# **Chapitre 1 : la Spécification *OpenAPI***

1. **Introduction :**

Actuellement, le développement d’applications est facilité par l’implémentation de différents services de différents fournisseurs. La plupart de ces services utilisent le Web pour leurs interactions via un réseau de communication. Ces différentes interactions sont assurées par différentes APIs des services correspondants.

Une API (*Application Programming Interface*) ou interface de programmation d'applications en français, est un ensemble de protocoles et d'outils utilisés pour créer des applications logicielles [3]. Elle permet de connecter les applications de manière transparente. En effet, son intégration passe souvent inaperçue car c’est la manière dont deux applications ou plus sont liées les unes aux autres via leurs API respectives et est l’un des éléments les plus essentiels dans un logiciel de sécurité réussi [3]. Sans cette intégration les applications ne seront plus en mesure de partager des données entre elles. Les APIs sont regroupés en deux grands types : le type REST (*Representational State Transfer*) et SOAP (Simple Object Access Protocol). Le type REST s’est largement imposé face à SOAP par sa flexibilité.

Ces dernières années, le développement des APIs REST ne fait qu’augmenter. En moyenne, plus de 2000 nouvelles APIs publiques ont été publiées chaque année depuis 2005 [4]. Une croissance similaire se retrouve au sein des entreprises, car les APIs internes permettent un développement plus efficace et plus rapide. Par conséquent, la définition des APIs Web a été uniformisée dans une spécification afin de faciliter leur gestion et leur utilisation. La spécification OpenAPI définit une interface standard permettant de décrire les API Web client-serveur indépendamment du langage de programmation. Cette spécification permet aux humains et aux ordinateurs de consulter et de comprendre les capacités du service sans accès au code source ou à la documentation. Lorsqu'ils sont correctement définis, les clients et les serveurs peuvent se comprendre et interagir les uns avec les autres avec un minimum d’intervention du programmeur.

Dans ce chapitre nous allons d’écrire la spécification OpenAPI, sa syntaxe ainsi que l’outils utilisé pour la description de la spécification.

Tous les exemples illustratifs dans ce document sont en YAML et la version de spécification utilisée est 3.0.0.

1. **Définition et Description :**

Une API est une interface de programmation, c'est-à-dire un ensemble d’opération que les développeurs peuvent utiliser pour communiquer avec d'autres logiciels sans savoir nécessairement comment [5]. Une spécification est une description des services et fonctionnalités fournis par un logiciel. Ainsi la spécification OpenAPI (OAS *The OpenAPI Specification*) anciennement connue sous le nom de spécification *Swagger*, est une spécification qui décrit une d'interface standard pour les APIs web qui, permet aux humains et aux ordinateurs de découvrir, de consulter et de comprendre les capacités du service web sans connaissance de l'implémentation du serveur ou sans avoir accès au code source [6].

OpenAPI offre huit objets parents qui ne sont pas tous obligatoires pour bien décrire une API et peuvent contenir plusieurs autres objets. Ces huit objets sont indiqués dans la figure suivants :

paths

object

servers

object

info

object

openapi

string

externalsDocs

object

tags

object

security

object

components

object

**Figure 1.1** : objets parents de OpenAPI

1. ***openapi***:

C’est une chaine correspondant à la version de la spécification de OpenAPI qui est remplacée par « *Swagger* » pour les versions de spécification inférieure à 3.

1. ***info***:

Il contient les informations de base à propos de l’API parmi lesquelles 2 sont obligatoires :

* **title**: il s’agit du titre de l’API du type chaine de caractère
* **version**: ce champ indique la version de l’API
* **Autres** : les facultatifs comme : description, contact et licence etc.

1. ***servers***:

Il spécifie le lien et la description du serveur utilisé. On a la possibilité de définir plusieurs serveurs [6].

1. ***paths***:

C’est un objet JSON dans lequel les opérations telles que GET, PUT, POST et DELETE seront définis. C’est une des parties dans laquelle nous pouvons ajouter des extensions c’est-à-dire créer notre propre operateur.

