API Design Guidelines

Objective

The objective of this page is to document what makes a good API design for RBS group private banking APIs.

Please read this guideline document in addition to the RBS group wide API design and guidelines draft.

All samples and screenshots given in this document has followed Open API Specification. However the guidelines are also applicable if developers write their specification in RAML.

Prerequisite

What is API and what are the benefits of an API first strategy (#ToDo : link to learning page)

What are Restful APIs. (#ToDo : link to learning page)

Why OpenAPI(Swagger)

How to use OpenAPI Specification to define an API. (#ToDo : link to learning page)

History

Name	Version	Date	Comment
Avik Sengupta	0.1 Draft	9.10.2018	Initial Draft

Key concepts

Key terms

Resource

The fundamental concept in any RESTful API is the resource. A resource is an object with a type, associated data, relationships to other resources.

In core banking terms, these can be customers, accounts etc.

Sub-Resource

These are resources, but in a hierarchy, which has one or more parent resources. For instance, 'direct-debits' can be a sub resource of 'accounts' resource.

It's imperative that all published APIs will be restful in design and have the correct usage of nouns and HTTP verbs (Restful API)

Verbs

Verbs are actions on resource. Verbs signal the intent of the requester in terms of what the request wants to do with the Resource.

These are standard HTTP Verbs of GET, POST, PUT, PATCH, DELETE among others.

Correct way to combine the above

For an easy understanding use this structure for every resource:

Resource	GET read	POST create	PUT update	DELETE
/cars	Returns a list of cars	Create a new car	Bulk update of cars	Delete all cars
/cars/711	Returns a specific car	Method not allowed (405)	Updates a specific car	Deletes a specific car

Do not use verbs in defining resource:

```
/getAllCars
/createNewCar
/deleteAllRedCars
```

Correct use of verbs and idempotency (Idempotency)

<u>Don’t Misuse Safe Methods</u>

Safe methods are HTTP methods which return the same resource representation irrespective of how many times are called by client. GET, HEAD, OPTIONS and TRACE methods are defined as safe, meaning they are only intended for retrieving data and should not change a state of a resource on a server.

Use HTTP methods according to the action which needs to be performed.

Use **PUT, POST** and **DELETE** methods instead of the **GET** method to alter the state.
Do not use **GET** for state changes:

GET cars/711/activate ✗

GET cars/711/delete ✗

DELETE cars/711 ✓

Pluralization of Resources

Do not mix up singular and plural nouns. Keep it simple and use only plural nouns for all resources.

```
/cars instead of /car
/users instead of /user
/products instead of /product
/settings instead of /setting
```

Return Representation

POST, PUT or PATCH methods, used to create a resource or update fields in a resource, should always return updated resource representation as a response with appropriate status code as described in further points.

POST if successful to add new resource should return HTTP status code 201 along with URI of newly created resource in Location header (as per HTTP specification)

POST	Create	201 created	New Resource
PUT/PATCH	Update	200 success	Modified Resource
DELETE	Delete	204 no content/200 success	empty/message
GET	Retrieval	200 success	Resource/ResourceList
Not Found	Any	404 not found	Error Resource
Client error	Any	4xx	Error Resource
Server error	Any	5xx	Error Resource

API Granularity

Use one API to expose one resource. *Accounts API* ✓ *Direct Debit and Standing Order API* ✗

Sub-Resources should also be added in their own API specification, and not added

However, the uri path must reflect the hierarchy

URI Structure

Protocol	host name	versioned resource	path identifier	optional query params
https://	api.coutts.com/	accounts/v2/	{id}	?name='x'
https://	api.coutts.com/	accounts/v2/{id}/direct-debits/	n/a	n/a

The *basepath* should always be declared as the common path of an API to avoid duplication.

Open API specification 3.0.0, new element 'servers' is replacing basepath

Versioning

Version Structure: major.minor.patch

no	***Description***
Major	*Version upgrade with major change of functionality.*
Minor	*Minor Improvements with no loss of backward compatibility*
Patch	*Defect fixes with no change in API specification. Beneficial for testing and issue tracking.*

Version in Uri

Use only major version number in the Uri of the base path.

