REST API Standards

The purpose of this document is to outline the standards to be followed when creation of REST API.

REST API URL Naming Standards

- **Use nouns for naming URIs:** RESTful URIs should include **JAVA ENTITY** and not indicate any kind of CRUD (Create, Read, Update, Delete) functionality. Instead, REST APIs allow you to manipulate a resource which should take the form of a noun
    - *Example: HTTP GET /assets – CORRECT*
    - *Example: HTTP GET /getAssets – INCORRECT*

- **Pluralised Resources**: API Url should be plural i.e the resource/java entity should always be plural
    - *Example : HTTP GET /assets – To fetch details for assets*
    - *Example : HTTP GET /assets/123– To fetch details of asset with assetId=123*
- **Forward slashes for hierarchy:** Forward slashes are conventionally used to show the hierarchy between individual resources and collections
    - *Example: /assets/{id}/locations clearly falls under the /assets/{id} resource which falls under the /assets collection.*
- **Multiple or list of resources:** When we need to return multiple resource for a list of AssetId, it should be part of query params separated with comma. /assets return all the assets and /assets? id= {id1},{id2} returns multiple assets with id = id1 and id2.

    - *Example: /assets?id={id1},{id2} to access multiple resources*
- **Query parameters where necessary:** In order to sort or filter a collection, a REST API should allow query parameters to be passed in the URL.

    - *Example: /assets?location=USA to find all users living in the United State*
- **Lowercase letters and dashes:** By convention, resource names should use exclusively lowercase letters. Similarly, dashes (-) are conventionally used in place of underscores (_).

    - *Example:/assets/ asset-lines -CORRECT*
    - *Example:/assets/create_event or /assets/assetLines - INCORRECT*
- **Do not use file extensions :** While the result of an API call may be a particular filetype, file extensions are largely seen as unnecessary in URIs—they add length and complexity. If you want to specify the filetype of the results, you can use the Content-Type header instead.

REST Headers

- **x-oal-tenantCode:** This header is used to support multi tenancy in our application. One can also set boundaries for users from a particular tenant. While testing, developers often use dummy values, and there is generally a large volume of data as the system is put to different types of testing. On viewing data by tenant code, we can avoid the large volume of meaningless test data from the result. Developers can use ORCL_TEST tenant code for testing purposes, ORCL_DEV for development, ORCL_UAT for UAT and so on.
- **x-oal-appCode:** This is generally provided by the user/client and is specific to the application. For example, SCA is the app-code for Sales Crediting Application and OKSIB is the app-code for Service Contracts / Installed Base application
- **x-oal-username:** This is used to authorise the user. Oal-svc-gateway-service service validates whether if that particular user has enough rights to access the endpoint. With implementation of OAuth, use of x-oal-userName header should be excluded but for now we are still using it.

REST query parameters

Query parameters can be used to control what data is returned in endpoint responses.

- Query parameters are passed after the URL string by appending a question mark followed by the parameter name , then equal to (“=”) sign and then the parameter value. Example: */oalapp/services/api/iboksint/assets?status=COMPLETED*
- Multiple parameters are separated by “&” symbol. Example: */oalapp/services/api/iboksint/assets?status=COMPLETED&active=Y*
- Query parameter's name must start with a letter, generally accepted the camelCase and the snake_case also, but it has to be consistent. Use camelCase since we use this mostly.
- Query parameters can be multivalued, there is multiple way to handle them but the most recommended way is:
    - concatenated: *statuses=COMPLETED,IN_PROGRESS* . The parameter's name is plurar. The data has to be split at the servers side by the separator (comma). If the data elements can contain the separator it just won't work fine, not recommended.
- Query parameter can be optional or required. Usually it should be optional, unless there is cause to make it required.
- Query parameter's value always has to be encoded! For example: *name=Cloud&Services* would be interpreted as a parameter named name with value Cloud, and another parameter named Services without value. Properly (encoded) it should be sent as *name=Cloud%26Services*, that means a parameter named name with value Cloud&Services. More about encoding: HTML URL Encoding Reference

