MINISTÈRE
DE L'ÉCONOMIE,
DES FINANCES
ET DE LA SOUVERAINETÉ
INDUSTRIELLE ET NUMÉRIQUE
Liberté
Égalité
Fraternité

Agence pour l'informatique financière de l'État

Guide de bonnes pratiques pour la conception et le développement d'API

Destinataires :
- Direction AIFE
- Responsables de département
- HFD

Pour information :
- Membres du département SIFE

Historique

DATE	PRENOM NOM	STATUT / SUIVI DES MODIFICATIONS
20/09/2018	AIFE	V1.00/ Initialisation du document
05/10/2018	AIFE	V1.07/ Version initiale
05/08/2019	AIFE	V1.08/ Complément sur les URI
09/08/2019	AIFE	V1.09/ Complément sur les informations nécessaire dans le swagger
25/09/2019	AIFE	V1.10/ Ajout du §3.7 Etat de santé de l'API
21/06/2022	AIFE	V2.00/ Ajout du §3.9 Open API Initiative Ajout du §3.6 Serveur API et URL de base

Objectif du document	Guide décrivant les recommandations et conseils à suivre par les fournisseurs d'API
Mots clefs	Design, REST,
Résumé	Résumé s'il y a lieu

Sommaire :

Tél : 00 00 00 00
Mél : prénom.nom@xxx.fr
Adresse, code postal, ville

1. Introduction

Ce document décrit les principes directeurs et les recommandations associés à la conception, au développement et à l'exposition d'API par un fournisseur de services souhaitant exposer ses API sur la plateforme API Mutualisée de l'AIFE.

10, Rue du Centre
93160 NOISY-LE-GRAND
Tél : 01 57 33 95 87
https://aife.economie.gouv.fr

Il s'adresse aux équipes en charge de la mise en œuvre d'API au sein de leur SI respectif et souhaitant exposer leurs API sur la plateforme mutualisée.

2. Conception d'API

Ce chapitre décrit les principes à adopter dans le cadre de la conception d'API. Il rassemble les recommandations les plus communément appliquées dans ce domaine et s'adresse aux développeurs ou équipes en charge du développement d'API collaborant avec les équipes métier.

2.1 L'approche service

L'API est l'industrialisation du processus de consommation des ressources de l'entreprise sur le Web. Elles doivent être conçues comme des services en pensant d'abord aux usages qui seront fait des ressources mises à disposition et aux actions qui pourront leur être appliquées, avant de penser aux données échangées. Les réponses aux questions suivantes pourront ainsi permettre de mieux cibler ces usages et actions :

- Qui va utiliser le service ?
- Que veut faire le client de mon service ?
- Que peut et doit faire le client des ressources mises à disposition ?

Les API sont avant tout destinées aux développeurs des applications ou sites web destinés à les consommer.
« Faire simple » doit donc rester le mot d'ordre de la conception d'API : il faut offrir aux consommateurs des services simples, clairs, intuitifs et compréhensibles, sans avoir systématiquement besoin de se reporter à la documentation.

L'exposition d'API cohérentes et complémentaires permet par ailleurs :

- Un couplage faible entre les API ;
- D'éviter la redondance au niveau des API disponibles ;
- De maîtriser les interactions et limiter le risque d'enchaînement en cascade d'appels d'API.

2.2 L'exposition des ressources et le choix des actions

2.2.1 Choix des ressources

Les services web conformes au style d'architecture REST (RESTful) se doivent de respecter un découpage en ressources. Il est recommandé de proposer des ressources dont la sémantique métier est de granularité moyenne : un découpage trop fin risquerait de perdre le développeur dans un catalogue trop large, tandis qu'un découpage trop large entrainerait des échanges de données disproportionnés pour l'usage final.
Il est ainsi conseillé :

- De ne grouper que les ressources qui seront accédées à la suite de manière quasi systématique ;
- De ne pas grouper sous une même ressource les collections pouvant assembler de nombreux composants. Par exemple, la liste des expériences professionnelles peut être très longue ;
- Que l'arborescence des ressources ne comporte pas d'imbrications de plus de deux niveaux de profondeur.

```
GET /users/007
{"id":"007",
  "first_name":"James",
  "name":"Bond",
  "address":{
                "street":"Horsen Ferry Road",
                "city":{"name":"London"}
                }
}
```


