API Conventions

Naming conventions

- PATHs should use nouns rather than verbs, as RESTful APIs should be used to access resources, not to trigger workflows.

  Example:
  ```
  POST /deals/{deal_id}/acknowledgement
  (rather than /deals/{deal_id}/acknowledge)
  ```

- Use snake_case in PATHs (mainly because URLs are not case sensitive).

  Example:
  ```
  /checkout/v3/orders/{order_id}
  ```

- Attributes / Parameters will use snake_case

  Example:
  ```
  parameters:
          - name: website_url
  ```

- Object names in definitions will be CamelCase

  Example:
  ```
  definitions:
   OfferStatus:
      type: object
      properties:
        id:
          type: string
          description: Id of the offer.
        status:
          type: string
          description: Description of the offer.
  ```

- Response data should always be wrapped in object

  Example:
  ```
  responses:
   "200":
    description: list of offers' ids
    schema:
     type: object
     properties:
      data:
       type: array
       items:
         $ref: '#/definitions/OfferUser'
  ```

Path conventions:

- Use in-URL parameters where appropriate (e.g. /offers/{offer-id})
- Wherever possible, make sure path parameters refer to the immediately preceding group of objects, and have other parameters you want in the URL as query parameters. This makes it clearer what the ID relates to when you see the actual URL.

Example:

```
/cats/{cat_id}?owner_id={owner_id}
```

- basePath field must be included: *"The base path on which the API is served, which is relative to the host. If it is not included, the API is served directly under the host. The value MUST start with a leading slash (/)"* (*http://swagger.io/specification/#swaggerObject*)

Example:

```
swagger: '2.0'
info:
  title: eu.operando.core.pfb
  description: Operando PfB module.
  version: "1.0.0"
  license:
    name: MIT
    url: http://opensource.org/licenses/MIT
host: localhost:8080
basePath: /operando/core/pfb
```

POST-ing new objects / PUT-ing existing objects:

- When accessing objects by IDs, duplication of IDs in *Path*, Query, or *Body* of the API call shall be avoided.
- If ID is defined in the object definitions, and also is required to be specified in *Path, Query,* or *Body* then a new version of the object, without the ID attribute, shall be specified for this API call.

Example:

POST and GET requests to /regulations/ with different object definitions.

```
/regulations/:
    post:
      summary: Create a new legislation entry in the database.
      parameters:
        - name: regulation
          in: body
          description: The first instance of this regulation document
          required: true
          schema:
            $ref: 'https://.../definitions/PrivacyRegulation.yaml#/PrivacyRegulationRequest'
/regulations/{reg_id}/:
    get:
      summary: Read a given legislation with its ID.
      parameters:
        - name: reg_id
          in: path
          description: Unique identifier representing a specific user
          type: integer
          format: int64
          required: true
      responses:
        '200':
          description: The regulation document requested to be read is returned in full
          schema:
            $ref: 'https://.../definitions/PrivacyRegulation.yaml#/PrivacyRegulation'
```

PrivacyRegulation.yaml

```
PrivacyRegulation:
    type: object
    description: |
        A privacy rule that reflects a given privacy legislation as described
        by a particular set of laws in a given jurisdiction.
    properties:
      reg_id:
```

```
          type: integer
          format: int64
      legislation_sector:
        type: string
            action:
        type: string
        description: The action being carried out on the data
        ...
PrivacyRegulationRequest:
    type: object
    description: |
        A privacy rule that reflects a given privacy legislation as described
        by a particular set of laws in a given jurisdiction.
    properties:
      legislation_sector:
        type: string
            action:
        type: string
        description: The action being carried out on the data
        ...
```

Storage:

- All API specifications should be stored in the **op-api-doc** repo in GitHub, in the folder for the relevant work package.