**Exemple** :

paths:

'/chemin:

x-monMethode:

description: ma propre methode

**Figure** 1.2 : objet *paths* avec une extension

1. ***components***:

C’est cette partie qui définit les objets réutilisables. Parfois, plusieurs opérations d’une API peuvent avoir les mêmes paramètres communs ou retourner les mêmes structures de réponse. Pour éviter la duplication du code on peut le définir dans une place commune (components) et le référencé en utilisant la propriété **$*ref*** (mais « *securitySchemes* » est une exception car il est référencé par son nom) qui indique le chemin où le code est défini. Il serve de conteneur et ces composantes n’ont pas d’effet direct sur l’API a moins qu’ils soient référencés quelques parts à l’extérieur de l’objet « *components* ». Il est considéré comme une enveloppe dans laquelle plusieurs composantes sont définies, parmi ces composantes les plus importantes sont :

* ***schemas***: il décrit la structure d’une requête, une réponse, un paramètre ou un type. Ce type peut être composé ou primitif, un type d’entrée ou fourni comme sortie. Il précise aussi les contraintes d’un type, son modèle, son format, énumérer les valeurs prise en compte par ce type, définir une valeur par défaut ou autres seulement en définissant les propriétés correspondantes.

**Exemple**:

components:

schemas:

personne:

type: object

properties:

nom:

type: string

prenom:

type: string

adresse:

type: object

properties:

rue:

type: integer

codePostal:

type: integer

format: int64

default: 1500

ville:

type: string

example: Tizi-Ouzou

pays:

type: string

example: Algérie

status:

type: string

enum:

- Celibateur

- Mariée

**Figure 1.3** : utilisation de *schemas* dans *components*

* ***parameters***: il décrit les paramètres d’une opération comme ceux nécessité dans une requête et peut contenir un ***schema*** (= *schemas*).

**Exemple :**

components:

parameters:

parametre:

name: ID

in: query

description: "identifiant"

required: true

schema:

type: "integer"

format: "int32"

default: 1

**Figure 1.4** : utilisations de *parameters* dans *components*

* ***securitySchemes***: il décrit les différents modes de sécurité applicables aux différentes parties fonctionnelles d’une API (voir figure 1.15).
* ***requestBody***: il décrit les manières de représentation d’une réponse tel que en JSON, XML, TEXT/PLAIN.

**Exemple :**

requestBody:

content:

application/json:

schema:

$ref: "#/components/schemas/etudiant"

application/xml:

schema:

$ref: "#/components/schemas/etudiant"

**Figure 1.5** : la *requestBody* d’une opération

1. ***security***:

L’objet *security* de la spécification de OpenAPI version 3 décrit les manières dont une partie d’une API ou une API peut être sécurisée. Il spécifie le protocole de sécurité ou d'autorisation utilisé lors de la soumission des demandes. Cette spécification supporte quatre approches de définition de sécurité qui sont : "apiKey", "http", "oauth2 (le plus utilisé)"et, "openIdConnect".

1. apiKey: cette technique utilise une clé secrète pour l’authentification. Elle doit être fournie lors de l’appelle de l’API avec la clé dans une requête par exemple :

GET /something?api\_key=abcdef12345.

securitySchemes:

api\_key: #nom de la securité

name: apiKey1

type: apiKey

in: query

security:

- api\_key: []

**Figure 1.6 :** type de sécurité « *apikey »*

1. http : il existe plusieurs catégorie dans ce type de sécurité lié au protocole http dont l’un est un simple envoie de requête http avec le mot *basic* suivit de username : password par exemple « Authorization: Basic ZGVtbzpwQDU1dzByZA »

components:

securitySchemes:

basicAuth:

type: http

scheme: basic

security:

- basicAuth: []

**Figure 1.7 :** type de sécurité « http »

1. oauth2 : est un protocole libre ou une délégation d’autorisation permettant de limiter l’accès à une API sécurisée tels que : APIs GitHub, Google, Facebook, etc. Cette délégation d’autorisation permet à une application tierce d’avoir un accès limité sur une ressource (ressource server), avec le consentement du propriétaire. Il utilise plusieurs scénarios appelés *flows* qui contrôlent aussi la lecture et l’écriture.

components:

securitySchemes: #description de la sécurité

oAuthSample: #nom de sécurité aléatoire

type: oauth2

description: This API uses OAuth 2

flows:

implicit:

authorizationUrl:

scopes:

read\_donnée: lecture de données

write\_donnée: modifier mes données

security: #on l’applique à l’API

- oAuthSample:

- read\_donnée: lecture de données

- write\_donnée

**Figure 1.8 :** type de sécurité « *oauth2* »

1. openIdConnect: ce type de sécurité réclame des informations sur l’utilisateur par exemple en l’obligeant à s’inscrire et ses informations sont mis dans un fichier JSON appelé ID.

components:

securitySchemes:

openId: #nom de sécurité aléatoire

type: openIdConnect

openIdConnectUrl:

security:

- openId: securitySchemes

- pets\_read

- pets\_write

- admin

**Figure 1.9 :** type de sécurité « *openIdConnect* »

Plusieurs types de sécurité peuvent être définit et réutilisé dans la même et différentes parties de la spécification avec l’objet *security*.

