API Guidelines

# Version v1

REVISION LOG **3**

[OVERVIEW **4**](#_TOC_250031)

[RESTFUL API DESIGN GUIDELINES **4**](#_TOC_250030)

[API URL NAMING CONVENTIONS 4](#_TOC_250029)

[URI VERSIONING 4](#_TOC_250028)

[URI FORMAT 5](#_TOC_250027)

[RESOURCE NAMING CONVENTION 6](#_TOC_250026)

[MODELLING RESOURCES AND SUB-RESOURCES 6](#_TOC_250025)

[HTTP VERBS 7](#_TOC_250024)

[API PAYLOAD FORMAT 7](#_TOC_250023)

[API HEADERS 8](#_TOC_250022)

[ERROR HANDLING 8](#_TOC_250021)

[Http Error Response Codes 9](#_TOC_250020)

[RAML GUIDELINES **10**](#_TOC_250019)

[RAML FILES 11](#_TOC_250018)

[RAML HEADER 11](#_TOC_250017)

[API Titles 11](#_TOC_250016)

[Version 11](#_TOC_250015)

[Protocols 11](#_TOC_250014)

[RAML MEDIA TYPES 12](#_TOC_250013)

[Default Media Types 12](#_TOC_250012)

[RAML DOCUMENTATION 12](#_TOC_250011)

[RAML DATA TYPES 13](#_TOC_250010)

[RAML RESOURCE TYPES 14](#_TOC_250009)

[RAML TRAITS 15](#_TOC_250008)

[RAML SECURITY SCHEMES 16](#_TOC_250007)

[STRUCTURING RAML RESOURCES 17](#_TOC_250006)

[ERROR MESSAGES 18](#_TOC_250005)

[OAS 2.0 **18**](#_TOC_250004)

[GUIDELINES 19](#_TOC_250003)

[SECURITY FIRST 19](#_TOC_250002)

[DEFINITIONS 19](#_TOC_250001)

[NAMING CONVENTION 21](#_TOC_250000)

EXAMPLES 23

Revision Log

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Date | Version No. | Change Reference  to the document | Created By | Reviewed By |
| 15/09/2021 | 0.1 | Initial Draft | Purnachandrarao V |  |

# Overview

An API (Application Programing Interface) defines the bridge for communication between an application and the backend service. These interfaces provide a pre-determined set of communication “methods” along with input/output parameters for sending and receiving messages for communicating with an application over the network.

RESTful web APIs, provide a simplified approach for communication using the http protocol and REST API design principles. In this document, we would look at the guidelines and design standards to be followed for building a RESTful API. These standards will help to design APIs that are intuitive and can be easily understood and consumed by developers consuming them.

# RESTful API Design Guidelines

## API URL Naming Conventions

API URLs must follow a naming convention that meets the following criteria:

* It must provide the address or location of the server hosting the API
* It must specify the service name/capability of the API
* It must specify the version of the API service
* It must specify the resource/entity on which the service operation is being performed
* It must follow a hierarchical approach for traversing nested sub-resources Following convention must be followed for forming the API URL: *http[s]://[<server-name>]/api/[<service-name>]/[version]/[entity]*

where

* ***server-name*** - The hostname or IP given to the installed web server that is running the API.
* ***service-name*** - The API name of the service you want to access

Example: *‘https://api.<companyname>.com/api/amenities’*

## URI Versioning

Versioning is an important aspect of API design. Every API must have a version number associated with it. Hence, the URL must specify a version number to identify the version of the API. There are various approaches for versioning. But to keep things simple, it is recommended to use an integer number prefixed with a ‘v’ to denote the version of the API.

Example: *‘https://api.<companyname>.com/api/amenities/****v1****’*

## URI Format

A URI consists of segments separated by forward slashes (‘/’). Each segment must identify a resource. If a resource has sub-resources, the URI must specify the path to the sub-resource in a hierarchical manner as follows:

