Chaque équipe est responsable du respect de ces standards pendant les développements d'API et chaque membre est encouragé à faire évoluer ces guidelines de manière collaborative, avec les membres de la communauté de pratique API.

Dans le cas d'une évolution de ces standards API, il faut suivre les règles suivantes:

- les API existantes ne doivent pas être systématiquement changées, bien que cela soit conseillé,
- les nouvelles API doivent respecter la version courante des standards.

Ce présent document utilise les termes **DOIT**, **DEVRAIT**, **PEUT** comme mots-clés pour définir le niveaux d'exigence d'une spécification telle que définit dans la RFC 2119 (version française).

Terme	Définition
DOIT	Ce mot, ou les termes "**OBLIGATOIRE**" ou "**DEVRA**", veut dire que la définition est une exigence absolue de la spécification.
NE DOIT PAS	Cette phrase, ou la phrase "**NE DEVRA PAS**", veut dire que la définition est une interdiction absolue de la spécification.

Terme	Définition
DEVRAIT	Ce mot, ou l'adjectif "**RECOMMANDÉ**", veut dire qu'il peut exister des raisons valides dans des circonstances particulières pour ignorer un item particulier, mais les répercussions doivent être comprises et soigneusement évaluées avant de choisir un cours différent.

Terme	Définition
NE DEVRAIT PAS	Cette phrase, ou la phrase “**NON RECOMMANDÉE**” veut dire qu’il peut exister des raisons valides dans des circonstances particulières quand le comportement spécifique est acceptable ou même utile, mais les répercussions complètent devraient être comprise et le cas soigneusement évalué avant d’implémenter tout comportement décrit avec cette étiquette.

Terme	Définition
PEUT	Ce mot ou l'adjectif "**OPTIONNELLE**", signifie qu'un item est vraiment optionnel. Un vendeur peut choisir d'inclure l'item parce qu'une place de marché spécifique l'exige ou parce que le vendeur pressent que cela améliore le produit alors qu'un autre vendeur peut omettre le même item. Une implémentation qui n'inclut pas une option particulière **DOIT** être préparée à inter-opérer avec une autre implémentation qui n'inclut pas l'option, même peut-être avec une fonctionnalité réduite. Dans la même veine, une implémentation qui inclut vraiment une option particulière **DOIT** être préparée à

Terme	Définition

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.

Requête

Cette section couvre les standards liés aux requêtes (i.e. filtre, pagination, tri, asynchronisme, etc).

Asynchronisme

Lors d'une opération conduite de manière asynchrone par le serveur, celui-ci **DOIT** retourner un code HTTP 202 avec un header Location désignant l'emplacement de l'URL de suivi de l'opération. Cet URL pointera sur une ressource de type operations.

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

La ressource `operation` **DEVRAIT** contenir l'état actuel de l'opération (`notStarted`, `running`, `succeeded`, `failed`).

- Si l'état est `notStarted` ou `running`, alors le code de retour **DOIT** être 202 et le header location reste le même,
- Si l'état est `notStarted` ou `running`, alors le header Retry-After **DEVRAIT** indiquer le nombre de secondes à attendre avant de vérifier l'état de l'opération,

- Si l'état est `succeeded`, alors le code de retour **DOIT** être 200 et le header location doit désormais retourner l'emplacement de la ressource en question.

Impersonation

L'implémentation de l'impersonation **NE DEVRAIT PAS** être implémentée uniquement au niveau du client, mais **DEVRAIT** être au niveau de l'API. L'impersonation **DEVRAIT** se faire au moyen d'un header custom:

```
Va-Impersonate: sio
```

L'API **DEVRAIT** logguer le fait que l'action a été effectuée par un utilisateur A impersonant un utilisateur B.

JSON Patch

La modification d'un objet peut se faire via une requête http `PUT`. De plus, l'utilisation du `PATCH` est possible via la description d'opérations telles que décrites par la RFC-6902 (**JavaScript Object Notation (JSON) Patch**).

On **DEVRAIT** se limiter aux opérations add, remove, replace. Les autres opérations décrites dans la RFC ne DEVRAIENT pas être utilisées.

```
si un objet est
{ firstName:"Albert", contactDetails: { phoneNumbers: [] } };

et que l'on applique la suite d'opérations suivantes:
[
  { op:"replace", path:"/firstName", value:"Joachim" },
  { op:"add", path:"/lastName", value:"Wester" },
 { op:"add", path:"/contactDetails/phoneNumbers/0", value: { number:"555-123" } }
];

L'objet DOIT être transformé en
{ firstName:"Joachim", lastName:"Wester", contactDetails: { phoneNumbers: [{number:"555-123"}]
```

Attention, il a été constaté que le swagger peut ne pas être généré correctement. Dans ce cas, il **DOIT** contenir une description textuelle décrivant qu'il s'agit d'une opération json-patch et quel type d'objet elle reçoit.

Localisation

La langue désirée **DEVRAIT** être définie en utilisant le header `Accept-Language`.

A noter que le contenu de la payload JSON ainsi que les paramètres transmis dans l'URL **DOIVENT** être formatés selon le standard JSON.

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


[...]
```

Pagination

L'accès à des listes de données **DOIT** supporter la pagination pour une meilleure expérience du côté du consommateur. Cette affirmation est vraie pour toutes les listes qui sont potentiellement plus grandes que quelques centaines d'enregistrements.

Il existe deux types de techniques d'itération:

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

Il est important de prendre en compte la façon dont est utilisée la pagination c/o les consommateurs. Il semblerait que l'accès direct à une page spécifique est bien moins utilisé que la navigation via des liens de type *page suivante/page précédente*. De ce fait, il vaut mieux favoriser la pagination de type *cursor-based*.

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/...`

Protocole

Cette section traite des problématiques au niveau du protocole et de ses standards.

HTTP

Protocole HTTP

Toutes les API **DOIVENT** supporter le protocole HTTP et sa sémantique.

Codes HTTP

Quelques règles pour l'utilisation des codes HTTP, le développeur d'API

- **NE DOIT PAS** inventer des nouveaux codes HTTP ou dériver de leur sens originel,
- **DOIT** fournir une documentation de qualité lors de l'utilisation de codes HTTP non-listés ci-dessous.

2XX Success

La requêtes a été traitée avec succès.

Code	Definition
200 OK	Succès de la requête
201 Created	Resource créée avec succès
202 Accepted	Requête acceptée mais non complétée (process asynchrone…)
204 No content	Succès de la requête, réponse vide
206 Partial	Résultat partiel (voir pagination)

4XX Client Errors

La requête contenait une erreur de la part de l'appelant.

Code	Definition
400 Bad request	La requête n'est pas valide (syntaxe, taille,…)
401 Unauthorized	Le client n'est pas authentifié
403 Forbidden	Le client ne possède pas des droits nécessaires
404 Not found	La ressource demandée n'existe pas
416 Range Not Satisfiable	Range Not Satisfiable
418 I'm a teapot	Une requête de café a été envoyé à une théière
422 Business validation	Un échec de la requête est dû à une erreur de validation métier

Remarque: dans le cas d'une collection vide, le résultat se doit d'être un 200 retournant un tableau vide. Le 404 n'est pas approprié puisque, bien que vide, la collection existe.

5XX Server Errors

Le serveur n'a pas pu traiter la requête.

Code	Definition
500 Internal server error	Une exception inattendue s'est produite

TLS

Une API utilisant le protocole HTTP **DEVRAIT** utiliser l'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
}
```