Example:

api.couttsch.com/v1/customer/{id} ✓

api.couttsch.com/v1.1.3/customer/{id} ✗

Automatically overwrite minor and patch releases with previous versions.

Tolerant Reader

Very rarely a single release of API (or for that matter any software) can address the needs for all future needs. It's usually more effective to deliver small incremental pieces of functionality over the time. Unfortunately, this introduces the possibility of the introducing breaking changes for API consumers as elements are added, deleted or changed.

The best way to achieve that is to be guided by the design principal of Tolerant Reader pattern. The API should be happy to accept more than it needs (within limitation) so that if needs be, it can just ignore additional information that it doesn't support instead of throwing an error back. The API should also send the bare minimum in it's response. Similarly, at the other side of the spectrum, an API consumer should design to take the minimum it needs and ignores the rest.

(Martin Fowler : Tolerant Reader)

Support for multiple versions

Support two or three major versions of the API in the production environment. This ensures plenty of time for consumers to adopt a new backwardly-incompatible version. APIs should be versioned through the addition of a version number in the URL root. See URL construction.

Deprecation Policies

Set clear API deprecation policies so you’re not supporting old client applications forever.

State how long users have to upgrade, and how you’ll notify them of these deadlines.

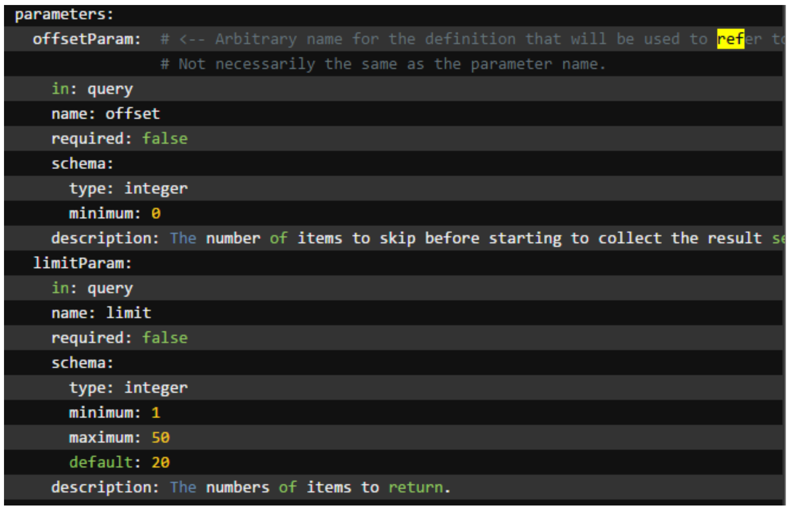
- Use the list of primary technical contacts for each API consumer to contact them directly via email. Specify exactly when the API version will be demised.
- Also announce deprecation in HTTP responses using a ‘Warning’ header.
- Announce on the Portal that the version is/will be deprecated.

Parameters

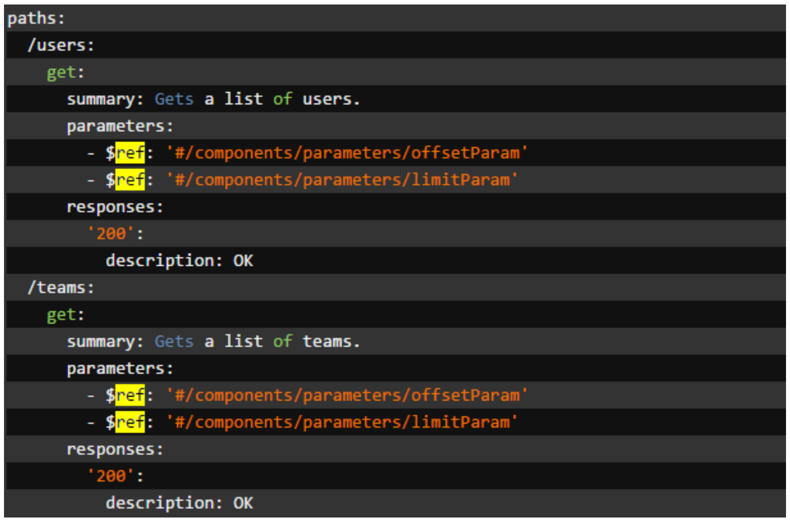
Global

Different API paths may have common parameters, such as pagination parameters.