/resource/path/to/sub-resource

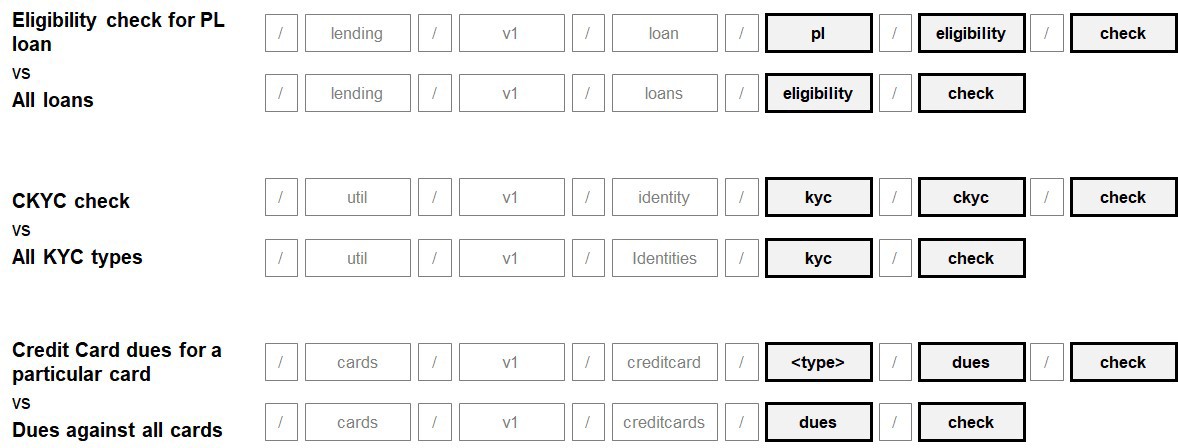
Example: *‘https://api.<companyname>.com/api/amenities/v1/****flights/UA881/cabin***

**The structure of API Taxonomy**

Diagram

Description automatically generated

**The taxonomy will use the structure defined to create logical aggregations**



## Resource Naming Convention

Every resource of an RESTful API must have a meaningful name to identify itself. It is recommended to name a resource using noun as opposed to verb or action. The URI of the resource must refer to a thing rather than an action. Also, CRUD function names should not be used in the resource names. Hence use of resource names like ‘getLoans’ to retrieve information about loans must be avoided.

A collection of resources can be named using a plural noun. Eg.

*/api/lending/v1/****loans***

Following are some of the recommended naming conventions for URI Path for a RESTful API

* Name collection resource with *plural noun*: Eg.

*https://api.abcbank.com/api/lending/v1/loans*

* Name singular resource with *singular noun* : Eg.

*https://api.abcbank.com/api/lending/v1/loans/UA123*

* Name a controller resource using a *verb* : Eg.

*https://api.abcbank.com/api/lending/v1/loans/UA123/book*

* Avoid using CRUD operation names in the URIs. For example, do not use URIs like

*https://api.abcbank.com/api/lending/v1/createLoans*

* Use lowercase for naming URIs. Avoid mixed and upper cases in URIs. Mixed case is harder to type in and read.
* Use hyphen instead of space or underline. They are aesthetic and easier to read. Spaces in URL will get transformed into URL encoded %20s, degrading readability further. For example, use URIs like *https://api.abcbank.com/api/lending/v1/loans-asset*
* Avoid using characters that require URL encoding. Eg. Spaces

## Modelling Resources and Sub-Resources

A resource can be a single instance of an object or a collection of objects. For example, a collection of loans can be represented by the object ‘/loans’. Again, a single flight within this collection can be identified by the loan number as ‘/loans/UA123’. There can be further objects that can be related to each other either as parent child or in some other ways. For example, a loan may have different status. Hence ‘status’ can be a sub-resource of a parent ‘loan’ resource and can be related as follows: /loans/UA123/status

The following approach should be followed for designing URI path with resources and related sub-resources:

|  |  |
| --- | --- |
| **URL** | **Description** |
| /servicename/v1 | This is the entry point for the API |
| /servicename/v1/{ResColName} | Resource name of a top-level collection |
| /servicename/v1/{ResColName}/{ResId} | A resource instance within the collection of  resource |
| /servicename/v1/{ResColName}/{ResId}/{SubResColName} | A sub-resource collection under the  resource ResId |
| /servicename/v1/{ResColName}/{ResId}/{SubResColName}/{SubRedId} | SubRedId inside the collection of  SubResource |