General query parameters:

- sort: information about how to sort the result, contains the field name, and the direction (separated by a comma), can be multivalued
    - sort=status : sorts asc by field named status
    - sort=name,asc&sort=creationDate,desc : sorts asc by the name firstly, and sorts desc by the creationDate secondly (in case of name equality)
- page: which page of a paginated result should be send back in the response, starts from 0, can not be multivalued, for example: page=2
- size: size of a page of a paginated result, starts from 1, can not be multivalued, for example: size=10

REST Verbs

Http method	CRUD operation	Use Case
POST	Create	To Create single or multiple new resources
GET	Read	To read the resource
PUT	Update/Replace	To modify the resource entirely instead of few attributes
PATCH	Update/Modify	To update few attributes of the resource
DELETE	Delete	To delete the resource

GET

- Used to retrieve an element or a collection of a resource
- In GET, input is in query params form and no payload should be there .
- Used to perform READ operation
- GET is idempotent: Calling it multiple times returns the same resource as calling once thus, idempotent
- Typical response codes:
    - Success cases: 200 (Ok) - read data
    - Error cases:
        - 400 (Bad Request) - if the input parameters are invalid
        - 404 (Not Found) - if ID not found or invalid, or the collection is empty. ALM link showcasing client and server side implementation for 404 not found response in GET call.
        - 404 Not Found Client side:https://alm.oraclecorp.com/oal/#projects/gxp/scm/rest-standard-client-sample.git/blob/src/main/java/oal/oracle/apps/ic/application/service/OrderServiceImpl.java?revision=changes-get and 404 Not found Server side : https://alm.oraclecorp.com/oal/#projects/gxp/scm/OAL-REST-Standards.git/blob/src/main/java/oal/oracle/apps/ic/application/web/OrderController.java?revision=ft-getRequestForOrders

POST

- POST method mostly used to create one or multiple resources.
- While making post call the request body should never consist of WHO columns like CREATION_DATE, LAST_UPDATE_DATE, CREATED_BY, LAST_UPDATED_BY
- Request body should not contain other columns like Primary Key or ID as it should be an auto generated sequence at server side.
- Other columns like VERSION and ENV is taken care by spring and the profile. So no need to pass them explicitly in the request body.
- Success cases:

    - 201 (Created) - new resource has been created

  Error cases:

    - 404 (Not Found) - if ID not found, or the collection is empty
    - 409 (Conflicts) - to one resource, if the resource exists
    - 400 (Bad Request) - if the input parameters are invalid
- **Idempotency**: POST is not idempotent as same post call when made multiple times will create multiple resources, thus **NOT** idempotent.
- The **Location** response-header field is used to redirect the recipient to a location other than the Request-URI for completion of the request or identification of a new resource. For 201 (Created) responses, the Location is that of the new resource which was created by the request.

PUT

- PUT method mostly used to update/replace a single collection resource entirely.
- While making put call the request body should never consist of WHO columns like CREATION_DATE, LAST_UPDATE_DATE, CREATED_BY, LAST_UPDATED_BY
- Request body should not contain other columns like Primary Key or ID as it should be an auto generated sequence.
- Other columns like VERSION and ENV is taken care by spring and the profile. So no need to pass them explicitly in the request body.
- Success cases:

    - 200 (Ok) - updated

  Error cases:

    - 404 (Not Found) - if ID not found or invalid, or the collection is empty
    - 400 (Bad Request) - if the input parameters are invalid
    - 405 (Method not allowed) - in case of the whole collection replacement should not be allowed
- PUT is **idempotent** – if we update the same resource with same data, then it remains the same. But if we have counter in the resource, then it is not idempotent(version).

DELETE

- DELETE method mostly used to delete one element .
- Example: HTTP DELETE */oalapp/services/api/iboksint/assets/123 : This will delete the asset with assetId 123 and return 200 OK response.*
- No body needs to be passed for DELETE operation.