You can define common parameters under parameters in the global section and reference them elsewhere via **`$ref`**.



```
parameters:
  offsetParam:  # <-- Arbitrary name for the definition that will be used to refer t
                # Not necessarily the same as the parameter name.
    in: query
    name: offset
    required: false
    schema:
      type: integer
      minimum: 0
    description: The number of items to skip before starting to collect the result s
  limitParam:
    in: query
    name: limit
    required: false
    schema:
      type: integer
      minimum: 1
      maximum: 50
      default: 20
    description: The numbers of items to return.
```



```
paths:
  /users:
    get:
      summary: Gets a list of users.
      parameters:
        - $ref: '#/components/parameters/offsetParam'
        - $ref: '#/components/parameters/limitParam'
      responses:
        '200':
          description: OK
  /teams:
    get:
      summary: Gets a list of teams.
      parameters:
        - $ref: '#/components/parameters/offsetParam'
        - $ref: '#/components/parameters/limitParam'
      responses:
        '200':
          description: OK
```

Paths and Query Params

Path parameters should be solely used to identify the resource, ie. they are the resource identifier.

For everything else, use query params, i.e they are the filter parameters.

Each route Uri should uniquely identify and resolve the operation endpoint without causing any route ambiguity.

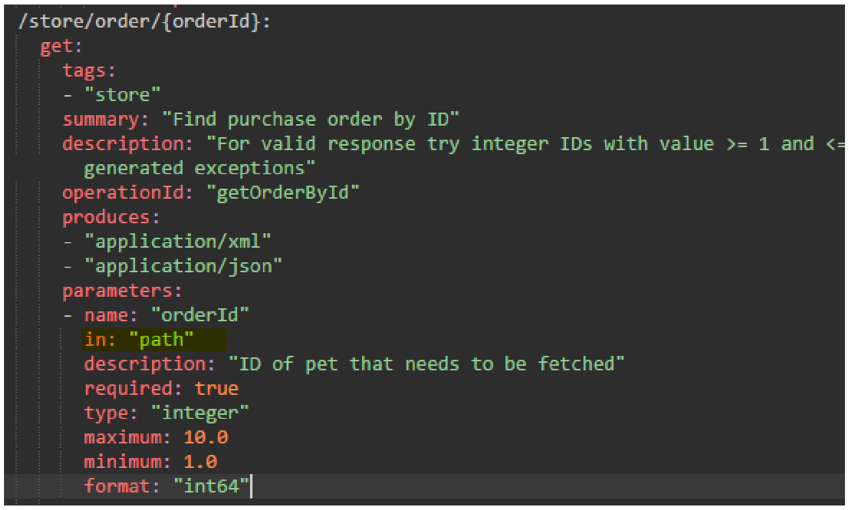
Each route Url should be should be self explanatory to API consumers, that is discoverability of a resource by looking at the URL should be very apparent.

