Documenter une API REST avec OpenAPI

→ Qu'est ce que OpenAPI ?
OpenAPI est une specification permettant de décrire une API REST, cette description permet de générer une documentation et de tester son API avec les outils fournis par Swagger.

→ Quel intérêt ?
La spécification OpenAPI fonctionne notamment avec tout l'écosystème de Swagger, qui permet de génerer de la documentation, des tests, et du code côté client ou serveur dans de nombreux langages.

→ Quel notation utiliser ?
Les définitions OpenAPI peuvent être réalisées en YAML ou en JSON. Les exemples fournis ici sont en YAML.

Structure de base
Une définition OpenAPI contient les informations suivantes :

- ***Métadonnées*** : les informations au sujet de la description elle même et non de l'API
- ***Serveurs*** : L'URL du ou des serveurs fournissant l'API
- ***Chemins*** : Les différents points d'entrée (endpoints) de l'API et les opérations que l'on peut y effectuer (représentés par un verbe HTTP)
- ***Paramètres*** : Les opérations peuvent prendre des paramètres qui modifient leur comportement
- ***Requêtes*** : description du contenu des requêtes
- ***Réponses*** : Les différentes réponses possibles à une requête
- ***Structure des données*** : Permet de définir la structure des informations qui entrent et sortent de l'API sous forme de schemas réutilisables.
- ***Authentification*** : Les méthodes d'authentification utilisées par l'API

Metadonnées
Il faut tout d'abord préciser la version de la spécification OpenAPI sur laquelle se base votre définition.

```
openapi : 3.0.0
```

La section « info » contient le reste des métadonnées.

```
info:
  title: Sample API
  description: Optional multiline or single-line description in
[CommonMark](http://commonmark.org/help/) or HTML.
  version: 0.1.9
```

- *title*: le nom de l'API
- *description*: informations au sujet de l'API, possibilité d'utiliser du markdown
- *version*: la version de votre API

Serveurs

La section « *server* » indique le ou les serveurs utilisés par l'API et qui sera utilisée comme l'URL de base des requête.

```
servers:
  - url: http://api.example.com/v1
    description: Optional server description, e.g. Main (production)
server
  - url: http://staging-api.example.com
    description: Optional server description, e.g. Internal staging
server for testing
```

Tout les chemins définis ensuite seront relatifs à ces URL.

Chemins

La section « *path* » définit les différents chemins proposés par l'API et les requêtes et réponses associées a chacun d'entre eux.

```
paths:
  /users:
    get:
      summary: Returns a list of users.
      description: Optional extended description in CommonMark or HTML.
      responses:
        '200':    # status code
          description: A JSON array of user names
          content:
            application/json:
              schema:
                type: array
                items:
                  type: string
```

- */users*: le chemin associé à l'URL, la bonne pratique REST veut qu'il soit au pluriel

- *get*: le verbe HTTP de la requête, OpenAPI 3.0 accepte get, post, put, patch, delete, head, options, et trace
- *summary*: Résumé de ce que fait la requête
- *description*: peut être multiligne et permet l'usage du format markdown
- *Responses*: les différentes réponses que le serveur peut renvoyer, il faut définir au minimum une réponse par requête.

Remplacer le serveur principal d'un chemin: Il est possible de forcer un serveur en particulier sur un chemin avec un champ server

Exemple:

```
paths:
  /files:
    description: File upload and download operations
    servers:
      - url: https://files.example.com
        description: Override base path for all operations with the
/files path
```

Paramètres

les paramètres sont définis dans la section *parameters* d'une requête.
Les paramètres se distinguent par leur emplacement, qui est défini avec le champ « *in* »

- paramètres de chemin */users/{id}*
- paramètres de requête */users?role=admin*
- paramètres d'en-tête *X-myHeader : Value*
- cookies, contenus dans l'en-tête *Cookie: debug=0 ;*

Paramètres de chemin :

```
paths:
  /users/{id}:
    get:
      parameters:
        - in: path
          name: id   # Notez que le nom est le meme dans le chemin
          required: true
          schema:
            type: integer
            minimum: 1
          description: The user ID
```

- Le champ « in » indique bien qu'il s'agit d'une paramètre de chemin.
- Pensez a ajouter « required : true » car les paramètres de chemins sont toujours requis.
- Le paramétre est noté entre crochets dans l'URL du point d'entrée