## HTTP Verbs

After identification of the resources, the next question is about the action to be performed on these resources. A HTTP verb is normally used to specify the action to be performed. It forms an important part of RESTful API design. The primary and most commonly used HTTP verb are POST, GET, PUT and DELETE. These verbs help to perform the CRUD operations on the resource. As a guideline, these verbs should be used as follows:

* **GET** – Used to retrieve or read the information about the requested resource entity identified by the request-URI
* **POST** – Used to create a new resource, which is subordinate to the parent resource identified by the request URI
* **PUT** – Used to update an existing resource entity identified by the request URI
* **DELETE** – Used to delete the resource represented by the request URI

Other HTTP verbs like PATCH, OPTIONS and HEAD can be used for specific usage requirements

## API Payload Format

The API request or response body content is referred to as the API payload. There are many options for exposing the API payload. But in the RESTful world there are 2 well adopted formats

* viz. JSON and XML. JSON is the preferred of the two due to the following benefits that it offers over XML:
  + JSON is a more compact format, meaning it weighs far less on the wire than the more verbose XML. It is a good benefit for mobile devices with a limited bandwidth to save costs and improve loading speed.
  + JSON parsing is generally faster than XML parsing.
  + JSON is easier to work with in some languages (such as javascript, python, and php)
  + Formatted JSON is generally easier to read than formatted XML.
  + JSON specifies how to represent complex datatypes, there is no single best way to represent a data structure in XML.

Due to these advantages, JSON data format should be preferred and default format used to specify the RESTful API payload. XML output can be supported based on the ‘Accept’ header specified in the request.

## API Headers

One area of REST API design that warrants attention is the use of "Media Types", which are also known as either MIME Types or Content Types. Media types have the following syntax:

*type "/" subtype \*( ";" parameter )*

REST APIs typically work with media types that fall under the "application" type. Note that parameters may follow the type/subtype in the form of attribute=value pairs that are separated by a leading semi-colon (;) character. HTTP/1.1 uses media types in the values of the ‘Accept’ and ‘Content-Type’ headers. As shown in the example below, client applications can convey their preference for a response body's media type using HTTP/1.1's ‘Accept’ request header.

*Accept: application/json,application/xml;q=0.9,text/html;q=0.8,\*/\*;q=0.7*

In the ‘Content-Type’ header of an HTTP/1.1 request or response, a media type reference indicates the "type" associated with the message body's byte sequence. The example below demonstrates a Content-Type header value that references a media type with a "charset" parameter:

*Content-type: application/json; charset=ISO-8859-4*

REST APIs use either the "application/json" or the "application/xml" media type in the ‘Content- Type’ header of an HTTP/1.1 request or response.

## Error Handling

Communicating error information properly to the API consumer is critical for the success of the REST API. API consumers and app developers using APIs learn to write code through errors.

Well defined error messages are helpful for troubleshooting and resolving issues after the applications built using the APIs are in the hands of the end users.

RESTful APIs must communicate error information using proper HTTP response status code. There are different types of HTTP response status codes to communicate the different success and error information as follows:

* + ***2xx: Success*** – Used to communicate that the request from the client was successfully received, understood, and accepted
  + ***3xx: Redirection*** – Used to communicate that additional action needs to be taken by the user agent like browser in order to fulfil the request
  + ***4xx: Client Error*** - Used to indicate errors caused by the client
  + ***5xx: Server Error*** – Used to indicate that server is aware that an error occurred while processing the request and cannot process it further

### Http Error Response Codes

The following table summarizes the HTTP error response codes that may be returned by the API under different error scenarios