/store/order/{orderId} *resourceIdentifier* ✓

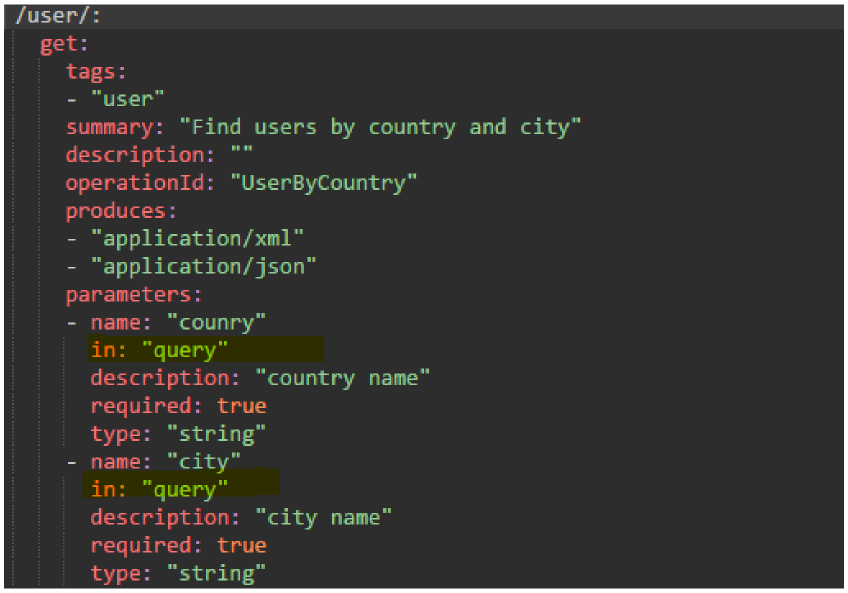
/user?country=country-name&city=city-name *filter params* ✓

/store/order?orderId=id ✗

/user/{country-name}/{city-name} ✗



```
/store/order/{orderId}:
  get:
    tags:
    - "store"
    summary: "Find purchase order by ID"
    description: "For valid response try integer IDs with value >= 1 and <=
      generated exceptions"
    operationId: "getOrderById"
    produces:
    - "application/xml"
    - "application/json"
    parameters:
    - name: "orderId"
      in: "path"
      description: "ID of pet that needs to be fetched"
      required: true
      type: "integer"
      maximum: 10.0
      minimum: 1.0
      format: "int64"
```



```
/user/:
  get:
    tags:
    - "user"
    summary: "Find users by country and city"
    description: ""
    operationId: "UserByCountry"
    produces:
    - "application/xml"
    - "application/json"
    parameters:
    - name: "counry"
      in: "query"
      description: "country name"
      required: true
      type: "string"
    - name: "city"
      in: "query"
      description: "city name"
      required: true
      type: "string"
```

Schema/Data Model

Use JSON Schema Specification to define data model.

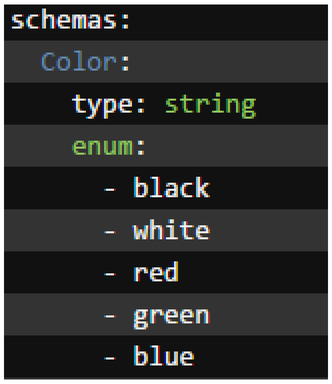
Specify data type and format for properties.

Specify Regex pattern for properties where applicable using the pattern *element*

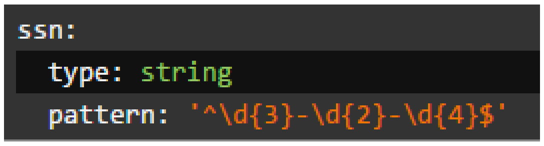
Specify min and max values where applicable.

Specify Enums with valid values where applicable.

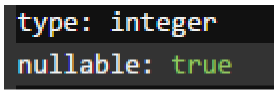
Use Object Hierarchy when applicable.



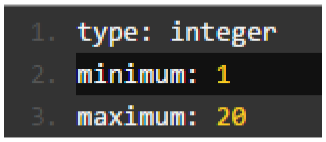
```
schemas:
  Color:
    type: string
    enum:
      - black
      - white
      - red
      - green
      - blue
```



```
ssn:
  type: string
  pattern: '^\d{3}-\d{2}-\d{4}$'
```



```
type: integer
nullable: true
```



```
type: integer
minimum: 1
maximum: 20
```

Sorting, Filtering and Pagination

When retrieving a large list of data over network, computational costs associated with retrieval or the networking cost of passing it over to the caller can be catastrophic.