Après avoir défini les schémas de sécurité dans la section *securitySchemes* (dans components), la sécurité peut être appliquée à l'ensemble de l'API ou à des opérations individuelles en ajoutant respectivement la section de sécurité(*security*) au niveau racine ou au niveau opération. Dans le premier cas, les schémas de sécurité spécifiés sont appliqués globalement à toutes les opérations de l'API et dans l’autre ils ne concernent que l’opération en question (voir figure 1.15).

1. ***tags***:

C’est un tableau dont les entrés doivent être les mêmes que les valeurs de la propriété tags de l’objet d’un opérateur. Il donne un résumé des différentes parties de l’objet ***paths***. Il peut être définis soit comme étant un objet parent ou dans une opération (voir figure 1.10)

1. ***externalsDocs***:

Il permet d’inclure des liens vers un document externe. Cet objet peut être aussi définit dans les opérateurs dans l’objet ***paths***.

Exemple:

paths:

/etudiants:

get:

tags:

- **Récupération**

responses:

200:

description: une liste d’étudiant

tags:

- name: **Récupération**

description**: “”**

- name: Modification

description: **“”**

externalsDocs: https://swagger.io/specification/

**Figure 1.10 :** utilisation de *tags* et *externalsDocs* dans l’objet *paths*

Une API est décrite grâce à la définition des propriétés de ces différents objets. L’inclusion de ces objets en suivant la syntaxe (en JSON ou en YAML) permet une description détaillée ce qui renseigne l’utilisateur : de la version, du format, des types de donnée prise en charge par l’API, le type de sécurité utilisé, etc. C’est un formalisme de description : de structure des documents APIs, des manières pour adresser des requêtes correctes aux serveurs http, de quelle forme est la réponse et sur quelle contrainte les services sont fournis.

1. **Les langages et outils utilisés** :
2. **Les langages utilisés** :
3. **JSON (JavaScript Objet Notation)** :

*JavaScript Objet Notation* est un format de données basé sur les types de données de la programmation en langage JavaScript [7]. C’est un langage léger et est devenu le format principal d’échange d’information. Il permet de représenter les données avec une structure arborescence dont l’affichage respect la règle clé-valeur. C'est devenu le format le plus populaire pour envoyer des requêtes API et des réponses via le protocole HTTP car il est facile à comprendre pour développeurs et les machines. C’est une représentation d’objet sachant que ces objets peuvent contenir d’autres objets. JSON prend en charge six (6) type de données qui sont : objets (*objects*), tableaux (*arrays*), chaînes (*strings*), nombres (*numbers*), valeurs booléennes (*boolean values*) et null.

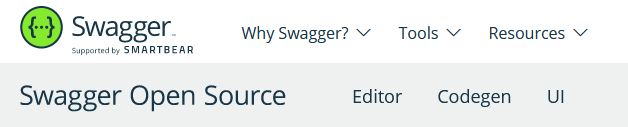
1. **YAML (Yet Another Markup Language)**:

YAML est un langage de sérialisation de données conçu pour être fonctionnel et lisible pour les humains [8]. Ce n’est pas un langage de balisage, il est utilisé dans les applications où les données sont stockées ou transmises. Il est compréhensible par les programmeurs, respect le principe d’indentation et utilise les tirets comme une nouvelle entrée dans un tableau. Ce langage spécifie les mêmes types d’informations que JSON. YAML a été spécialement créé pour bien fonctionner dans les cas d'utilisation courants tels que : fichiers de configuration, fichiers journaux, messagerie interprocessus, partage de données en plusieurs langues afin de faciliter la compréhension et la visualisation pour simplifier la programmation.

1. **Les outils utilisés** : le *Swagger*

*Swagger* actuelle est un ensemble d’outils pour aider les développeurs dans la conception, la construction, la documentation et la consommation d’API. C’est outil qui a subi des greffes d’autres outils tels que Swagger UI, Swagger Editor et Swagger Codegen [9].

