**Introduction Générale**

Sur le web d’aujourd’hui l’architecture avec API est une l’une des architectures le plus réputée suite aux outils de description d’API comme Swagger, Postman (une plate-forme pour la création, le test et l'utilisation d'API), etc. Alors qu’il y a 20 Ans de cela que Swagger n’existait pas, chaque entreprise avait leur propre compréhension, description et documentation d’API. Au fil des années certains experts utilisateurs et développeurs ont exigé la normalisation des API pour simplifier la compréhension et l’implémentation. Car le défi était de mettre en place une manière d’uniformisation pour une consommation simple. De ce fait, les APIs synchrones basée sur le protocole http ont été standardisés et outillés avec l’écosystème Swagger et *OpenAPI* que nous utilisons dans nos micro services et applications pour générer une interface utilisateur, pour la documentation et la collaboration [1]. Swagger étant un ensemble d’outils permettant de travailler avec *OpenAPI*. *OpenAPI* est la spécification open source indépendante du langage de programmation, lisible par les humains ainsi que les machine. Cette spécification sert d’uniformiser les API http tout en offrant une implémentation simplifiée pour les développeurs. Quant aux APIs asynchrones, ils ont remarqué une variété de protocole (AMQP, MQTT, Kafka etc.) différent qui complexifie une unification globale. Ce qui a conduit à voir différentes implémentations architecturales. Malgré cette différence d’architecture des APIs asynchrone aux synchrones, elles ont des objectifs en commun tels que : l’indépendance en langage de programmation, simple à implémenté etc. Vu que les APIs asynchrones sont en traines de gagner en exposition au sein de la communauté API suite à leur partenariat avec Postman. Il est donc crucial d’établir un standard de spécification compatible aux différents protocoles pour décrire uniformément ces interfaces asynchrones. Pour mettre en place une spécification efficace asynchrone, nous devons s’intéresser à un modèle asynchrone. Le modèle publication/abonnement (publish/subscribe ou Pub/Sub) est un modèle asynchrones efficace. Depuis qu’il est connu pour la diffusion d'information, la contribution dans la recherche et dans la commercialisation des produits, il est utilisé dans plusieurs contextes d'applications comme IoT (internet of Things). Les systèmes de communication dotés de ce paradigme (pub/sub) ont connu un gain de popularité significatif et ont montré son efficacité dans un grand nombre de domaine. La conception de base de pub/sub dérive de la manière dont les notifications circulent des expéditeurs aux destinataires et prend son efficacité de la manière dont l'information est communiquée. C'est-à-dire que, les destinateurs expriment leur intérêt en émettant des abonnements pour des notifications spécifiques, indépendamment des éditeurs (Publisher) qui les produisent, puis ils sont notifiés de manière asynchrone pour toutes les notifications concernant son abonnement [2]. Pub/Sub est modèle dont nous nous sommes intéressés pour établir une spécification pour les API asynchrones.

Toutes les APIs suivant cette architecture doivent offrir aux moins une fonction de publication (publish) et une fonction d’abonnement (subscribe). Donc notre but n’est pas seulement de mettre au point une spécification indépendante des langages de programmation ce qui est commun avec *OpenAPI*, mais aussi utilisable avec les différents protocoles asynchrones quel qu’il soit. Ce qui veut dire, qu’elle prend en charge le concept distribution des messages et aussi les langages de description API qui sont JSON et YAML. Mais supporter les variétés des protocoles (ce qui est généralement rencontré dans les applications IoT et similaire) reste important. Cette nouvelle spécification s’appelle *OpenPS*, elle est open source n’a pas d’outil comme Swagger pour documenter et générer les code développeurs. Elle permettra une uniformisation des APIs asynchrones au lieu que chacun se bat de sont coté pour créer une API IoT.