Following best practices must be used to avoid that.

- Use relevant query parameters which can run a filter at the source of data in order to limit the number of items returned.
- Use a query parameter to limit the number of resources returned for a resource type with a default and a max value.
- Use a query parameter to determine the starting position of the resources to be returned.
- Use a combination of above two to get a certain number of items.
- Use a query parameter of type enum to define the sorting options (including sorting direction, asc/dsc) available with a default value.

<u>Examples</u>

- */users?country=country-name (Return first N sorted by default attribute)*
- */users?country=country-name&limit=20 (Return first 20 sorted by default attribute)*
- */users?country=country-name&limit=20&offset=20 (Return records 21-40 sorted by default attribute)*
- */users?country=country-name&limit=20&offset=20&sort=user-name-asc (Return records 21-40 sorted by user name in ascending order)*

Naming Conventions

Use Plural for resource names.

accounts, customers etc.

Use forward slash (/) to indicate a hierarchical relationships

http://api.couttsch.com/accounts

http://api.couttsch.com/accounts/{id}/direct-debits

Use hyphens (-) on word break in path and parameters to improve the readability of URIs

http://api.couttsch.com/business-partners/{id}?

http://api.couttsch.com/business-partners?country-name=x

Use lowercase letters in URIs

RFC 3986 defines URIs as case-sensitive, and a mixed case can cause hard to find defects in the system.

it is always best to avoid ambiguity with casing and stick to lower case.

Resource/Type names in Pascal and resource properties in camel case.

Camel case: Camel case (stylized as camelCase; also known as camel caps or more formally as medial capitals) is the practice of writing compound words or phrases such that each word or abbreviation in the middle of the phrase begins with a capital letter, with no intervening spaces or punctuation.

Pascal case: Pascal case is a subset of the above camel case where the first letter is also capitalized.

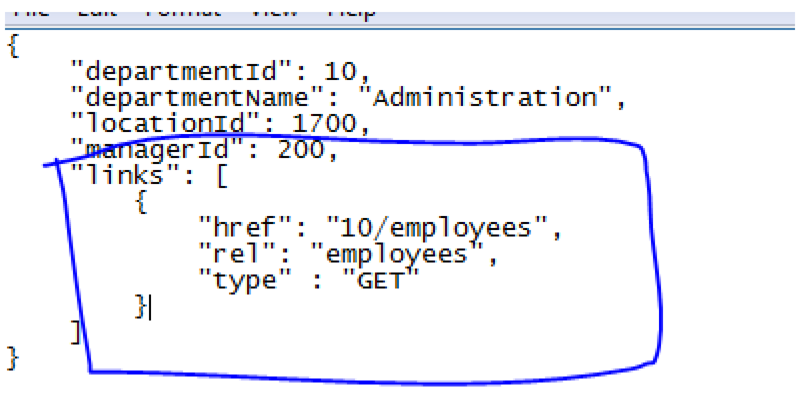
```
Position:
  properties:
    posId:
      type: integer
    posType:
      type: string
    name:
      type: string
```

HATEOS

Hypermedia as the Engine of Application State is a principle that hypertext links should be used to create a better navigation through the API.

HATEOAS (Hypermedia as the Engine of Application State) is a constraint of the REST application architecture that keeps the RESTful style architecture unique from most other network application architectures. The term “hypermedia” refers to any content that contains links to other forms of media such as images, movies, and text.