|  |  |  |
| --- | --- | --- |
| 400 | Bad Request | Indicates that the request had some mal-formed syntax error due to which it could not be understood by the server. Probable |
|  |  | reason could be missing mandatory parameters or syntax error |
| 401 | Unauthorized | Indicates that the request could not be authorized possibly due |
|  |  | to missing or incorrect authentication token information |
| 403 | Forbidden | Indicates that the request was understood by the server but it |
|  |  | could not be processed due to some policy violation or the client |
|  |  | does not have access to the requested resource |
| 404 | Not Found | Indicates that the server did not find anything matching the Request-URI |
| 405 | Method Not Allowed | Indicates that the method specified in the Request-Line is not |
|  |  | allowed for the resource identified by the Request-URI |
| 408 | Request Timeout | The server did not receive a complete request message within the time that it was prepared to wait. |
| 409 | Conflict | Indicates that the request could not be processed due to a |
|  |  | conflict with the current state of the resource |
| 414 | Request URI Too Long | Indicates that the Request URI length is longer than the allowed |
|  |  | limit for the sever |
| 415 | Unsupported Media Type | Indicates that the request format is not supported by the server |
| 429 | Too Many Request | Indicates that the client has submitted the request too often and needs to slow down |
| 500 | Internal Server Error | Indicates that the request could not be processed due to an |
|  |  | unexpected error in the server. |
| 501  502  503  504 | Not Implemented  Bad Gateway    Service Unavailable  Gateway Timeout | Indicates that the server does not support the functionality  required to fulfil the request  Indicates that the server while acting as a gateway or proxy received an invalid response from the backend server  Indicates that the server is currently unable to process the request due to temporary overloading or maintenance of the server. Trying the request at a later point of time might result in success  Indicates that the server while active as a gateway or proxy did not receive a timely response from the backend server |
|  |  |  |

Along with the Http error status code, the response message must also provide additional information to clarify the error. The following payload format can be used to communicate additional information about the error:

*{*

*“status” : {status [optional]}, “code” : {code [optional]} , “message”: “Error message”, "errors": [optional]*

*[*

*{ "code": {error code}, "message": "Error message"*

*}*

*]*

*}*

Extra fields can be added as needed. The errors array is an optional attribute, which will often be used when the service captures multiple errors to return to the consumer

Example for error message payload is as follows:

*{*

*“status” : 400,*

*“code” : 40010,*

*“message”: “SMS message body is not specified”, "errors": [*

*{ "code": 1, "message": "SMS message body is required" } ,*

*{ "code": 2, "message": "SMS recipient is required" }*

*]*

*}*

# RAML Guidelines

RAML is an API design language that allows developers to take advantage of the full API Design Lifecycle, meaning that they can visually design their APIs, test them, and get user feedback without ever having to write a single line of code.

RAML describes APIs in a human readable format - plain text. In addition to better readability, RAML supports two well-used developer best practices: design patterns and code reuse. Rather than having to rely on human verification to verify that each API is consistent, RAML provides a variety of “syntactic sugars” that ensures each API is complete and concise while minimizing actual development time.

## RAML Files

RAML file names are in lowercase, alphanumeric characters using kebab-case (i.e. hyphen- separated).

* For example:
  + acme-banking-api.xml
  + account-example.raml
  + acme-bank-doc.raml

## RAML Header

### API Titles

The title of the API should be a short, plain-text label for the API.

* For example:

title: ACME Banking API

### Version

Each API should have a integer version number.

* For example:

version: v1

### Protocols

The *optional* protocols node specifies the protocols that an API supports. If the protocols node is not explicitly specified, the protocol in the baseUri node is used.

* For example:

protocols: [HTTPS]

## RAML Media Types

### Default Media Types

Specifying the OPTIONAL mediaType node sets the default media type for responses and requests that have a body.

* For example:

mediaType: application/json

**NOTE:** If mediaType is set at the root level, the media type within each body definition does not need to be specified.

**NOTE:** Explicitly defining a mediaType node for a body of an API request or response overrides the default media type.

## RAML Documentation

The documentation node provides a concise, human-friendly description of the API. All documentation should be formatted using GitHub-Flavored Markdown (https://help.github.com/categories/writing-on-github/) as a separate file and included under the documentation tag.

* For example (acme-bank-doc.raml):

#%RAML 1.0 DocumentationItem title: ACME Banking API Home content: |

\*\*ACME Banking API\*\* enables developers to build applications that make use of the information from resource methods implemented in the API.

This API contains functionality that allows developers to retrieve and manipulate

\_customer\_, \_account\_ and \_transaction\_ information. Check out the [API Portal]() for more details.

The documentation node within the API is used to refer to external Documentation Item: documentation:

* !include documentation/acme-bank-doc.raml

## RAML Data Types

RAML DataTypes provide a concise and powerful way of describing the information utilized within the API. Data types add rules for validating data against a type declaration. DataTypes should be created as separate files and included at the start of the API RAML file.

