REST Interface Design Standards v1.1

Contents

[Document Revisions 2](#_Toc444523699)

[Example Implementation (Template Project) 2](#_Toc444523700)

[Appendix 3](#_Toc444523701)

[Previous Documentation 3](#_Toc444523702)

[REST Web Service HTTP Verb Usage Standards 4](#_Toc444523703)

[REST Web Service URI Path Definition Standards 4](#_Toc444523704)

[*Do Not Use Verb in URI* 4](#_Toc444523705)

[*Avoid Resource Representation Based on File Extensions* 4](#_Toc444523706)

[*Resource Grouping* 5](#_Toc444523707)

[*Resource Naming* 5](#_Toc444523708)

[*Versioning* 6](#_Toc444523709)

[*Avoid Resource URI Overloading* 6](#_Toc444523710)

[*Use Request Payload for Data Transfer avoid URI Tunnelling* 6](#_Toc444523711)

[*Use URI Template* 6](#_Toc444523712)

[*Use Query Parameters for Optional Properties, Sorting, Filtering, Ordering, Paging* 7](#_Toc444523713)

[REST Web Service Payload Definition Standards 7](#_Toc444523714)

[*HATEOAS (Hypermedia as the Engine of Application State)* 8](#_Toc444523715)

[REST Web Service HTTP Response Code Standards 9](#_Toc444523716)

[REST Interface Design Tool 9](#_Toc444523717)

# Document Revisions

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Date | Author | Role | Version | Comments |
| 28/01/2016 | Ali Sahin | Developer | v1.0 | 1st draft |
| 29/02/2016 | Ali Sahin | Developer | v1.1 | Standard Design Tool |
| 09/03/2016 | Ali Sahin | Developer | V1.1 | Fixing typo for PUT definition in Pg.4 |

# Example Implementation (Template Project)

<http://galaxy-gerrit.service.test.group:8080/#/admin/projects/galaxy/rest/example-rest-service>

**master** branch: Simple REST service implantation with CXF.

**with-ea-components**branch: EA components implemented (Security Filter)

# Appendix

**AFT:**Automated Functional Testing, BDD based end to end automated acceptance testing

**API:**Application Programming Interface, an interface from an application for other applications consumption

**CRUD:**Create, Read, Delete, Update basic data transformation operations

**HTTP:**Hyper Text Transfer Protocol, A Web protocol that defines the data transfer over Web

**REST:** Representation State Transfer, A distributed system approach that work over Web

**RPC:**Remote Procedure Call, a Web approach to call Web Services

**SOAP:**Simple Object Access Protocol, a protocol that defines reach to the web services via XML based payloads

**URI:**Uniform Resource Identifier, A data structure to identify data resource location

**W3C:** World Wide Web Consortium, An organisation defines Web Standards

**Web Service:**Software which serves data and logic over Web

**WSDL:**Web Service Definition Language, XML based SOAP based web service contracts

**XML:**Extensible Markup Language, A data structure widely used in Web

**XPATH:**path basedidentifierstructure for XML documents

**XSD:**XML Schema Definition, definition documents for the data structures represented by XML

# PreviousDocumentation

Representational State Transfer Standard by John Jackson: <http://teamspace.intranet.group/sites/SOAGovernance/REST%20Documents%20and%20Standards/Representational%20State%20Transfer%20Standard%20V0.87.docx>

# REST Web Service HTTP VerbUsage Standards

The HTTP request verbs should be used as they are defined by the W3C HTTP standards documentation (<http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html>). CRUD (Create, Read, Delete and Update) operationsare well supported with the most popular HTTP verbs.

* **POST**: is used for data transfer from client to the server. This request should contain the request payload “Content-Type” in the request if payload exists. POST matches with “Create” in the CRUD services. If the service create a resource for the request, it should return with “201 Created“ response code with “Location” header give the created resources URI.
* **GET**: is defined as “safe” operation which means it can be executed multiple times andthe request does not make a state change in the resourceas a result of the request itself. In other words, the resource statemay change during the calls but this state change should not be caused by the GET request. GET matches with “Read” operation in CRUD services. For happy path scenarios, service should return “200 Ok” response code with the “Content-Type” header.
* **PUT**:There is a slight difference between POST and PUT. In PUT operation, the request provides full resource URI. Hence, we use PUT request to create or override (update) the resource under this specific URI. That makes it a good match for “Update” operation in CRUD services.
* **DELETE**: is used for resource deletion. Request requires full URI of the REST resource that will be affected by this operation. A happy path returns with “200 Ok” response code. This request does not contain a payload.

# REST Web Service URI Path Definition Standards

While we design the REST service interface, we need to follow the standards for resource URI definitions.