Paramètres de requête :
Le type le plus fréquent de paramètre, il apparaît a la fin de l'URL après un «? » avec différente paires « clé=valeur » séparées par des « & ». Ils peuvent étre optionnels.

```
  - in: query
          name: limit
          schema:
            type: integer
          description: The numbers of items to return
```

→ Les clés d'API passées sous la forme de paramètres de requête sont définies dans la section « security » à l'aide de schemas « securitySchemes »

Paramètres d'en-tête :
Il est possible de définir des en-têtes personnalisés

```
paths:
  /ping:
    get:
      summary: Checks if the server is alive
      parameters:
        - in: header
          name: X-Request-ID
          schema:
```

→ Les en-têtes ne peuvent pas s'appeler « Accept, Content-Type et Authorization » car d'autre champs sont prévus pour les décrire.

Cookies :

```
  - in: cookie
          name: csrftoken
          schema:
            type: string
```

Note : Certains paramètres peuvent recevoir des chaines ou des tableaux voir objets qu'il est possible de sérialiser (voir ici)

Décrire le contenu d'une requête

Le contenu d'une requête est défini dans le champ « requestBody » optionnel par défaut il peut être rendu obligatoire avec « required : true »

```yaml
paths:
  /pets:
    post:
      summary: Add a new pet
      requestBody:
        description: Optional description in *Markdown*
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Pet'
          application/xml:
            schema:
              $ref: '#/components/schemas/Pet'
          application/x-www-form-urlencoded:
            schema:
              $ref: '#/components/schemas/PetForm'
          text/plain:
            schema:
              type: string
```

requestBody peut donc contenir 3 champs :

- *description* : accepte le markdown
- *required* : false par défaut si omis
- *content* : l'objet contenu qui indique les différents média type et le schema de leurs données, content autorise les jokers pour les média types à l'aide de « * »

  *Exemple : « */* » ou « image/* »*

Définir une réponse

Votre spécification doit définir des réponses de la part du serveur pour les différentes requêtes, il en faut au moins une pour chacune d'entre elles, les réponses peuvent être définies sous formes de schemas pour être réutilisables.

- *'200'*: Le code HTTP de la réponse, (1XX, 2XX, 3XX, 4XX, 5XX) il n'est pas nécessaire de couvrir toutes les réponses possibles mais les réponses de succès et d'erreur connues sont attendues
- *content:* contenu de la réponse, le mot clé *schema* est utilisé pour définir le corps de la réponse, il est possible d'omettre ce champ si la réponse n'a pas de contenu

- *application/json:* le media type indique le format des données contenu dans le corps d'un . Celui ci doit correspondre au RFC 6838 sur les media types. Il est possible d'indiquer plusieurs media-types

- schema: décrit la structure des données, permet de définir des objets ou array (API json et xml) des types primitifs (nombres, chaines des caractères) pour les réponses en texte brut, et également des fichiers. Le schema peut être défini dans la section *components* afin d'être réutilisé.

```
responses:
        '200':
          description: A User object
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/User'
```

Exemple d'un schema défini dans les composants et appelé avec $ref

- *headers : Il est possible de définir des données d'en-tête personnalisées pour la réponse, ils sont également définis par un schema.*

```
    '200':
        description: OK
        headers:
          X-RateLimit-Limit:
            schema:
              type: integer
            description: Request limit per hour.
```

Modèles de données

Ce sont les fameux schemas qui permettent de définir la structure des données qui entrent et sortent de l'API, ces schemas peuvent être stockés dans la section components afin d'être réutilisés plusieurs fois.

Le type des données est indiqué dans le champ « type », les types suivants sont possibles :

- string (dates et fichiers compris)
- number
- integer
- boolean
- array
- object

Il n'y a pas de type null mais un champ booléen « nullable » indique que la valeur peut être nulle

```yaml
# Correct
type: integer
nullable: true
```

Si les valeurs reçues sont mixtes ils est possible d'indiquer différents types avec les champs « oneOf » et « anyOf » :

```yaml
# Correct
oneOf:
  - type: string
  - type: integer
```

Les données les plus fréquentes seront sous forme d'array ou d'object car les API REST utilisent majoritairement le format JSON nous allons nous pencher dessus :

```yaml
type: array
items:
  type: string
```

Le type array nécessite un champ items qui indique le type des données stockées à l'intérieur, ce type peut lui même être un autre array ou objet :

```yaml
# [ {"id": 5}, {"id": 8} ]
type: array
items:
  type: object
  properties:
    id:
      type: integer
```

Voici donc un tableau d'objets plus complexe.

