Every team is accountable to respect these standards during API development, and every member is encouraged to work collaboratively with members of the API Community of Practice to improve these guidelines.

In the case of an evolution of these API standards, the following rules must be followed:

- Existing APIs should not be systematically changed, although this is advisable,
- New APIs must respect the current version of the standards.

This document uses the terms **RIGHT**, **SHOULD**, **MUST**, **MUST** as keywords to define the requirement levels of a specification as defined in RFC 2119.

Table des matières

Basic principles

This section covers the basic principles.

API First

Design API before implementation

The signature of the API – also called interface or contract – **MUST** be done before implementation (OpenAPI specification, Stub, etc).

The goal is to allow stakeholders to give early feedback and to show self-discipline by focusing on:

- knowledge of the functional domain and the common requirements,
- entities and business resources, i.e. avoid having APIs for specific use-cases,
- a clear divide between the WHAT and the HOW

The API contract is the unique source of truth, not the implementation. If your development language does not support automatic creation of documentation, you **SHOULD** write the documentation manually.

The implementation of an API **MUST** always be consistent with its description : it represents the contract between the API and the consumers.

Compatibility

Do not break backward compatibility

API updates in the same major version **MUST NOT** break backward compatibility. An API is a contract between the consumers and the producer which cannot be broken by unilateral decisions.

There are two ways to update an API without breaking it:

- follow the compatible extension rules,
- introduce a new version of the API while maintaining the previous versions.

Rules to extand an API

Each new minor version of an API **MUST** follow these extension rules:

- **MUST NOT** remove fields/properties,
- **MUST NOT** make mandatory fields that were initially described as optional,

- **MUST NOT** delete an existing endpoint,
- Every new addition to a minor version **MUST** be optional.

If, for any reason, these rules cannot be followed, then a new major version **MUST** be deployed.

Documentation

General documentation

An API **MUST** be documented in a Wiki and **MUST** contain at least the following:

- a full description
- the team in charge of the API
- a link to the swagger documentation

This Wiki page **MUST** be added to our internal directory.

Nota Bene : Our API community of practice is aware that this solution is temporary. In the future, we will have a centralised directory such as an API Management.

Documentation

An API **MUST** provide a full, explicit and up-to-date documentation of its endpoints and **SHOULD** expose it as a Swagger.

REST

Resources instead of Verbs

APIs **MUST** be designed around resources and **MUST** not represent actions. An API **MAY** include hypermedia (HATEOAS).

Maturity level

Ideally, we are aiming for Richardson's second maturity level, however it is possible to use level 3. Further information is available on https://martinfowler.com/articles/richardsonMaturityModel.html.

REST is based on entities/resources and usage of standard HTTP methods (such as GET/POST/PUT/DELETE) as operations. URLs **MUST** contain names and no verb.

For example, instead of having the verb *cancel* in the URL, it is preferrable to use the resource *cancellation*.

Use of verbs

Standard HTTP methods are not meaningless: they **MUST** be used to specify the type of action required.

Although these methods are not equivalent to CRUD, it is preferable, in our case, to use them as they are for simplification purposes and to keep only non idempotent creations.

Method	Action	Definition	
POST	Non-idempotent	Create a resource	C
GET	Nullipotent (Safe)	Get one (or many) resource(s)	R
PUT	Idempotent	Update a ressource	U
DELETE	Idempotent	Delete a resource	D

POST

- A POST (create, in our case) successfully executed will return a 201. The header **MUST** contain `Location` with a link to the newly created entity.
- Asynchronous operations **MUST** return a 202 containing a header `Location` in order to monitor the operation.

GET

- A successful GET returns a resource and a 200.
- A successful GET returns multiple resources and a 200 if all resources are present or a 206 if some of the resources are returned (paging, top n). In this case, the response **MUST** contain a `Content-Range` header.

PUT

- A successful PUT (update, in our case) returns a 200 or a 204.
- Asynchronous operations **MUST** return a 202 containing a `Location` header to monitor the status of the operation.