* + For example (account.raml):

#%RAML 1.0 DataType type: object properties:

accountID: string accountType:

enum: [Checking, Savings, Overdraft Savings, Credit Card] accountNumber: string

accountOwner:

type: array

items: !include account-owner.raml accountBalance: !include money.raml IBAN:

type: string

pattern: ^[A-Z]{2,2}[0-9]{2,2}[a-zA-Z0-9]{1,30}$

bank: !include bank.raml interestRate?:

type: number format: double createdAt: datetime modifiedAt?: datetime

The types node within the API RAML is used to refer to external DataTypes: types:

customer: !include datatypes/customer.raml

account: !include datatypes/account.raml transaction: !include datatypes/transaction.raml

DataType declarations can also be described using schemas:

* + For JSON, use JSON Schema
    - JSON Schema files should use the .schema.json suffix
  + For XML, use XML Schema (XSD)
* XML Schema files should use the .xsd suffix

## RAML Resource Types

ResourceTypes are a powerful way to reuse patterns across multiple resources and methods. The patterns provided by ResourceTypes encourages consistency and reduces complexity for API designers and consumers.

Usage of ResourceTypes is *optional* but is highly encouraged since it simplifies the API design and improves readability. ResourceTypes should be created as separate files and included at the start of the API RAML file.

* + For example (collection.raml):

#%RAML 1.0 ResourceType post?:

description: Add a new <<resourcePathName | !singularize>> displayName: Add new <<resourcePathName | !singularize>> body:

type: <<resourcePathName | !singularize | !uppercamelcase>> responses:

201:

headers:

Location:

description: URL of the new <<resourcePathName | !singularize>> information example: /<<resourcePathName>>/8f19cb50-3f57-4d38

body:

202:

description: The request has been accepted for processing. Use the URI provided in the Location header of the response to monitor the status.

headers:

Location:

example: /<<resourcePathName>>/1234/status 503:

body:

type: <<customErrorDataType>>

The resourceTypes node within the API RAML is used to refer to external ResourceTypes: resourceTypes:

collection: !include resourceTypes/collection.raml

member: !include resourceTypes/member.raml

Resource types can be mapped in the API RAML by referencing the assigned type:

/customers: type:

collection:

customErrorDataType: customErrorMessage

## RAML Traits

A Trait, like a method, can provide method-level nodes such as description, headers, query string parameters, and responses. Methods that use one or more traits inherit nodes of those traits.

Usage of Traits, like ResourceTypes, is *optional* but is highly encouraged since it simplifies the API design and improves readability. Traits should be created as separate files and included at the start of the API RAML file.

* + For example (cacheable.raml):

#%RAML 1.0 Trait

usage: Apply this trait to any GET method that supports caching control. responses:

200:

headers:

Cache-Control:

description: |

Activates caching and defines cache behavior through cache response directives. Usually defines public or private (cacheable by proxy or not) and max-age for resource. See <http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html>for more information. example: private, max-age=31536000

Expires:

description: |

Sets a date in RFC 1123 format from which the cached resource should no longer be considered valid.

If both the Expires header and max-age in the Cache-Control header are set, max-age will take precedence.

type: datetime

example: Tue, 18 Apr 2017 09:30:41 GMT format: rfc2616

The traits node within the API RAML is used to refer to external Traits: traits:

cacheable: !include traits/cacheable.raml

Traits can be mapped in the API RAML by referencing the assigned trait:

/customers:

get:

is: [ cacheable ]

description: Retrieve a list of customers displayName: Get all customers securedBy: oauth2\_0

## RAML Security Schemes

Each authentication pattern supported by the API must be expressed as an element of the securitySchemes node value. The security schemes should be created under a separate folder and included within the API RAML file.

* + For example (oauth2.raml):

#%RAML 1.0 SecurityScheme

type: OAuth 2.0

description: Apply the OAuth 2.0 security policy to resource methods for authenticating API requests

describedBy:

headers: Authorization:

description: |

Used to send a valid OAuth2 access token. Do not use with the "access\_token" query. string parameter.

type: string queryParameters:

access\_token:

description: |

Used to send a valid OAuth2 access token. Do not use together with the "Authorization" header.

type: string responses:

401:

description: |

Bad or expired token. This can happen if the API consumer uses a revoked or expired access token. To fix, you should re-authenticate the user.

403:

description: |

Bad OAuth request (wrong consumer key, bad nonce, expired timestamp...).