This architectural style lets you use hypermedia links in the response contents so that the client can dynamically navigate to the appropriate resource by traversing the hypermedia links. This is conceptually the same as a web user navigating through web pages by clicking the appropriate hyperlinks in order to achieve a final goal.



```
{
    "departmentId": 10,
    "departmentName": "Administration",
    "locationId": 1700,
    "managerId": 200,
    "links": [
        {
            "href": "10/employees",
            "rel": "employees",
            "type" : "GET"
        }
    ]
}
```

Documentation

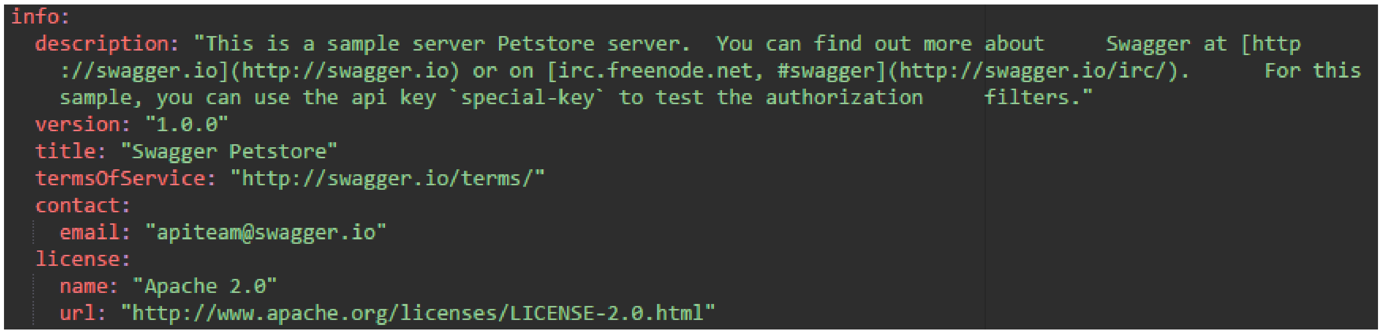
It's imperative to add correct and adequate documentation in the API definition, so that it doesn't need be supported with additional documents.

Info

Use *info* section to add basic API meta.

Use markup to add *description* to improve readability.

Use a distribution list rather than an individual's detail for email.



```
info:
  description: "This is a sample server Petstore server.  You can find out more about     Swagger at [http
    ://swagger.io](http://swagger.io) or on [irc.freenode.net, #swagger](http://swagger.io/irc/).      For this
    sample, you can use the api key `special-key` to test the authorization     filters."
  version: "1.0.0"
  title: "Swagger Petstore"
  termsOfService: "http://swagger.io/terms/"
  contact:
    email: "apiteam@swagger.io"
  license:
    name: "Apache 2.0"
    url: "http://www.apache.org/licenses/LICENSE-2.0.html"
```

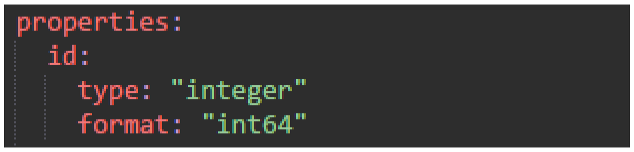
Method/Operations

Use summary to add a headline against the method

Use Description to add as many details as you can. This can include the authentication and authorization mechanism needed to call this method as well describing the method itself.

Parameters and properties

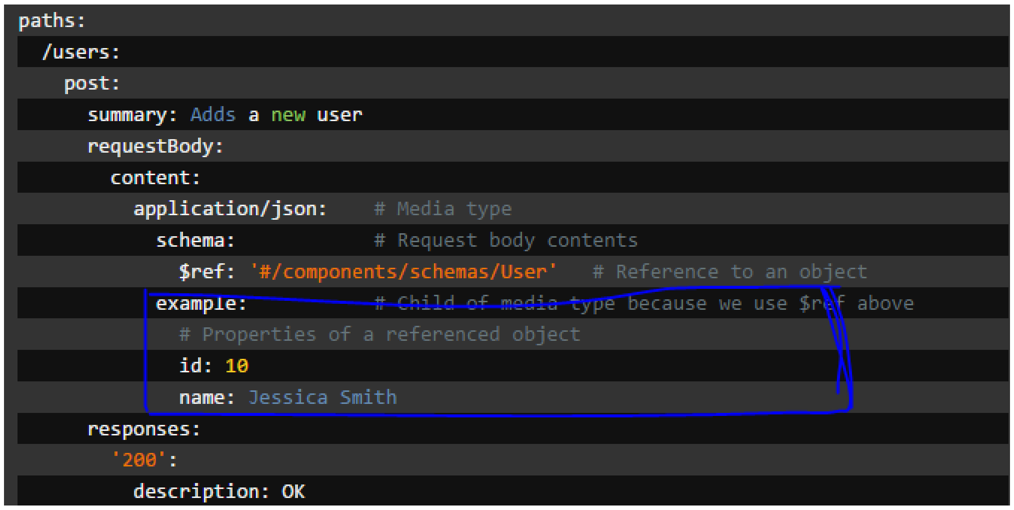
Use accurate data type and format where possible