DELETE

- A successful DELETE returns a 200 or a 204.
- Asynchronous operations **MUST** return a 202 containing a `Location` header to monitor the status of the operation.

Message

This section covers governance about the structure of messages.

Data and description

Encoding

Data **SHOULD** be encoded in UTF-8.

Enums

Data **SHOULD** be displayed as enumerations rather than cryptic codes. Also, enumeration positions **SHOULD** be serialized as `camelCase` characters to avoid mapping errors.

```
Content-type: application/x.va.validation+json
{
    // No ambiguity
    "title":"baron"

    // Risk of mapping error
    "title": 4
}
```

Data and display

When a property can be conveyed either as raw data or as data ready to be displayed, the API **SHOULD** state it clearly.

```
Content-type: application/x.va.validation+json
{
    // By default, it is data
    "myDateTime": "1997-09-02T19:20:30.45+01:00",

    // Is it long enough ? Explain when it is a displayable property
    "myDateTimeDisplay": "Monday 2 September at 7pm 20mn 30sec",

    "myDate": "1985-08-09", // By default, it is data
    "myDateDisplay": "Vendredi 9 août 1985", // Explain when it is displayable (and the bir

    "gender":"M",
    "genderDisplay":"Male"
}
```

Booleans

Booleans properties name **MAY** be prefixed by `is` or `has` in order to make it intuitive.

Identifiers

For security reasons, technical identifiers **SHOULD** be non-sequential and non-deterministic, e.g., `UUID` v4 RFC-4122.

Identical representation of business data

The API **SHOULD** be based on identical representation of business data. For more information, have a look at our Représentation communes des données business (Internal link).

Business validations

Format of business validations

When a request fails because of business validations, it **SHOULD** respond a 422 HTTP code, **SHOULD** have the following `Content-Type`

`Content-type: application/vnd.va.validation+json`

and **SHOULD** return this kind of payload

```
{
    "validations": [
        {
            // Field translated according the i18n/l10n and visible to the user
            "display": "The name is required",

            // ValidationCode used to configure the label
            "code":"validationRequired",

            // Related field(s)
            "fields":["firstName"],

            // Variable value constraint (Validation property)
            "valParams":{}
        },
        {
            // Field translated according the i18n/l10n and visible to the user
            "display": "Le npa devrait comporter au moins 42 caractères",
```

```
            // ValidationCode used to configure the label
            "code":"validationMinLength",

            // Related field(s)
            "fields":["address[0].npa"],

            // Variable value constraint (Validation property)
            "valParams":{
                "min": 42
            }
        },
        {
            ...
        }
    ]
}
```

Business errors

Structure of business errors

When a business operation fails, the response status **MUST** be in the range of 4XX, `Content-Type` **SHOULD** be

```
Content-type: application/vnd.va.error+json
```

and the payload **SHOULD** be similar to

```
{
    // Technical field
    "message": "This message will not be displayed to the user",

    // i18n/l10n field which can be displayed to the user
    "display": "If this error occurs again, please call your mama!",

    // Standard error code used client-side to define a specific label to display

    "code":"uniqueErrorCodeForDoesNotWork"
}
```

Exception

Exception structure

On production environments, software exceptions **MUST** return an HTTP status code 500 and **MUST NOT** return a stack trace.

On non-production environments, payloads **SHOULD** be similar to

```
Content-type: application/vnd.va.exception+json
{
    // Usual technical fields
    "message": "object not set to an instance",
    "stackTrace": "...",
    "innerException": {...}
}
```

JSON Payload

Format - content negociation

Payloads **SHOULD** be returned in the `application/json` format and **MUST** comply with its conventions (`camelCase`, etc). A webservice **MAY** process other formats (such as xml, yml) via the standard `Accept` header.

JSON'ception

Properties contained in a JSON **MUST NOT** contain JSON or XML themselves.

Request

This section covers query standards (i.e. filter, paging, sorting, asynchronism, etc).