10, Rue du Centre
93160 NOISY-LE-GRAND
Tél : 01 57 33 95 87
https://aife.economie.gouv.fr

2.2.2 Nommage des ressources

Afin d'éviter toute confusion avec les actions, l'usage de verbes devra être interdit dans les noms de ressources : seuls les noms doivent être utilisés pour nommer les ressources.
L'utilisation de noms oriente vers un usage de type CRUD (Create/Read/Update/Delete, opérations de base pour la gestion de collections), tandis que l'utilisation de verbes oriente vers un usage de type RPC (Remote Procedure Call, protocole permettant de faire des appels de procédure sur un ordinateur distant).

Il est également recommandé d'utiliser systématiquement le pluriel pour gérer les deux types de ressources (instance et collection). L'utilisation du pluriel permet ainsi d'homogénéiser l'interface : l'URL n'est jamais modifiée que l'appel concerne une instance ou une collection.
L'utilisation du singulier doit être réservée à une ressource unique (singleton) au sein du système (la ressource recherche par exemple, voir §3.2.4 Recherche)

Il est recommandé d'utiliser systématiquement pour les noms de variables et de ressources métier le français, tandis que l'usage de l'anglais sera réservé aux ressources techniques (on pourra par exemple utiliser « search » pour la ressource technique de recherche, voir §3.2.4 Recherche).

Le choix de la casse revêt moins d'importance mais il est indispensable de s'y tenir et de ne plus en changer. La casse peut cependant être différente entre la description des ressources dans les URL et le corps des requêtes (http body). On pourra ainsi choisir parmi les casses suivantes : UpperCamelCase, lowerCamelCase, snake_case et spinal-case.
Pour les URL et URI, nous recommandons l'utilisation de spinal-case (mise en avant par la RFC3986), ou snake_case (fréquemment utilisée par les Géants du Web), qui permettent d'assurer la compatibilité avec les serveurs ne tenant pas compte de la casse.
Pour le corps de la requête, nous recommandons l'utilisation de lowerCamelCase ou snake_case, plus faciles à lire par un humain.

2.2.3 Les actions

Le respect des principes REST conduira à l'organisation de l'API en ressources logiques, identifiées par des URL. La manipulation de ces ressources se fait au travers de l'utilisation de verbes d'action du protocole http :

- **POST** pour la création, équivalent de l'action CRUD Create ;
- **GET** pour la consultation/lecture, équivalent de l'action CRUD Read ;
- **PUT** pour la mise à jour, équivalent de l'action CRUD Update ;
- **DELETE** pour la suppression, équivalent de l'action CRUD Delete.

L'utilisation du verbe PATCH, dont l'usage est ambigu, n'est pas recommandée.

On s'assurera que l'utilisation du verbe GET est sécurisé et qu'il n'entraine aucun changement d'état sur les ressources (réservé aux autres verbes POST, PUT et DELETE). De même, les méthodes PUT et DELETE[1] se doivent d'être idempotentes pour l'API : le résultat doit être identique quel que soit le nombre de fois où la méthode a été appliquée, le client peut donc renvoyer la requête sans risque pour le système.

Il convient également de rappeler ici que les appels REST doivent être stateless : aucun contexte ne doit être associé aux appels.

3. Développement d'API

3.1 Spécifier et Documenter les API

[1] La méthode de suppression DELETE laisse toujours le système dans le même état : après le premier appel, la ressource est supprimée. Lors des appels suivants, la ressource est déjà supprimée, le système est donc dans le même état qu'après la première requête. Cependant, la réponse du serveur sera différente : un code 200 ou 204 sera renvoyé sur la première requête, tandis qu'un code 404 avertira le client de l'absence de la ressource pour les suivantes (voir §3.3.3 Codes retour http).