L’environnement Swagger est un environnement composé de différente partie :



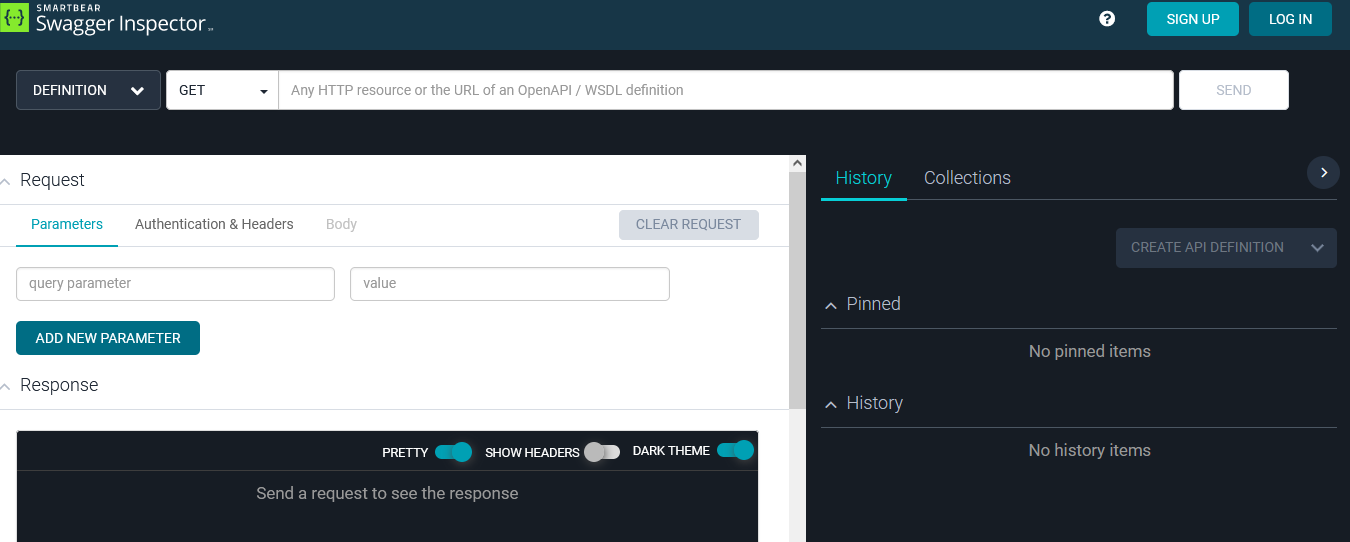
**Figure 1.11 :** différents outils de *Swagger*

* **Swagger editor :** permet de décrire des documents APIs web, de modifier des fichiers décrivant des APIs web et de concevoir de nouvelle API web. C’est un éditeur open source en ligne et c’est le moyen le plus facile pour commencer avec la spécification OpenAPI. Il permet la conversion de YAML en JSON et inversement car les deux langages sont supportés et composé d’une partie ou écrire le code et une autre partie permettant de visualiser sous formes d’interface le résultat du code sous forme d’API.

****

**Figure 1.12 :** interface de Swagger Editor

* **Swagger codegen** : ou le générateur de code, c’est un outil permettant de générer une librairie d’API en fournissant une spécification OpenAPI c’est-à-dire ensemble de fichier java permettant de générer le code client/serveur de différente plateforme de développement. Il génère le code pour la construction plus rapide de l’API définie avec la spécification OpenAPI.
* **Swagger User Interface** : L'interface utilisateur Swagger permet à n'importe qui, que ce soit une équipe de développement ou les consommateurs finaux, de visualiser et d'interagir avec les ressources de l'API sans avoir mis en place la logique de mise en œuvre [10]. Cette interface est générée automatiquement à partir de la spécification. Elle permet la documentation visuelle facilitant la mise en œuvre du back-end et la consommation côté client. C’est la partie visualisation de *Swagger* editor.
* **Swagger Inspector** : c’est un outil qui permet de tester facilement les API sur les Cloud. Il facilite la création de la documentation des APIs.

****

**Figure 1.13 :** interface de Swagger Inspector

1. **La syntaxe de cette spécification** :

La spécification OpenAPI est en générale versionnée de manière sémantique impliquant les fonctionnalités incompatibles apportée par rapport aux versions précédentes et l’ajout des nouvelles extensions. C’est une spécification qui commence de la version 3.0.0. Cette spécification doit suivre une syntaxe bien précise comme suit :

openapi: 3.0.0

info:

title: My API Specification

version: 1.0.0

servers:

- url: http

- url: https

paths:

/etudiants:

get:

tags:

- Récupération

responses:

200:

description: une liste d’étudiant

components:

securitySchemes:

api\_key:

name: apiKey1

type: apiKey

in: query

security:

- api\_key: []

tags:

- name: Récupération

- name: Modification

externalDocs:

url: https://swagger.io/specification/

description: Apprendre la spécification

**Figure 1.14** : Exemple d’une description d’une API

Cet exemple définit le code minimum pour la description d’une API en précisant la version de la spécification et de l’API, le nom de la spécification et l’opérateur GET.