Asynchronism

During an operation conducted asynchronously by the server, the server **MUST** return an HTTP code 202 with a header Location giving the location of the URL to follow the operation. This URL will point to a resource of type operations.

```
Location: https://VaHappyHi:8081/v2/operations/8156ab4e
```

The `operation` resource **SHOULD** contain the current state of the operation (`notStarted`, `running`, `succeeded`, `failed`).

- If the status is `notStarted` or `running`, then the return code **MUST** be 202 and the header location remains the same,
- If the status is `notStarted` or `running`, then the header Retry-After **SHOULD** indicate the number of seconds to wait before checking the status of the operation,
- If the state is `succeeded`, then the return code **MUST** be 200 and the header location should now return the location of the resource in question.

Impersonation

The impersonation implementation **SHOULD NOT** be implemented only at customer level, but **SHOULD** be at API level. Impersonation **SHOULD** be done using a custom header:

```
Va-Impersonate: sio
```

The API **SHOULD** log the fact that the action was performed by user A impersonating user B.

JSON Patch

The update of an object can be done via an http request `PUT`. In addition, use of 'PATCH' is possible using the operations described in RFC-6902 (**JavaScript Object Notation (JSON) Patch**).

We **SHOULD** only use the add, remove and replace operations. Other operations described in the RFC **SHOULD NOT** not be used.

```
if an object is
{ firstName:"Albert", contactDetails: { phoneNumbers: [] } };

and we apply the following operations:
[
  { op:"replace", path:"/firstName", value:"Joachim" },
  { op:"add", path:"/lastName", value:"Wester" },
 { op:"add", path:"/contactDetails/phoneNumbers/0", value: { number:"555-123" } }
];

The object MUST be transformed into
{ firstName:"Joachim", lastName:"Wester", contactDetails: { phoneNumbers: [{number:"555-123"}]
```

Warning, it has been noted that the swagger may not be generated correctly. In this case, it **MUST** contain a textual description describing that it is a json-patch operation and what type of object it accepts.

Localisation

The desired language **SHOULD** be set using the `Accept-Language` header.

Please note that the content of the JSON payload as well as the parameters transmitted in the URL **MUST** be formatted according to the JSON standard.

Exemple

HTTP Request

```
GET /contracts HTTP/1.1
Accept-Language: fr-ch, de-ch
```

HTTP Response

```
HTTP/1.1 200 OK
Content-Type: [...]
Content-Language: fr-ch


[...]
```

Paging

Access to data lists **MUST** support paging for a better consumer experience. This is true for all lists that are potentially larger than a few hundred records.

There are two types of iteration techniques:

- Offset/Limit-based,
- Cursor-based.

It is important to take into account the way pagination is used by the consumers. It seems that direct access to a specific page is less used than navigation via links of the type *next page/previous page*. Therefore, it is better to favour *cursor-based* pagination.

In all cases, we **MUST** start pagination at 0.

Nomenclature

This section covers standards linked to naming of resources, URIs, …

Global rules

Naming conventions

APIs **MUST** be developed in english, **MUST NOT** contain acronyms and **MUST** use 'camelCase' convention (unless otherwise specified).

Glossary

Field names **MUST** come from our business glossary (internal link), or be based on AFA's glossary (Specific Insurance Link).

URI

Each URI **MUST** follow the Standard naming conventions, except for 'camelCase'. Instead, a hyphen - **SHOULD** be used for compound words. Furthermore a URI **MUST NOT** end with a slash `/`.

Examples

```
// Returns all people
GET https://MyHappyApi:8081/v2/people
// Returns person d8a0f1ed
GET https://MyHappyApi:8081/v2/people/d8a0f1ed

// Returns a list of children resources 'home-in-one' for person d8a0f1ed
GET https://MyHappyApi:8081/v2/people/d8a0f1ed/home-in-one
/// Returns the child resource 'home-in-one' 587d038d for person d8a0f1ed
GET https://MyHappyApi:8081/v2/people/d8a0f1ed/home-in-one/587d038d

// Returns current config
GET https://MyHappyApi:8081/v2/configuration
// Returns config for person d8a0f1ed
GET https://MyHappyApi:8081/v2/people/d8a0f1ed/configuration
```

Versioning

The version of the API **SHOULD** be specified right after the server root segment and **MUST** match the first - *major* - digit from the semantic version.