10, Rue du Centre
93160 NOISY-LE-GRAND
Tél : 01 57 33 95 87
https://aife.economie.gouv.fr

Les spécifications doivent être suffisamment précises pour permettre de développer rapidement les API et faciliter leur maintenance évolutive.
Plusieurs outils permettent de réaliser ces spécifications, dans un format qui pourra être exploité par la plateforme API, afin de proposer aux développeurs une documentation précise et claire : les outils à utiliser sont Swagger 2.0 ou RAML.

Toute modification de l'API devra donc se faire dans une approche Spec driven development en passant par une mise à jour des spécifications, la réalisation des développements associés et la diffusion des nouvelles spécifications sur la plateforme API.

Nous recommandons également d'enrichir le champ "description" de l'API avec les informations suivantes :

- L'organisation qui expose l'API2;
- Description fonctionnelle de l'API ;
- Les éventuels clients ou profil de clients susceptibles d'utiliser l'API2;
- Version (voir §3.5 Version d'API) :

  Faire apparaitre les évolutions et corrections sur la dernière version ;
  Afficher les dates de fin de vie des versions ;
- Niveaux de services :
  - Disponibilité de l'API ;
  - Temps de réponse ;
  - Limites de consommation ;
- Sécurité mise en œuvre (autorisation, accès, authentification) ;
- Sources de données utilisées ;
- Documentation technique ou lien vers une documentation technique2;
- Des coordonnées de contact (mail, lien site Web, téléphone) en cas de question2.

Nous recommandons également d'enrichir le champ "description" de l'API avec les informations suivantes :

3.1.1 Fournir des exemples

Les documentations des API, et donc leurs spécifications, doivent contenir des exemples de représentation des ressources. De même, il faut y inclure des exemples d'utilisation pour permettre la mise en œuvre de tests par les développeurs, ainsi que par la plateforme API au travers du portail développeur (try-it) sous la forme de scripts cURL.

3.2 Les requêtes

3.2.1 Pagination

Afin d'anticiper l'évolution des quantités de données retournées par l'API, il est nécessaire de prévoir dès l'initialisation la pagination des ressources. Nous recommandons donc de paginer les ressources avec des valeurs par défaut lorsque celles-ci ne sont pas spécifiées par l'appelant, par exemple avec une plage de valeurs [0-25].
Il est recommandé d'utiliser le paramètre de requête ?range=0-25 et les Headers standards http pour la réponse (voir §3.3.2 Pagination) :

- Content-Range
- Accept-Range

D'un point de vue pratique, la pagination est souvent gérée dans l'url via la query-string. Nous recommandons d'accepter uniquement cette solution, et de ne pas tenir compte du header Range

[2] Ces informations sont nécessaires si cette API a pour vocation d'être référencée sur https://api.gouv.fr.

HTTP. La pagination est une information importante qui, par souci de lisibilité, doit être positionnée dans la requête.
Il est recommandé d'utiliser une plage de valeurs, via l'index des ressources de votre collection. Par exemple, les ressources de l'index 10 à l'index 25 inclus équivaut à? range=10-25.

3.2.2 Tri

Le tri du résultat d'un appel sur une collection de ressources passe par deux principaux paramètres :

- sort: contient les noms des attributs, séparés par une virgule, sur lesquels effectuer le tri.
- desc: par défaut le tri est ascendant (ou croissant). Afin de l'obtenir de façon descendant (ou décroissant), il suffit d'ajouter ce paramètre (sans valeur par défaut). On voudra dans certains cas spécifier quels attributs doivent être traités de façon ascendante ou descendante ; on mettra alors dans ce paramètre la liste des attributs descendants, les autres seront donc par défaut ascendants.

Exemple de tri avec modification de l'ordre de tri:

```
CURL –X GET \
-H "Accept: application/json" \
https://api.fakecompany.com/v1/restaurants?sort=rating,reviews,name&desc=rating,reviews
```

3.2.3 Filtrage

Il est recommandé d'utiliser le caractère '?' pour filtrer les ressources.
Un filtre consiste à limiter le nombre de ressources traitées par la requête, en spécifiant des attributs et leurs valeurs correspondantes attendues. Il est possible de filtrer une collection sur plusieurs attributs simultanément et de permettre plusieurs valeurs pour un même attribut filtré.
Pour cela, nous proposons d'utiliser directement le nom de l'attribut avec une égalité sur les valeurs attendues, chacune séparée par une virgule.