## *Do Not Use Verb in URI*

Verbs give information about the request action. In REST services, an action on a resource is defined by HTTP verbs. Adding the verbs in the URIs can cause semantic confusion in the HTTP request. For instance, let’s say we have a product arrangement service under the lloyds.co.uk service

*http://lloyds.co.uk/sales/products/arrangements/create*

In this example we explicitly mention about the REST resource as an arrangement creation request in the URI. And, let’s say we accept the POST requests for this URI. HTTP lets us to make GET or DELETE or PUT operations on the same resource. The client may call “*arrangement/create”* with GET and may expect to create a resource. This is apossible conflict between URI and HTTP verb semantics. This is a bad practice and we should avoid this.

## *Avoid Resource Representation Based on File Extensions*

We should avoid using fie extensions to represent the resource format in the logical REST resources. HTTP is capable of state representation of a resource in a different way. A resource is a piece of data or functionality servedunder a URI. For instance,“*http://foo.com/bar*“ is a representable resource and calling “*http://foo.com/bar.****txt***“ for text representation and “*http://foo.com/bar.****xml***“ for XML representation is fundamentally changing the resource URI, hence asking for a different resource. Different URIs can point the same resource, there is no restriction for that but instead we can handle this in more clean way with help of HTTP “Accept” header. So, the request should set header;

*Accept: “plain/text”*

for text format of the *“bar”* resource. Request should set the same header to *“application/xml”*for the same resource’s XML representation. Server should set the *“Content-Type”*in the response header with the resource format it returns to the client.

## *Resource Grouping*

We should group the resources logically based on the services we provide. For example, we provide sales service in a REST application. Under sales we provide different capabilities of sales for different versions. And finally, sales service consists of multiple sub services based on the service specifics.

*http://lloyds.co.uk/sales-api/v1/products/arrangements*

This is from the most general group to the most specific feature approach. So the standard structure to a REST resource should have the pattern below.

*http://<host>/<application/api-name>/<version>/<service>.../<resource>...*

## *Resource Naming*

**Hyphen between Words:** If your resource name consists of multiple words, as a standard these words should be separated by hyphen (-) character. E.g.

http://lloyds.co.uk/service-api/v1/account-arrangement

**Plural and Singular Resources:** If the resource we are dealing with is a CRUD[[1]](#footnote-2) resource then we should use plural naming.For other cases we use the appropriate naming which explains the resource.

For instance: if we **create** an arrangement,the URI for this operation in the example HTTP request should be like below:

POST /service-api/v1/arrangements

Likewise, we make **read, delete** or **update** on the arrangement resource and the resource should be named plural.

Read: GET /service-api/v1/arrangements/3345

Update: PUT /service-api/v1/arrangements/3345

Delete: DELETE /service-api/v1/arrangements/3345

On the other hand, REST services are not always about CRUD resources. Some resources can be logical resource. Let say, we have a REST service resource for calculating the time difference between two dates and does not do a persistence operation. The possible URI for this resource could be like below:

POST /service-api/v1/calculators/time-difference

Since this is not a CRUD service, self-explaining resource naming is important. In this case, calling the resource name as “time-differences” does not make sense.

## *Versioning*

Versioning the API in the URI is a standard for the REST Interface Design. Unless there is another reason the version should have *v<version>* format. E.g. v1,v2

## *Avoid Resource URI Overloading*

Use separate URI for the separate resources. Instead of reusing the same resource URI, try to group them under related paths. For example, if we have 2 different product arrangements for 2 different products “card” and “account”. Instead of using “product/arrangement” with two different payload structures per product, you should create 2 URIs under product for card as “/products/cards/arrangements” and account as “/products/accounts/arrangements”

## *Use Request Payload for Data Transfer avoid URI Tunnelling*

HTTP lets us to share data between client and server in different ways. With URI Tunnelling, we can specify parameters for request in URI. For instance, the request below

*http://lloyds.co.uk/sales/products/arrangements?account=classic&userId=5*

sends the account and user id information as part of the URI as query strings. This approach is not appropriate. First of all, as long as the request parameters get bigger, we will end up with complicated URIs. And for complicated parameter structures, URI is simply not enough. Instead, we can pass request parameters in request payload. We use JSON as the standard payload type

*POST /sales/products/arrangements HTTP/1.1*

*Host: lloyds.co.uk*

*Content-Type: application/json*

*{“account”:”classic”, “userId”:”5”}*

## *Use URI Template*

URI template is quite useful property of URI. It makes easy to understand and predict the service resources and behaviours.

*http://lloyds.co.uk/sales/products/arrangements/{arrangementId}*

is a template URI that we know we can predict CRUD behaviour of arrangement service behaviour as below.

**Create**: POST */sales/products/arrangements*

**Read**: GET */sales/products/arrangements/1234*

**Update**: PUT */sales/products/arrangements/1234*

**Delete**: DELETE */sales/products/arrangements/1234*

We can also use URL templates for more advanced resource definitions such as

/contents/{contentPath}/{contentKey} -> /contents/classic\_path/521

## *Use Query Parameters for Optional Properties, Sorting, Filtering, Ordering, Paging*

Any optional property on a resource can be specified as query parameter.

