the tenant abc.com, and the context for the API is /order, you would specify the context as: context="/t/abc.com/order". If you define the API in the Design view, however, the tenant domain is automatically prepended to the context, so you would just specify the context as /order without the tenant.

It is also possible to bind a given API to a user-defined hostname and/or a port number. For example, if your API is anchored at the URL `http://your.host.name/test`, the following requests will be handled by your API:

- `http://your.host.name/test/getNumbers`
- `http://your.host.name/test/calculation/getRate`

### Using patterns with resources

A resource can be associated with a user-defined **URL mapping** or **URI template**, which allow us to restrict the type of HTTP requests processed by a particular resource. A resource can also be bound to a specific subset of HTTP verbs and header values, providing additional control over what requests are handled by a given resource.

For example, consider a resource associated with the URL mapping “/foo/*” and the HTTP verb “GET”. This approach ensures that the resource will only process GET requests whose URL path matches the pattern “/foo/*”. Therefore, the following requests would be processed and mediated by the resource:

- `GET /test/foo/bar`
- `GET /test/foo/a?arg1=hello`

The following HTTP requests would **not** be handled by this resource:

- `GET /test/food/bar` (URL pattern does not match)
- `POST /test/foo/bar` (HTTP verb does not match)

### URL mappings

When a resource is defined with a URL mapping, only those requests that match the given pattern will be processed by the resource. There are three types of URL mappings:
Path mappings (/test/*, /foo/bar/*)
Extension mappings (*.jsp, *.do)
Exact mappings (/test, /test/foo)

URI templates

A URI template represents a class of URIs using patterns and variables. When a resource is associated with a URI template, all requests that match the template will be processed by the resource. Some examples of valid URI templates are as follows:

/order/{orderId}
/dictionary/{char}/{word}

The identifiers within curly braces are considered variables. For example, a URL that matches the template "/order/{orderId}" is as follows:

/order/A0001

In the above URL instance, the variable "orderId" has been assigned the value “A0001”. Similarly, the following URL adheres to the template “/dictionary/{char}/{word}”:

/dictionary/c/cat

In this case, the variable “char” has the value “c” and the variable “word” is given the value “cat”.

The ESB provides access to the exact values of the template variables through message context properties. For example, if you have a resource configured with the URI template “/dictionary/{char}/{word}”, and a request “/dictionary/c/cat” is sent to the ESB, it will be dispatched to this resource, and we will be able to retrieve the exact values of the two variables using the "get-property" XPath extension of WSO2 ESB and prefixing the variable with "uri.var." as shown in the following example:

<log level="custom">
  <property name="Character" expression="get-property('uri.var.char')"/>
  <property name="Word" expression="get-property('uri.var.word')"/>
</log>

This log mediator configuration would generate the following output for the request “/dictionary/c/cat”:

LogMediator Character = c, Word = cat

Examples

Let's look at some sample API configurations to understand how we use context, resources, and patterns to control how requests are processed.
You can define a URL mapping to a set of operations as shown in the API_1 definition, or you can define separate mappings for separate operations as shown in API_2. Also note the last resource definition in API_3, which does not specify a URL mapping nor a URI template. This is called the default resource of the API. Each API can have at most one default resource. Any request received by the API that does not match any of the enclosed resource definitions will be dispatched to the default resource of the API. In the case of API_3, a DELETE request on the URL ”/payments” will be dispatched to the default resource as none of the other resources in API_3 are configured to handle DELETE requests.

For a comprehensive example of using APIs with the ESB, see the article How to GET a Cup of Coffee the WSO2 Way.

Configuring non-HTTP endpoints

When using a non-HTTP endpoint, such as a JMS endpoint, in the API definition, you must remove the REST_URL_POSTFIX property to avoid any characters specified after the context (such as a trailing slash) in the request from being appended to the JMS endpoint. For example:
Notice that we have specified the REST_URL_POSTFIX property with the value set to "remove". When invoking this API, even if the request contains a trailing slash after the context (e.g., POST http://127.0.0.1:8287/orderdelayAPI/ instead of POST http://127.0.0.1:8287/orderdelayAPI), the endpoint will be called correctly.

Basic REST architectural principles

The structure of APIs in the ESB is based on REST architectural principles. This section highlights some of the basic architectural principles of REST based on the following resources:

- Architectural Styles and the Design of Network-based Software Architectures
- Restful Service Best Practices

Uniform interface

REST provides a uniform interface between clients and servers, which simplifies and decouples the architecture, allowing both to evolve independently. To achieve a uniform interface, the following constraints are applied:

- Resource-based: Individual resources are identified in requests using URIs as resource identifiers.
- Manipulation of resources through representations: When a client holds a representation of a resource including any metadata attached, it has enough information to modify or delete the resource on the server, provided it has the permission to do so.
- Self-descriptive messages: Each message includes enough information to describe how to process the message.
- Hypermedia as the engine of application state (HATEOAS): Clients deliver state via body contents, query-string parameters, request headers, and the requested URI. Services deliver state to clients via body content, response codes, and response headers. This is technically referred to as hypermedia (or hyperlinks within hypertext).

Stateless
The necessary state to handle the request is contained within the request itself, whether as part of the URI, query-string parameters, body, or headers.

Cacheable

As on the World Wide Web, clients can cache responses. Responses must therefore, implicitly or explicitly, define themselves as cacheable or non-cacheable to prevent clients from reusing stale or inappropriate data in response to further requests.

Client–server

The uniform interface separates clients from servers. For example, clients are not concerned with data storage, which remains internal to each server, so that the portability of client code is improved. Servers are not concerned with the user interface or user state, so that servers can be simpler and more scalable. Servers and clients may also be replaced and developed independently, as long as the interface is not altered.

Layered system

A client cannot ordinarily detect whether it is connected directly to the back-end server or to an intermediary along the way. This allows for load balancing and caching.

Best practices for designing APIs for use with REST

This section highlights some best practices from https://s3.amazonaws.com/tppearsoncollege/bestpractices/RESTf ul+Best+Practices.pdf to keep in mind when designing your APIs.

- **Use meaningful resource names** to clarify what a given request does. A RESTful URI should refer to a resource that is a thing instead of an action. The name and structure of URIs should convey meaning to those consumers.
- **Use plurals in node names** to keep your API URIs consistent across all HTTP methods.
- **Use HTTP methods appropriately.** Use POST, GET, PUT, DELETE, OPTIONS and HEAD in requests to clarify the purpose of the request. The POST, GET, PUT and DELETE methods map to the CRUD methods Create, Read, Update, and Delete, respectively. Each resource should have at least one method.
- **Create at most only one default resource** (a resource with neither a uri-template nor a url-mapping) for each API.
- **Offer both XML and JSON** whenever possible.
- **Use abstraction when it's helpful.** The API implementation does not need to mimic the underlying implementation.
- **Implement resource discoverability through links (HATEOAS).** As mentioned in the previous section, the application state should be communicated via hypertext. The API should be usable and understandable given an initial URI without prior knowledge or out-of-band information.
- **Version your APIs** as early as possible in the development cycle. At present, the ESB identifies each API by its unique context name. If you introduce a version in the API context (e.g., /Service/1.0.0), you can update it when you upgrade the same API (e.g., /Service/1.0.1).
- **Secure your services** using OAuth2, OpenID, or another authentication/authorization mechanism. See also Securing APIs.