- Success cases:
    - 200 (Ok) - deleted
    - 204 (No Content) - no element to update/delete
- Error cases:
    - 404 (Not Found) - if ID not found or invalid, or the collection is empty
    - 400 (Bad Request) - if the input parameters are invalid
    - 405 (Method not allowed) - in case of whole collection replacement should not be allowed.
- **Idempotency**: DELETE is idempotent. If we repeatedly delete the resource, we receive same response. Calling DELETE for second time will return “404 Not found”

PATCH

- PATCH is used to update the resource, but complete resource need not be provided, only changes would suffice.
- While making patch call the request body should never consist of WHO columns like CREATION_DATE, LAST_UPDATE_DATE, CREATED_BY, LAST_UPDATED_BY
- Request body should not contain other columns like Primary Key or ID as it should be an auto generated sequence.
- Other columns like VERSION and ENV is taken care by spring and the profile. So no need to pass them explicitly in the request body.
- Success cases:
    - 200 (Ok) - updated
    - 204 (No Content) - no element to update
- Error cases:
    - 404 (Not Found) - if ID not found or invalid, or the collection is empty
    - 400 (Bad Request) - if the input parameters are invalid
    - 405 (Method not allowed) - in case of the whole collection replacement should not be allowed
- **Idempotency**: Non Idempotent

OPTIONS

- The OPTIONS method requests information about the communication options available for the target resource, at either the origin server or an intervening intermediary. This method allows a client to determine the options and/or requirements associated with a resource, or the capabilities of a server, without implying a resource action.
- Requests using the HTTP OPTIONS method should only retrieve data (server must not change its state)
- The HTTP OPTIONS method is defined as **idempotent**, which means that multiple identical OPTIONS requests should have the same effect as a single request.
- For every Rest Endpoint it is not mandatory to implement OPTIONS.
- Request and Response:

Options Request

```
OPTIONS /assets/options HTTP/1.1
Host: host.com
HTTP/1.1 200 OK
```

Options Response

```
Allow: GET,POST,PUT,PATCH,DELETE,OPTIONS
Access-Control-Allow-Origin: https://host.com
Access-Control-Allow-Methods: GET,POST,PUT,PATCH,DELETE,OPTIONS
Access-Control-Allow-Headers: Content-Type
```

HEAD

The HEAD method is identical to GET except that the server *MUST NOT* send a message body in the response (i.e., the response terminates at the end of the header section). The server *SHOULD* send the same header fields in response to a HEAD request as it would have sent if the request had been a GET. Implement only when Required.

For example, if a URL might produce a large download, a `HEAD` request could read its Content-Length header to check the file size without actually downloading the file.

REST vs RPC - When to use REST vs RPC?

RPC: Remote Procedure Call (RPC) is a protocol that one program can use to request a service from a program located in another computer on a network without having to understand the network’s details. RPC is just a bunch of functions, but in the context of an HTTP API, that entails putting the method in the URL and the arguments in the query string or body.

Recommendation: We strongly recommend developers to use REST based API calls for accessing resources. In rare cases where we want to perform actions (like aggregate, count, transform etc.,), it is recommended to use RPC style API.

See examples below.

- To retrieve the count of records from a specific resource say, "assets", the following RPC style operation is used
    - *GET: http:/<hostname>/oalapp/services/api/iboksint/assets/count?status=COMPLETE*
    - *Note that parameters are passed to filter the results. Similarly this should support pagination as well*