For example if you’d like to get the resource sorted by a resource property you shouldspecify this as a query parameter. Plus (+) symbolises the ascending hyphen (-) symbolises the descending sorting. Default should be ascending sorting.

*http://lloyds.co.uk/sales/arrangements?sort=+arrangementDate,-id,type*

If you’d like to display specific fields of a resource that should be specified in fields query parameter. For example:

*http://lloyds.co.uk/sales/arrangements/1234?fields=arrangementDate,product,type*

For collection resources, we can specify the filtering via query parameters as an example below:

*http://lloyds.co.uk/sales/arrangements?status=Success&accountType=Classic*

Paging in the collection resources should be passed as a query parameter. For instance, if we would view arrangements between 30th and 40th, we use the URI below:

*http://lloyds.co.uk/sales/arrangements?offset=30&limit=10*

Any required parameter should be created as a path parameter. Usage of query parametersfor URI Tunnelling should be avoided as it is explained above.

# REST Web Service Payload Definition Standards

JSON is the standard payload format for both request and response. Payload should not carry any information regarding to the requested action in Remote Procedure Call (RPC) manner. Payload should carry all the information the server needs to process the action under the resource defined by the URI and the client needs to understand the server response.

An error response payload should carry at least 2 fields.

* Error Code: A unique code for the error situation that represents the error scenario for the user
* Error Message: A useful error messaging for the client (useful for debugging)

Payload should be associated with a “Content-Type” header in both request and response.

JSON payload keys should be named meaningfully, camel cased (ifthe key consist of multiple words). Generic naming should be avoided. An example arrangement payload should look like below:

{

arrangementId:5,

arrangementRetryCount:1,

arrangementOptions:[“Current Account”, “No Interest”],

mnemonic: “CLASSIC\_ACC”

}

Naming of the keys in the JSON payload should be consistent among all the services and the interface designer should avoid synonymous naming. E.g. Use of “lastName” in some services and “surname” in others is not desired.

## *HATEOAS (Hypermedia as the Engine of Application State)*

HATEOAS is a useful feature for REST services to develop loosely coupled relationship with the service clients. Basically, a service share all possible application states for the further steps as part of the response.

For example: a response for a request which retrieves an arrangement returns response payload with HATEOAS like below;

*Request:*

*GET /products/arrangements/12345 HTTP/1.1*

*Response:*

*HTTP/1.1 200 OK*

*Content-Type:application/json*

*{*

*“arrangementId”:”12345”,*

*“type”:”Classic Account”,*

*“\_links”:*

*[*

*{*

*“rel”:“self”,*

*”href”:”http://lloyds.co.uk/products/arrangements/12345”*

*},{*

*“rel”:“update”,*

*”href”:”http://lloyds.co.uk/products/arrangements/12345”*

*},{*

*“rel”:“activate”,*

*“href”:”http://lloyds.co.uk/products/activations/12345”*

*}*

*]*

*}*

As part of the response body we create a list of possible next step links for the application state. As long as the client depends on the link name instead of the direct link, the server can change the link structure without a dependency to the client. This is very useful and strong approach and a fundamental for fully integrated REST services. HATEOAS should be implemented in an agreement with the client application.

# REST Web Service HTTP Response Code Standards

HTTP response codes are important elements of REST web services. Usage of the response codes must match with the HTTP standards defined by W3C

(<http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>). Some commonly used response codes are;

* 200 (OK): This response code is returned for specific HTTP verbs such as GET, HEAD, POST, and TRACE. This status code indicates that the request has succeeded.
* 201 (Created): The request has succeeded and a new resource is created. The new resource should be returned in the “Location” header.
* 204 (No Content): Request has succeeded and response does not contain a payload.
* 400 (Bad Request): Request could not be understood by the server. It has a bad or invalid format. The error response should tell to the end user about the reason that why the request is in bad format (if there is a missing or invalid formatted field that needs to be told to the end user).
* 404 (Not Found): The URI inside the request does not have a matching resource in the server side.
* 500 (Internal Server Error): Unexpected error happened in the server.
* 503 (Service Unavailable): The server is currently unavailable.

# REST Interface Design Tool

Swagger is the strategic direction for API specification. It has better tooling and support for API Gateway vendors. All green field projects should use Swagger. For more information about swagger and its mark-up language (YAML), please visit [www.swagger.io](http://www.swagger.io). Architecture, Method and Innovation (AMI) contact is John Jackson, John.Jackson@Lloydsbanking.com.

1. Create Read Delete Update: 4 basic functions of persistent storage [↑](#footnote-ref-2)