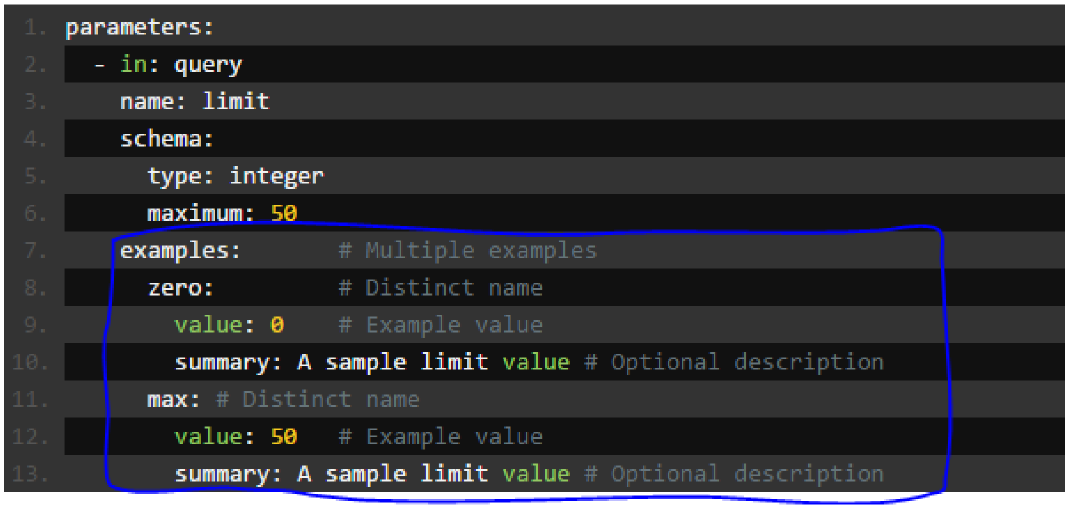
```
properties:
  id:
    type: "integer"
    format: "int64"
```

Examples

Use examples to request body, parameters, properties and objects to help developer on boarding.



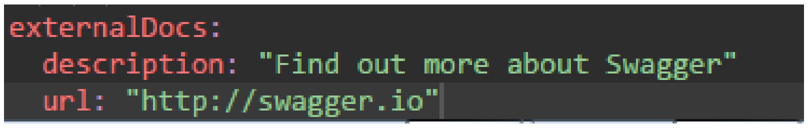
```
paths:
  /users:
    post:
      summary: Adds a new user
      requestBody:
        content:
          application/json:     # Media type
            schema:             # Request body contents
              $ref: '#/components/schemas/User'   # Reference to an object
            example:            # Child of media type because we use $ref above
              # Properties of a referenced object
              id: 10
              name: Jessica Smith
      responses:
        '200':
          description: OK
```



```yaml
parameters:
  - in: query
    name: limit
    schema:
      type: integer
      maximum: 50
    examples:       # Multiple examples
      zero:         # Distinct name
        value: 0    # Example value
        summary: A sample limit value # Optional description
      max: # Distinct name
        value: 50   # Example value
        summary: A sample limit value # Optional description
```

External Docs

It is also possible to refer to external documentation for additional reading.



```yaml
externalDocs:
  description: "Find out more about Swagger"
  url: "http://swagger.io"
```