3.2.4 Recherche

Lorsque le filtrage ne suffit pas (pour faire du partiel ou de l'approchant par exemple), on passera alors par une recherche sur les ressources.
Une recherche est en elle-même une sous-ressource de votre collection, car les résultats qu'elle fournit auront un format différent des ressources recherchées et de la collection elle-même. Cela permet d'ajouter des suggestions, des corrections et des informations propres à la recherche.
Les paramètres sont fournis de la même manière que pour le filtre, via la query-string, mais ceux-ci ne seront pas nécessairement des valeurs exactes et pourront avoir une nomenclature permettant de faire de l'approchant.
La recherche étant une ressource à part entière, elle doit supporter la pagination de la même manière que les autres ressources de votre API.

Le modèle Google est recommandé pour la prise en compte des recherches sur plusieurs ressources :

GET /search?q=running+paid

3.3 Les réponses

3.3.1 Représentation des ressources

Les ressources peuvent être présentées sous différentes formes ou représentations et nous recommandons de gérer plusieurs formats de distribution du contenu de votre API. On utilisera pour

cela l'entête HTTP prévue à cet effet : « Accept », dans lequel l'ordre de préférence de formats demandés sera indiqué.
Il est conseillé de gérer au minimum deux formats : JSON et XML. La tendance actuelle privilégie le format JSON au format XML, pour plusieurs raisons :

- Le XML est difficile à lire pour un humain, alors qu'un document JSON est bien plus intuitif ;
- Le XML est verbeux ;
- Le XML est plus compliqué à parser. Le JSON, quant à lui, est lisible par du code Javascript, utilisé par les frameworks front-end, dès sa réception.

Dans les cas où il n'est pas possible de fournir le format demandé, une erreur http 406 est retournée (voir §3.3.3 Codes retour http).

Remarque: L'utilisation de valeurs dans les clés est à proscrire.
Un bon exemple :

{"id": "125", "name": "Environment"},{"id": "834", "name": "Water Quality"}. Un mauvais exemple : {"125" : "Environment"},{"834" : "Water Quality"}.

3.3.2 Pagination

Le code retour http correspondant au retour d'une requête paginée sera 206 (voir §3.3.3 Codes retour http), sauf si les valeurs demandées provoquent la remontée de l'ensemble des données de la collection, auquel cas le code retour sera 200 (voir §3.3.3 Codes retour http).

La réponse de votre API sur une collection devra obligatoirement fournir dans les en-têtes http :

- Content-Range offset – limit / count
  - **offset** : l'index du premier élément retourné par la requête.
  - **limit** : l'index du dernier élément retourné par la requête.
  - **count** : le nombre total d'élément que contient la collection.

- Accept-Range resource max
  - **resource : le type de la pagination, on parlera ici systématiquement de la ressource en cours d'utilisation, par exemple : client, order, restaurant, …**
  - **max : le nombre maximum pouvant être requêté en une seule fois.**

Dans le cas où la pagination demandée ne rentre pas dans les valeurs tolérées par l'API, la réponse http sera un code erreur 400 (voir §3.3.3 Codes retour http), avec une description explicite de l'erreur dans le body.

IL est fortement conseillé d'incorporer dans les entêtes http de vos réponses la balise Link. Celle-ci vous permet d'ajouter, entre autres, des liens de navigations tels que la plage suivante, la plage précédente, la première plage, la dernière plage, etc.
Pour retourner les liens vers les autres plages, nous préconisons la notation ci-dessous, utilisée par GitHub, compatible avec la RFC5988 (et qui permet de gérer les clients qui ne supportent pas plusieurs

```
CURL –X GET \
-H "Accept: application/json" \
https://api.fakecompany.com/v1/orders?range=48-55
< 206 Partial Content
< Content-Range: 48-55/971
< Accept-Range: order 10
< Link : <https://api.fakecompany.com/v1/orders?range=0-7>; rel="first",
<https://api.fakecompany.com/v1/orders?range=40-47>; rel="prev",
<https://api.fakecompany.com/v1/orders?range=56-64>; rel="next",
<https://api.fakecompany.com/v1/orders?range=968-975>; rel="last"
```