Definitions:

- Definitions of object models should be stored in the definitions directory of the above repository. This makes it easier to reuse definitions across multiple APIs, keeping them consistent even if they change in the future.
- Each module should have one definition file containing multiple object specifications.
- Definitions files shall be named:

Example:
eu.operando.definitions.{module name}

```
eu.operando.definitions.pdb.yaml
eu.operando.definitions.ude.yaml
```

General comments:

- Add tags to ease navigation in documentation
- Combine API calls into single call with multiple optional parameters where possible

Example:
Single search API instead of multiple endpoints (/offers/offers_by_osp , /offers/offers_by_uid, etc…)

```
/offers/search:
    get:
      summary: Search offers in databse
      parameters:
        - name: website_url
          in: query
          description: URL of the website to get offers related to specific URL
          type: string
        - name: website_id
          in: query
          description: ID of the website to get offers related to specific website ID
          type: string
        - name: osp_id
          in: query
          description: ID of the OSP to list of all offers created by specific OSP.
          type: string
        - name: user_id
          in: query
          description: ID of the user to limit offers to those apllicable to specific user.
          type: string
```

Valid data type format for Swagger:

Common Name	type	format	Comments
integer	integer	int32	signed 32 bits
long	integer	int64	signed 64 bits
float	number	float	
double	number	double	
string	string		
byte	string	byte	base64 encoded characters
binary	string	binary	any sequence of octets
boolean	boolean		
date	string	date	As defined by full-date - RFC3339
dateTime	string	date-time	As defined by date-time - RFC3339
password	string	password	Used to hint UIs the input needs to be obscured.

Example:

```
expiration_date:
        type: string
        format: date-time
        description: Date when the offer expires.
```

Adherence to HTTP verbs:

Use appropriate HTTP verbs for each action.

Verb	Description
HEAD	Can be issued against any resource to get just the HTTP header info.
GET	Used for retrieving resources.
POST	Used for creating resources.
PATCH	Used for updating resources with partial JSON data. For instance, an Issue resource has `title` and `body` attributes. A PATCH request may accept one or more of the attributes to update the resource. PATCH is a relatively new and uncommon HTTP verb, so resource endpoints also accept `POST` requests.
PUT	Used for replacing resources or collections. For `PUT` requests with no `body` attribute, be sure to set the `Content-Length` header to zero.
DELETE	Used for deleting resources.

HTTP Verb	CRUD	Entire Collection (e.g. /customers)	Specific Item (e.g. /customers/{id})
POST	Create	201 (Created), 'Location' header with link to /customers/{id} containing new ID.	404 (Not Found), 409 (Conflict) if resource already exists.
GET	Read	200 (OK), list of entities. Use pagination, sorting and filtering to navigate big lists.	200 (OK), single entity. 404 (Not Found), if ID not found or invalid.

HTTP Verb	CRUD	Entire Collection (e.g. /customers)	Specific Item (e.g. /customers/{id})
PUT	Update/Replace	404 (Not Found), unless you want to update/replace every resource in the entire collection.	200 (OK) or 204 (No Content). 404 (Not Found), if ID not found or invalid.
PATCH	Update/Modify	404 (Not Found), unless you want to modify the collection itself.	200 (OK) or 204 (No Content). 404 (Not Found), if ID not found or invalid.
DELETE	Delete	404 (Not Found), unless you want to delete the whole collection—not often desirable.	200 (OK). 404 (Not Found), if ID not found or invalid.

- POST

  The POST verb is most-often utilized to **create** new resources. In particular, it's used to create subordinate resources. That is, subordinate to some other (e.g. parent) resource. In other words, when creating a new resource, POST to the parent and the service takes care of associating the new resource with the parent, assigning an ID (new resource URI), etc.

  On successful creation, return HTTP status 201, returning a Location header with a link to the newly-created resource with the 201 HTTP status.

  POST is neither safe nor idempotent. It is therefore recommended for non-idempotent resource requests. Making two identical POST requests will most-likely result in two resources containing the same information.

  Examples:

  - *POST http://www.example.com/customers*
  - *POST http://www.example.com/customers/12345/orders*

- GET

  The HTTP GET method is used to **read** (or retrieve) a representation of a resource. In the “happy” (or non-error) path, GET returns a representation in XML or JSON and an HTTP response code of 200 (OK). In an error case, it most often returns a 404 (NOT FOUND) or 400 (BAD REQUEST).

  According to the design of the HTTP specification, GET (along with HEAD) requests are used only to read data and not change it. Therefore, when used this way, they are considered safe. That is, they can be called without risk of data modification or corruption—calling it once has the same effect as calling it 10 times, or none at all. Additionally, GET (and HEAD) is idempotent, which means that making multiple identical requests ends up having the same result as a single request.

  Do not expose unsafe operations via GET—it should never modify any resources on the server.