1. **Ses Caractéristiques :**

Une spécification de OpenAPI peut être reconnue par :

* Son utilisation d’un ensemble d’objet JSON (peut être traduit en YAML) avec leur schéma spécifique pour définir la dénomination, l’ordre, et le contenu de chaque part d’une API.
* Sa description des APIs (REST API) dans le format standard (open source) et l’utilisation du modèle Client/serveur.
* La validité de sa description quel qu’en soit le langage de programmation utilisé par les clients.
* Elle supporte l’ajout des extensions.

1. **Son principe de fonctionnement :**

La spécification de OpenAPI est conçue sur la base d’apporter de l’aide aux utilisateurs des services APIs pendant les développements logiciels. Elle met le comportement d’une API en évidence vis-à-vis des utilisateurs avec usage des objets, la mise en place d’une opération, le type de serveurs (http, https) utilisés.

Elle propose un bref résumé sur la structure des différentes opérations en précisant les paramètres (id) dans l’objet « *parameter* » d’un operateur et si nécessaire définir une sécurité d’authentification avant leur utilisation.

La communication est basée uniquement sur le protocole http qui constitue l’envoie de requêtes et récupération de réponses. C’est une spécification basée sur le modèle Client/serveur. Elle indique des informations sur les codes d’état http pour préciser le succès (200), la redirection (301,302), accès refusés (403), l’erreur client et erreur serveur (500,503). Si ces codes sont définis, l’état de la requête sera interprété par la description du code correspondant.

**Exemple d’utilisation** :

openapi: 3.0.0

info:

title: My API Specification

version: 1.0.0

components:

securitySchemes:

api\_key:

name: apiKey1

type: apiKey

in: query

parameters:

parametre:

name: ID

in: query

description: "identifiant d’un item"

required: true

schema:

type: "integer"

format: "int32"

schemas:

etudiant:

type: object

properties:

matricule:

type: string

example: MH54PP55232

nom:

type: string

prenom:

type: string

dateNaissance:

type: string

example: 01/01/2021

paths:

/etudiants:

get:

tags:

- Operations

parameters:

- $ref: "#/components/parameters/parametre"

responses:

200:

description: une d’étudiant

default:

description: Autre reponse

#Cas 2 : sécurisée GET

security:

- api\_key: []

#Cas 1 : sécurisée l'API

security:

- api\_key: []

**Figure 1.15** : Exemple de l’opérateur **GET** et sécurisation de l’API

Ce code décrit l’opérateur **GET**, sa réponse, la structure du succès (200) d’une requête et default pour les autres, les manières de description de sécurité cas 1 (sécurisée l’API) cas 2 (sécurisée une opération) et comment référencé un type défini dans components (paramètre ou autre).

1. **Son utilité** :

La spécification de *OpenAPI* prépare le terrain pour les consommateurs des services API en mettant à leurs dispositions les outils nécessaires pour l’utilisation de l’API. Elle précise comment formuler une requête prise en charge par l’API, de quoi s’attendre comme réponse d’une requête, quelle sont les différents champs, quelle sont les contraintes de sécurité (méthodes d'authentification) et comment avoir l’accès, ce qui permet aux clients de bénéficier une plus grande clarté dans le processus de conception. En cas de besoin, les informations de contact pertinentes, la licence et les conditions d'utilisation son présents. Avec ses outils cette spécification est tout en un car il permet aussi la génération automatique de la documentation à partir du code et tout changement dans le code met à jour automatiquement la documentation ; les deux sont donc étroitement liés [9].

1. **Conclusion** :

La spécification de *OpenAPI* est essentiellement le manuel de référence pour garantir aux développeurs une excellente expérience avec la conception et le développement d'API, elle aide les utilisateurs et les développeurs d'API à définir clairement les attentes lors de la consommation et de la construction d'une API. Elle fournit une la description explicite de chaque partie d’une API d’où la compréhension globale du comportement d'une API.

OpenAPI permet aux développeurs d’API REST de se familiariser avec les outils ce qui réduit le temps nécessaire d’intégrer des nouveaux développeurs et consommateurs d'API.

Dans le chapitre suivant nous décrirons une nouvelle architecture appelée publish/subscribe, son avantage pour les systèmes distribués et la présentation d’un des protocoles populaires basés sur ce modèle.