Unfortunately, re-authenticating the user won't help here. settings:

authorizationUri: https://placeholder.com/oauth2/authorize accessTokenUri: https://placeholder.com/oauth2/access\_token authorizationGrants: implicit

The securitySchemes node within the API RAML is used to refer to external security schemes: securitySchemes:

oauth2\_0: !include securitySchemes/oauth2.raml

**NOTE**: The default security schema for all resources can be defined by adding the securedBy tag at the root level. Each method can override the default security scheme by having its own securedBy tag.

## Structuring RAML Resources

A RAML file may reference additional resources, like traits and resourceTypes, JSON Schemas, or example data. API authors are encouraged to follow certain conventions for structuring and naming external resources. According to these conventions, the following relative URL paths and file suffixes should be used:



|  |  |  |  |
| --- | --- | --- | --- |
| Resource | Path | Suffix | Example |
| API Definition | / | .raml | acme-banking.raml |
| Trait | /traits/ | .raml | cacheable.raml |
| Resource Types | /resourceTypes/ | .raml | collection.raml |
| Security Schemes | /securitySchemes/ | .raml | oauth2.raml |
| Data Types | /dataTypes/ | .raml | account.raml |
| .schema.json | account.schema.json |
| .xsd | account.xsd |
| Documentation | /documentation/ | .raml | acme-bank-doc.xml |
| Examples | /examples/ | .raml | account-example.raml |
| .json | account-example.json |
| .xml | account-example.xml |

References between these resources of the RAML API definition should be made by relative URLs.

## Error Messages

Every API should use a common error message schema for when an error needs to be returned in a response payload:

The common error type schema contains these attributes:

* + code – The HTTP Status code of the response.
  + reason – The human-readable text of the equivalent HTTP status code.
  + service – Name of the API returning the response.
  + detail – A detailed message describing the problem in relation to the error type in human-readable form.
  + transactionID – The unique ID associated with the transaction in which this API is involved (also known as a correlation ID).
  + invocationTimestamp – The time at which the API was invoked (*optional*).

{

"code": "201",

"reason": "Created",

"service": "Reporting Process API", "detail": "Report Generated Successfully.",

"transactionID": "c70332d0-260f-11e7-98a1-c4b301c8ce30", "invocationTimestamp": "2017-10-02T17:42:38Z"

}

# OAS 2.0

Swagger™ is a project used to describe and document RESTful APIs. The Swagger specification defines a set of files required to describe such an API. These files can then be used by the Swagger-UI project to display the API and Swagger-Codegen to generate clients in various languages. Additional utilities can also take advantage of the resulting files, such as testing tools.

https://swagger.io/specification/v2/

All work which include API creation should follow a design first approach. Yaml format should be followed.

## Guidelines

1. Clear and easily readable by architects, analysts, developers
2. Well documented, with explanations provided in description tags
3. Adhere to OpenAPI specifications - v2.0 at the time of this writing
4. Use a design which lends itself to a clean and easily consumable object model

## Security First

Often API designers focus on functionalities and add security later on. We would encourage to follow security first approach so that security is considered for every endpoint during the design.

OAuth 2.0 implementation should be created at the API manager level on the wrapper class or proxy creation

## Definitions

In the specification, models define what would be the request/response body for each endpoint. There are two different places to define models. Flattened/scattered in each endpoint or extracted into the definitions.

When the model definition scattered in each endpoint, there is no name and the same model might be duplicated in several endpoints. To avoid duplications, it is best to extract common data objects into the definitions section.

definitions:

Order:

type: object

properties:

id:

type: integer format: int64

petId:

type: integer format: int64

quantity:

type: integer format: int32

shipDate:

type: string

format: date-time

status:

type: string

description: Order Status

enum:

- placed

- approved

- delivered

complete:

type: boolean

default: **false**

xml:

name: Order

With Order is defined, you can put $ref in each point for object definition. Here is an example response that utilizes the Order definition.