  Examples:

  - *GET http://www.example.com/customers/12345*
  - *GET http://www.example.com/customers/12345/orders*
  - *GET http://www.example.com/buckets/sample*

- PUT

  PUT is most-often utilized for **update** capabilities, PUT-ing to a known resource URI with the request body containing the newly-updated representation of the original resource.

  However, PUT can also be used to create a resource in the case where the resource ID is chosen by the client instead of by the server. In other words, if the PUT is to a URI that contains the value of a non-existent resource ID. Again, the request body contains a resource representation. Many feel

this is convoluted and confusing. Consequently, this method of creation should be used sparingly, if at all.

Alternatively, use POST to create new resources and provide the client-defined ID in the body representation—presumably to a URI that doesn't include the ID of the resource (see POST below).

On successful update, return 200 (or 204 if not returning any content in the body) from a PUT. If using PUT for create, return HTTP status 201 on successful creation. A body in the response is optional—providing one consumes more bandwidth. It is not necessary to return a link via a Location header in the creation case since the client already set the resource ID.

PUT is not a safe operation, in that it modifies (or creates) state on the server, but it is idempotent. In other words, if you create or update a resource using PUT and then make that same call again, the resource is still there and still has the same state as it did with the first call.

If, for instance, calling PUT on a resource increments a counter within the resource, the call is no longer idempotent. Sometimes that happens and it may be enough to document that the call is not idempotent. However, it's recommended to keep PUT requests idempotent. It is strongly recommended to use POST for non-idempotent requests.

Examples:

- *PUT http://www.example.com/customers/12345*
- *PUT http://www.example.com/customers/12345/orders/98765*
- *PUT http://www.example.com/buckets/secret_stuff*

- PATCH

  PATCH is used for **modify** capabilities. The PATCH request only needs to contain the changes to the resource, not the complete resource.

  This resembles PUT, but the body contains a set of instructions describing how a resource currently residing on the server should be modified to produce a new version. This means that the PATCH body should not just be a modified part of the resource, but in some kind of patch language like JSON Patch or XML Patch.

  PATCH is neither safe nor idempotent. However, a PATCH request can be issued in such a way as to be idempotent, which also helps prevent bad outcomes from collisions between two PATCH requests on the same resource in a similar time frame. Collisions from multiple PATCH requests may be more dangerous than PUT collisions because some patch formats need to operate from a known base-point or else they will corrupt the resource. Clients using this kind of patch application should use a conditional request such that the request will fail if the resource has been updated since the client last accessed the resource. For example, the client can use a strong ETag in an If-Match header on the PATCH request.

  Examples:

  - *PATCH http://www.example.com/customers/12345*
  - *PATCH http://www.example.com/customers/12345/orders/98765*
  - *PATCH http://www.example.com/buckets/secret_stuff*

- DELETE

  On successful deletion, return HTTP status 200 (OK) along with a response body, perhaps the representation of the deleted item (often demands too much bandwidth), or a wrapped response (see Return Values below). Either that or return HTTP status 204 (NO CONTENT) with no response body. In other words, a 204 status with no body, or the JSEND-style response and HTTP status 200 are the recommended responses.

  HTTP-spec-wise, DELETE operations are idempotent. If you DELETE a resource, it's removed. Repeatedly calling DELETE on that resource ends up the same: the resource is gone. If calling DELETE say, decrements a counter (within the resource), the DELETE call is no longer idempotent. As mentioned previously, usage statistics and measurements may be

updated while still considering the service idempotent as long as no resource data is changed. Using POST for non-idempotent resource requests is recommended.

There is a caveat about DELETE idempotence, however. Calling DELETE on a resource a second time will often return a 404 (NOT FOUND) since it was already removed and therefore is no longer findable. This, by some opinions, makes DELETE operations no longer idempotent, however, the end-state of the resource is the same. Returning a 404 is acceptable and communicates accurately the status of the call.

Examples:

- *DELETE http://www.example.com/customers/12345*
- *DELETE http://www.example.com/customers/12345/orders*
- *DELETE http://www.example.com/bucket/sample*

References:

- https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol
- http://www.restapitutorial.com/lessons/httpmethods.html
- https://en.wikipedia.org/wiki/List_of_HTTP_status_codes
- http://swagger.io/specification/