`https://MyHappyApi:8081/v2/...`

Furthermore for non production environments, the latest version **COULD** be exposed through a *latest* segment, i.e.

`https://MyHappyApi:8081/latest/...`

Protocol

This section addresses issues related to the protocol and its standards.

HTTP

HTTP Protocol

All APIs **MUST** support the HTTP protocol and its semantics.

HTTP Codes

Some rules for the use of HTTP codes, the API developer

- **MUST NOT** invent new HTTP codes or derive from their original meaning,
- **MUST** provide high quality documentation when using HTTP codes not listed below.

2XX Success

The request was processed successfully.

Code	Definition
200 OK	Succès de la requête
201 Created	Resource created successfully
202 Accepted	Request accepted but not completed (asynchronous process…)
204 No content	Request successful, empty response
206 Partial	Résultat partiel (voir pagination)

4XX Client Errors

The request contained an error from the consumer.

Code	Definition
400 Bad request	The request is not valid (syntax, size, …)
401 Unauthorized	The client is not authenticated
403 Forbidden	The customer does not have the necessary rights
404 Not found	The requested resource does not exist
416 Range Not Satisfiable	Range Not Satisfiable
418 I'm a teapot	A request for coffee was sent to a teapot
422 Business validation	A request failed due to a business validation error

Note: in the case of an empty collection, the result must be a 200 returning an empty array. The 404 is not appropriate since, although empty, the collection exists.

5XX Server Errors

The server couldn't process the request.

Code	Definition
500 Internal server error	An unexpected exception occurred.

TLS

An API using the protocol HTTP **SHOULD** use HTTPS.

Operations

This section covers standards linked to operations.

Environments

An API **MUST** be deployed to a QA (also called UAT) environment before being pushed to production.

If more environments are required, an API developer **SHOULD** follow existing DNS naming conventions (internal link) to name environments.

Monitoring

Monitoring API consumption

The team in charge of an API running in a production environment **SHOULD** ensure it is being monitored.

Health check

An API **SHOULD** expose an endpoint to check its health status

```
{
  "name": "Va.Api.Business.MyAwesomeProduct",
  "status": "up",
  "dependencies": {
    "Va.Api.Tech.Dependency1": {
      "depth": 1,
      "status": "up"
    },
    "Va.Api.Tech.SubDependency": {
      "depth": 2,
      "status": "up"
```

```
    }
  }
}
```

Furthermore, continous integration tools **COULD** use the healthcheck endpoint to confirm that the API is running correctly.

Dependencies

In non-production environments, an API **SHOULD** expose an endpoint to list Vaudoise library dependencies being used.

```
{
  "product": "Va.XCut.Back.Actuators.Core",
  "version": "1.0.0.13490",
  "libraries": [
    {
      "name": "Va.XCut.Api.Template.Application",
      "product": "Va.XCut.Api.Template",
      "version": "0.0.0.13490",
      "informationalVersion": "0.0.0",
      "configuration": "Debug"
    },
    {
      "name": "Va.XCut.Back.Logger.Std",
      "product": "Va.XCut.Back.Logger.Std",
      "version": "1.0.0.13490",
      "informationalVersion": "1.0.0-Beta01",
      "configuration": "Debug"
  ]
}
```

Hosting

In non production environments, an API **SHOULD** expose an endpoint to give basic information about the hosting server.

```
{
  "machineDomain": "VAUDOISE",
  "machineName": "DEVABCDEF",
  "machineOS": "Microsoft Windows 10.0.10240 ",
  "machineProcessorCount": 8,
  "environmentName": ".NET Core 4.6.26606.02",
  "environmentArchitecture": "x64",
  "serviceName": "Va.XCut.Api.Template.Application",
  "serviceProcessId": 8752,
```

```
  "serviceStartTime": "2018-07-05T07:29:44.4771925+02:00",
  "serviceMemory": 92827648,
  "serviceThreads": 21
}
```