Header "Link") :
Une autre notation est fréquemment rencontrée, où la balise d'en-tête HTTP Link est constituée d'une URL suivie du type de liens associés. Cette balise est répétable autant de fois qu'il y a de liens associés à votre réponse :
Ou bien la notation suivante dans le payload, utilisée par Paypal :

```
[
 {"href":"https://api.fakecompany.com/v1/orders?range=0-7", "rel":"first", "method":"GET"},
 {"href":"https://api.fakecompany.com/v1/orders?range=40-47", "rel":"prev", "method":"GET"},
 {"href":"https://api.fakecompany.com/v1/orders?range=56-64", "rel":"next", "method":"GET"},
 {"href":"https://api.fakecompany.com/v1/orders?range=968-975", "rel":"last", "method":"GET"},
]
```

3.3.3 Codes retour http

L'utilisation des codes retours http de manière appropriée fait partie intégrante du modèle REST. Nous préconisons donc leur usage, en exploitant au minimum les codes suivants :

```
< Link: <https://api.fakecompany.com/v1/orders?range=0-7>; rel="first"
< Link: <https://api.fakecompany.com/v1/orders?range=40-47>; rel="prev"
< Link: <https://api.fakecompany.com/v1/orders?range=56-64>; rel="next"
< Link: <https://api.fakecompany.com/v1/orders?range=968-975>; rel="last"
```

Code	Message	Description
200	OK	Code retour par défaut en cas de succès
201	Created	Code retour en cas de succès du traitement et de la création d'une nouvelle ressource
202	Accepted	Indique que la requête a bien été prise en compte et sera traitée ultérieurement. Ce mode de fonctionnement est principalement utilisé dans le cas d'échanges asynchrones et nécessite la mise en place d'un mécanisme de Call Back côté client
204	No Content	Indique que la requête a bien été traitée mais qu'il n'y a pas de résultat à retourner. C'est par exemple le cas lors d'actions de suppression
400	Bad Request	Code erreur générique en cas d'informations non valides fournies au service
401	Unauthorized	Code utilisé par les services nécessitant une autorisation lorsque l'identification fournie n'est pas autorisée à utiliser le service
402	Unprocessable entity	Code retour générique lorsque la requête ne peut être traitée suite à des paramètres d'entrée non valides
404	Not Found	Code erreur en cas d'URI d'entrée inexistante
405	Method not Allowed	Code erreur en cas d'incompatibilité entre une URI et une méthode

406	Not Acceptable	Code retour lorsque les entêtes ne semblent pas compatibles avec le fonctionnement du service
429	Too Many Requests	Code retour lorsque le client émet trop de requêtes dans un délai donné
500	Server Error	Code erreur par défaut
503	Service Unavailable	Code retour lorsque le service n'est pas disponible

3.3.4 Gestion des erreurs

En cas d'erreur, il est nécessaire de fournir en plus du code http :

- un message à destination du développeur ;
- un message pour l'utilisateur final (si approprié) ;
- le code erreur interne (documenté par ailleurs) ;
- un lien vers la documentation.

L'utilisation de la structure JSON suivante est donc recommandée :

```
{
  "error": {
            "errorLabel": "description_courte",
            "errorDescription": "description longue, lisible par un humain ",
            "errorUri": "URI vers une description détaillée de l'erreur sur le portail développeur"}
}
```

L'attribut error n'est pas forcement redondant avec le code retour http : il est possible d'avoir deux statuts différents pour une même valeur sur la clé « error » et inversement.

- 400 & error=invalidUser
- 400 & error=invalidCart

Cette représentation est issue de la spécification OAuth2. Sa généralisation à l'API REST permettra au module client qui la consomme de ne pas avoir à gérer deux structures d'erreurs distinctes.