Les objets quand a eux nécessitent un champ properties lisant leurs propriétés :

```yaml
type: object
properties:
  id:
    type: integer
  username:
    type: string
  name:
    type: string
required:
  - id
  - username
```

On remarque que le champ required est un attribut an niveau de l'objet et non de la propriété.

Il est possible d'ajouter des example de valeurs aux paramètres, proriétés et objets de l'API afin de clarifier son fonctionnement, les champs *example* et *examples* sont utilisés

Un example de propriété :

```yaml
components:
 schemas:
  User:    # Schema name
   type: object
   properties:
    id:
     type: integer
     format: int64
     example: 1         # Property example
    name:
     type: string
     example: New order  # Property example
```

Un example d'objet :

```yaml
components:
  schemas:
    User:         # Schema name
      type: object
      properties:
        id:
          type: integer
        name:
          type: string
      example:    # Object-level example
        id: 1
        name: Jessica Smith
```

Authentification et autorisation

OpenAPI utilise des schemas de sécurité pour définir ces mécanismes, et prend en charges ces différents systèmes :

- Authentification HTTP
- Clé d'API (en-tête, paramètres ou cookies)
- Oauth 2
- OpenID Connect Discovery

Le champ **securitySchemes** permet de définir les schemas de sécurité utilisés dans l'API
Le champ **security** permet d'appliquer des schemas de sécurité à toute l'api ou à un requête en particulier.

Définir un schema de securité : tout les schemas sont définis dans la section « components/securityScheme », on choisit un nom pour chaque schema et on précise le type de sécurité utilisé

```yaml
components:
  securitySchemes:
    BasicAuth:
      type: http
      scheme: basic
    BearerAuth:
      type: http
      scheme: bearer
    ApiKeyAuth:
      type: apiKey
      in: header
      name: X-API-Key
```

Les 4 types disponibles :

- http
- apiKey
- oauth2
- openIdConnect

Appliquer un schema :
L'ajout d'un champ « security » au niveau racine appliquera un schema a toute l'API, l'ajout au niveau d'une requête limitera son effet a celle ci.

```yaml
paths:
  /billing_info:
    get:
      summary: Gets the account billing info
      security:
        - OAuth2: [admin]   # Use OAuth with a different scope
      responses:
        '200':
          description: OK
        '401':
          description: Not authenticated
        '403':
          description: Access token does not have the required scope
```

La section composants

Afin d'éviter la duplication de code les contenus de l'API peuvent être définis dans la section components et réutilisés ailleurs avec $ref

Les composants possibles :

- schemas
- parameters

- securitySchemes
- requestBodies
- responses
- headers
- examples

Exemple :

```yaml
components:
  #-----------------------------------
  # Reusable schemas (data models)
  #-----------------------------------
  schemas:
    User:             # $ref: '#/components/schemas/User'
      type: object
      properties:
        id:
          type: integer
          format: int64
        name:
          type: string
    Error:            # $ref :'#/components/schemas/Error'
      type: object
      properties:
        code:
          type: integer
        message:
          type: string
```

Composants externes :

```yaml
components:
  schemas:
    Pet:
      $ref: '../models/pet.yaml'
      # Can now use use '#/components/schemas/Pet' instead
    User:
      $ref:
'https://api.example.com/v2/openapi.yaml#/components/schemas/User'
      # Can now use '#/components/schemas/User' instead
```

Utiliser des références :
Les références permettent d'accéder à des ressources situées à d'autres emplacements

Les différents types de références :

- Locale ($ref : '#/definition/monElement', # représente la racine du document)

- Externe ($ref : '../document.json#/myElement') recherche la ref parmis les documents en local
- URL ($ref : 'http://chemin/vers/la/ressource') permet d'accéder au documents d'autres serveurs

Grouper les opérations avec des tags

Interprétés de maniére différente suivant les outils, les tags permettent de regrouper les opérations par catégories, on utilise la champ « tags » pour cela

```yaml
paths:
  /pet/findByStatus:
    get:
      summary: Finds pets by Status
      tags:
        - pets
      ...
  /pet:
    post:
      summary: Adds a new pet to the store
      tags:
        - pets
      ...
  /store/inventory:
    get:
      summary: Returns pet inventories
      tags:
        - store
      ...
```

Références

- Le guide openAPI de swagger
- La spécification OpenAPI