/store/order:

post:

tags:

- store

summary: Place an order for a pet

description: ''

operationId: placeOrder

produces:

- application/xml

- application/json

parameters:

- in: body

name: body

description: order placed for purchasing the pet

required: true

schema:

$ref: '#/definitions/Order'

responses:

'200':

description: successful operation

schema:

$ref: '#/definitions/Order'

'400':

description: Invalid Order

With Array

/user/createWithArray:

post:

tags:

- user

summary: Creates list of users with given input array

description: ''

operationId: createUsersWithArrayInput

produces:

- application/xml

- application/json

parameters:

- in: body

name: body

description: List of user object

required: true

schema:

type: array

items:

$ref: '#/definitions/User'

responses: default:

description: successful operation

## Naming Convention

As definition name is translated into Java class name or any other scripting language, it is better to follow the naming convention of Java or that language.

Elements should always start with an upper-case letter, to respect class definitions in generated code, which always start with an upper-case letter.

properties:

error:

$ref: 'org/APIERR0001/1.0.1/errors.yaml#/error'

...

vs

...

properties:

error:

$ref: 'org/APIERR0001/1.0.1/errors.yaml#/Error'

Elements should use only alpha-numeric characters and avoid underscores, @ signs or others. OpenAPI general guidelines recommend alpha-numeric only, and, while these would generate correct programming language code, it would break accepted programming guidelines.

"@nameID":

$ref: "org/APIDOMAIN0001/0.0.5/elementDefs.yaml#/NameID"

"@accessCode":

$ref: "org/APIDOMAIN0001/0.0.5/elementDefs.yaml#/Access\_Code"

...

filterList:

type: array

items:

$ref: "org/APIDOMAIN0001/0.0.5/elementDefs.yaml#/APIFilter\_Type"

...

vs

...

nameID:

$ref: "org/APIDOMAIN0001/0.0.5/elementDefs.yaml#/NameID"

accessCode:

$ref: "org/APIDOMAIN0001/0.0.5/elementDefs.yaml#/AccessCode"

...

filterList:

type: array

items:

$ref: "org/APIDOMAIN0001/0.0.5/elementDefs.yaml#/APIFilterType"

#### Object definitions are to be avoided from the declaration of Body elements

The objects should be moved to the *definitions* section of the specification or in an external document, for *shared definitions*.

The Body should not contain any declarations as in “type: object”.

#### Original version:

...

selection:

description: Identifies the selection to retrieve information for. Only one of the child elements of this structure are to be provided.

type: object

properties:

id:

$ref: "org/APIDOMAIN0001/0.0.1/elementDefs.yaml#/ID"

card:

$ref: "org/APIDOMAIN0001/0.0.1/elementDefs.yaml#/AlternativeID"

...

#### Updated version:

...

selection:

$ref: "#/definitions/Selection"

...

definitions:

# Selection structures.

Selection:

type: object

description: Identifies the selection to retrieve information for. Only one of the child elements of this structure are to be provided.

properties:

id:

$ref: "org/APIDOMAIN0001/0.0.2/elementDefs.yaml#/ID"

card:

$ref: "org/APIDOMAIN0001/0.0.2/elementDefs.yaml#/AlternativeID"

...

#### Implicit object definitions are to be avoided from the declaration of Body elements when used from collection declarations

An object should be defined in the *definitions* section of the specification or an external document, for *shared definitions*, and be referenced from the collection in the Body, instead of the declaration of an implicit object.

The Body should not contain any collection declarations with implicit object definitions

#### Ex.: names element: Original version:

...

names:

description: Contact information.

type: array items:

properties:

"id":

$ref: "org/APIDOMAIN0001/0.0.1/elementDefs.yaml#/ID"

name:

description: A name can contain up to two lines of name information.

type: array

items:

$ref: "org/APIDOMAIN0001/0.0.1/elementDefs.yaml#/Name"

...

#### Updated version:

...

names:

type: array

items:

$ref: "#/definitions/RetrieveNames"

...

definitions:

RetrieveNames:

description: Contact information

type: object

properties:

id:

$ref: "org/APIDOMAIN0001/0.0.2/elementDefs.yaml#/ID"

name:

description: A name can contain up to two lines of name information.

type: array

items:

$ref: "org/APIDOMAIN0001/0.0.2/elementDefs.yaml#/Name"

...

## Examples

Swagger specification gives you an opportunity to define an example response for each endpoint so that your API consumer can easily understand what would be expected when the endpoint is accessed.

Once you have examples defined, the generated project can be built and started with mock responses for consumers to start their work immediately without waiting for the provider to complete the API implementation.