Note: il peut être pertinent de fournir dans certains cas une collection de cette structure, pour retourner plusieurs erreurs simultanées, ce qui peut être utile dans le cas d'une validation de formulaire côté serveur par exemple.

3.3.5 Gestion des traitements longs

Afin de ne pas dégrader les niveaux de services de l'API, le maintien des connexions doit être évité autant que possible dans le cadre de traitements de longue durée, supérieure à 10 secondes. Des mécanismes adaptés doivent être ainsi mis en œuvre :

- Callback : l'utilisateur indique à l'API la localisation du système par lequel le résultat doit lui être transmis. Il s'agit généralement d'une API dédiée à la réception des retours. Pour l'utilisation de ce mécanisme, le fournisseur doit s'assurer qu'une connexion directe avec le fournisseur est possible, tant du point vue accès que sécurité (la réponse émise dans le cadre du callback ne passe pas par la plateforme API).
- Corrélation : l'utilisateur et le fournisseur de l'API s'entendent sur un identifiant unique permettant de corréler le résultat transmis à un précédent appel.

3.3.6 Contrôles hypermedia

L'ajout de contrôles hypermedia ou approche HATEOAS (Hypermedia As The Engine of Application State) permet de proposer dans les réponses — à l'instar des liens présents dans les pages web d'un site internet - des liens hypertexte pour guider les développeurs et leur faire découvrir les ressources et actions possibles de l'API, contextualisées aux ressources retournées.
Cette approche permet de développer l'affordance de l'API (capacité d'un objet à suggérer l'utilisation qui peut en être faite).

L'implémentation n'est cependant pas standardisée, on pourra par exemple utiliser celle de PayPal :

GET /clients/007

```
< 200 Ok
< { "id":"007", "firstname":"James",...,
< "links": [
<   {"rel":"self","href":"https://api.domain.com/v1/clients/007",  "method":"GET"},
<   {"rel":"addresses","href":"https://api.domain.com/v1/addresses/42",  "method":"GET"},
<   {"rel":"orders", "href":"https://api.domain.com/v1/orders/1234",  "method":"GET"},
<   ...
<   ]
> }
```

Ou, pour être compatible avec la RFC5988, la notation proposée par GitHub :

GET /clients/007

```
< 200 Ok
< { "id":"007", "firstname":"James",...}
< Link : <https://api.fakecompany.com/v1/clients>;  rel="self"; method:"GET",
<  <https://api.fakecompany.com/v1/addresses/42>;  rel="addresses"; method:"GET",
<  <https://api.fakecompany.com/v1/orders/1234>;  rel="orders"; method:"GET"
```

3.4 Cross-domain

Depuis des années, les navigateurs restreignent l'accès à des ressources n'appartenant pas au domaine depuis lequel un document a été chargé (règle dite de Same-Origin Policy).
Le fait que la balise <script> ne soit pas concernée par cette restriction a ouvert la voie à la technique JSONP (JSON with padding), permettant d'émettre des requêtes AJAX cross-domain retournant des données JSON encapsulées dans une fonction de rappel (callback) en JavaScript. Cette technique, bien qu'encore très utilisée, relève surtout de l'astuce.

Nous recommandons donc l'utilisation de CORS : c'est un mécanisme robuste et normalisé par le W3C permettant à tout navigateur compatible[3] d'effectuer des requêtes HTTP cross-domain. Il a valeur de standard industriel et étend le champ des possibilités bien au-delà de ce qui était jusque-là permis par JSONP.
La mise en œuvre de CORS coté serveur consiste en général à ajouter quelques directives sur les serveurs HTTP (Nginx/Apache/Nodejs...).
Coté client, la mise en œuvre est transparente : le navigateur effectuera avant chaque requête GET/POST/PUT/DELETE une requête HTTP avec le verbe OPTIONS.