REST	RPC
REST is noun-centric	RPC is verb-centric
Works with resources	Works with ACTION
In the REST model, the addressable units are the entities themselves and the behaviours of the system are hidden behind the entities as side-effects of creating, updating, or deleting them	In the RPC model, the addressable units are procedures, and the entities of the problem domain are hidden behind the procedures.
REST works with HTTP Methods: GET, POST, PUT, PATCH, DELETE	RPC works with just HTTP method POST
REST API models the various entities within the problem domain as resources.	RPC style exposes applications functionality as function calls that take parameters and in turn invokes these functions.
REST uses HTTP verbs to represent transactions against these resources- POST to create, PUT to update, and GET to read. All of these verbs, invoked on the same URL, provide different functionality.	For each operation we call different functions i.e. different URL, http verbs has no role to play
Common HTTP return codes are used to convey status of the requests	Return codes are hand coded.
Example: GET : /oalapp/services/api/iboksint/assets (assets is the resource or entity)	*GET: Aggregation :/oalapp/services/api/iboksint/assets /count?status=COMPLETE*

Guidelines for RPC Style Usage

- In case of **AGGREGATION** or **DATA MANIPULATION** only, RPC style should be used.
- **No** additional 3rd party dependency should be added to incorporate RPC style.
- It is verb-centric or Action Specific. It's URL's consist of verbs like COUNT for use-case like return count of records with status as complete or DERIVE for use-case where the tax needs to be returned based on fromDate and toDate etc
    - Example: HTTP GET */oalapp/services/api/iboksint/assets/**count**?status=COMPLETE*
    - *Example: HTTP GET /oalapp/services/api/iboksint/assets/**derive-tax**?fromDate="&toDate="*
- API call is success if the http response status code is 200 OK.
- In RPC style, for certain id if record is not found then response code returned is 200 with empty response.
- **HTTP GET** method is used when the API just brings the required information and do not perform any DML operation on any tables.
- In HTTP GET, there is no request body and all inputs is part of query params.
- In case GET operation is fetching information from multiple tables like Assets and AssetLines, the dominant java entity i.e Assets forms the part of URL.
- *Example UseCase:HTTP GET /oalapp/services/api/iboksint/assets/**count**?status=COMPLETE*

Response for GET count

```
{
 status : "COMPLETE",
 count : 100
}
```

- *Example: HTTP GET /oalapp/services/api/iboksint/assets/count : It fetches Count of records with all statuses*

Response for GET count - all statuses

```
[
{
 status : "COMPLETE",
 count : 100
},
 {
status : "PENDING",
 count : 5
 }
]
```

- *Example UseCase HTTP GET: /oalapp/services/api/iboksint/assets/derive-tax?status=ACTIVE& fromDate='1-1-2020'&toDate='31-12-2020'*
- This use-case basically derives the tax for all assets with status as Active and tax calculation occurs for a duration between fromDate and toDate. It is a GET call as no updates is required on any tables. It simply fetches the details from Assets table based on the criteria and do some calculation and sends back the response.

Response of GET - DeriveTax

```
[
{ id :121,
    tax : 150000.43
  },
  { id :122,
    tax :267000.19
  }
]
```

- **HTTP POST** is used for data manipulation which includes **UPDATE** on the table or any data transformation.
- In HTTP POST, the API URL consist of the verb or action to be performed. POST call consist of request payload instead of query params.
- There can be a use-case where data is read from multiple entities and updated in more than one entity. In such scenario the HTTP POST URL is constructed in such a way that the major entity which gets updated is part of the API URL.
- Example : */oalapp/services/api/iboksint/assets/derive-tax*
- This UseCase calculates tax based on the criteria sent in request payload and also perform major update operation on the Assets entity. Thus an HTTP POST method.

Request RPC POST-DeriveTax

```
{
status:"ACTIVE",
fromDate:"1-1-2020",
toDate:"31-12-2020"
}
```

Response of POST - DeriveTax

```
[
{ id :121,
    tax : 150000.43
  },
  { id :122,
    tax :267000.19
  }
]
```

- In case of failure RestResponseEntityExceptionHandler Class will take care and sends both status code and the error message in the response same as REST.

- ALM Link for sample code of RPC style Usage:https://alm.oraclecorp.com/oal/#projects/gxp/scm/OAL-REST-Standards.git/tree/?revision=ft-agg-rpc-style