[3] Navigateurs compatibles CORS : Internet Explorer 10+, Firefox 3.5+, Chrome 3+, Safari 4+

10, Rue du Centre
93160 NOISY-LE-GRAND
Tél : 01 57 33 95 87
https://aife.economie.gouv.fr

3.5 Version d'API et URI

Les API doivent naturellement être versionnées et il est déconseillé d'exposer une API sans version : nous recommandons de faire apparaitre la version de l'API dans l'URL, et de n'indiquer que la version majeure sous la forme vN (où N est le numéro de version).
Une API doit pouvoir supporter deux versions en même temps : maintenir plus de versions freine l'adoption des évolutions et apporte de la confusion aux développeurs. Dès la mise en œuvre de la nouvelle version « vN+1 », la version « vN » ne devrait pas être disponible au-delà de 12 mois. Ce délai doit permettre une migration de la version « vN » vers la version « vN+1 » en toute quiétude. La compatibilité ascendante doit être assurée dans la mesure du possible.

Les consommateurs doivent être avertis de la publication d'une nouvelle version, avec une communication sur les évolutions et les dates de fin de support des versions précédentes (voir §3.1 Spécifier et Documenter les API).

Les méthodes de l'API ainsi versionnée seront accessibles comme sous-ressources de la version. Les URI de consommation seront alors définies sur PISTE sous la forme suivante :

https://api.aife.economie.gouv.fr/<*nom_fournisseur*>/<*nom_api*>/v<*num_version*>/<*method_path*>

3.6 Serveur API et URL de base:

Si vous utilisez OpenAPI 3.0, vous devez spécifier une ou plusieurs URL pour votre API dans la section servers de votre swagger. servers remplace host, basePath et schemes utilisés en OpenAPI 2.0. Nous recommandons de mettre votre URL de base (basePath) dans Url de votre serveur API:

```
{
  "servers": [
    {
      "url": "https://{username}.gigantic-server.com:{port}/{basePath}",
      "description": "The production API server",
      "variables": {
        "username": {
          "default": "demo",
          "description": "this value is assigned by the service provider, in this example `gigantic-
        },
        "port": {
          "enum": [
            "8443",
            "443"
          ],
          "default": "8443"
        },
        "basePath": {
          "default": "v2"
        }
      }
    }
  ]
}
```

3.7 Internationalisation

Dans le cas du support de différents langages, il est préférable d'utiliser le header Accept-Language au

lieu d'un paramètre d'URL comme ?language=fr.
L'utilisation du standard ISO 8601 pour les dates, heures et timestamps est obligatoire (exemples : 1978-05-10T06:06:06+00:00 ou 1978-05-10).

3.8 Etat de santé de l'API

Afin de faciliter l'exploitation de l'API et permettre une surveillance de son état de la part de la plateforme PISTE qui sera chargée de l'exposer, il est fortement conseillé de prévoir une méthode « Healthcheck » ajoutée à l'API exposée. Cette méthode retournera le code http 200 (OK) si tout va bien, ou le code adapté (voir §3.3.3 Codes retour http ci-dessus) dans le cas contraire.
Si plusieurs API sont exposées, il est conseillé de mettre en œuvre une API dédiée à l'obtention de l'état de santé de chacune des API exposées, ainsi que des services dont elles dépendent. Cette API pourra éventuellement permettre l'obtention de données de performance (temps de réponse, nombre de connexions maximum, ...) pour augmenter la proactivité de la plateforme PISTE.

3.9 Open API Initiative

Le programme Open API Initiative a été créé par un consortium industriel afin de normaliser les descriptions d'API REST entre les différents fournisseurs. Dans le cadre de ce programme, la spécification Swagger 2.0 a été renommée en spécification OpenAPI (OAS) et placée sous le programme Open API Initiative.
OpenAPI promeut une approche qui commence par le contrat plutôt que par l'implémentation.
Cela signifie que vous commencez par concevoir le contrat d'API (l'interface), puis que vous écrivez le code qui implémente ce contrat.

Références

Designer une API REST, OCTO
Transformez votre API Web en une API Hypermedia, OCTO
Concevoir des API REST élégantes, ekino
De la conception d'une API REST, SUPINFO

Open API Initiative Microsoft

10, Rue du Centre
93160 NOISY-LE-GRAND
Tél : 01 57 33 95 87
https://aife.economie.gouv.fr