Pour une meilleure compréhension nous avons divisés ce travail en quatre (4) chapitre comme suit :

Dans le chapitre 1 de ce document, nous décrirons la spécification OpenAPI et l’outil de description, de documentation et génération de code de cette spécification qui est Swagger.

Dans le chapitre 2, nous présentons le modèle publication/abonnement et certain protocole de messagerie asynchrone de ce modèle.

Dans le chapitre 3, nous détaillerons une spécification (*OpenPS*) utilisant l’architecture de publication/abonnement pour la description et unification des APIs asynchrones.

Dans le chapitre 4, nous décrirons une spécification d’API asynchrone avec la spécification *OpenAPI* et générerons le code d’implémentation avec le générateur automatique locale de *OpenAPI* personnalisé appelé *OpenAPI-PS*.

# **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.

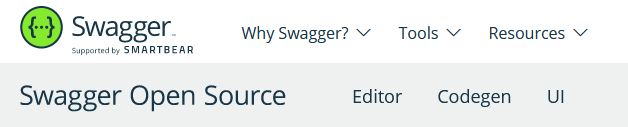
1. **YAML (Yet Another Markup Language)**:

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 :



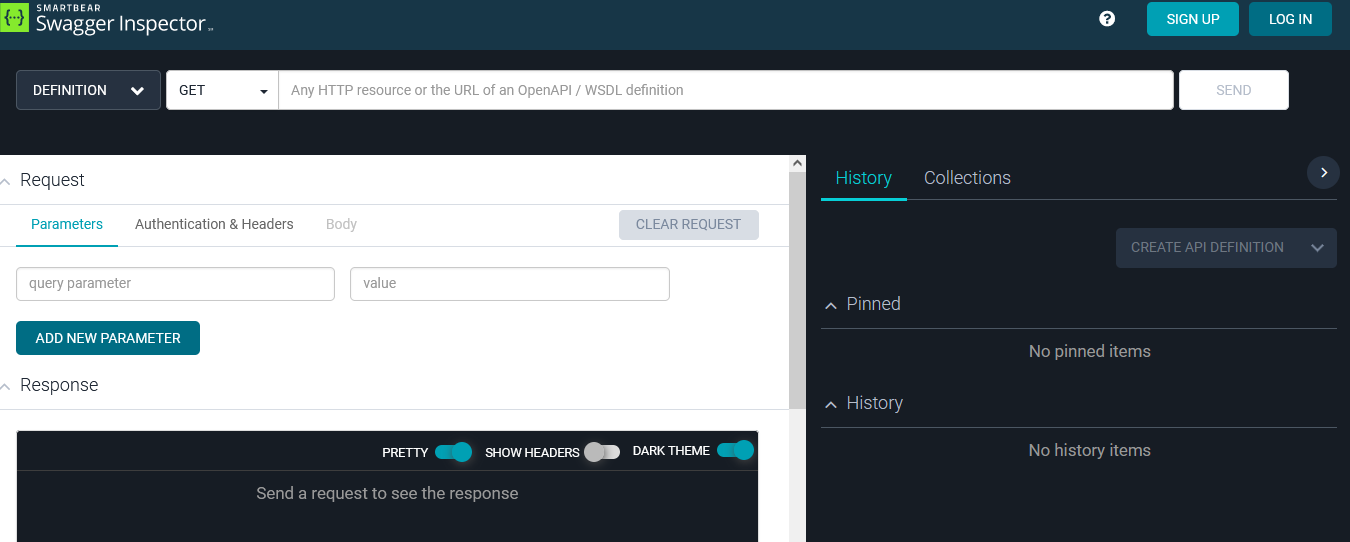
**Figure 1.11 :** différents outils de *Swagger*

* **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.

# **Chapitre 2 : le modèle publish/subscribe**

1. **Introduction** :

Aujourd’hui de nombreuses applications utilisent des modèles fortement orientés contenu et plus complexes que le modèle Web (client-serveur) pour répondre à des exigences de communication souhaité. En effet, les applications nécessitant ces modèles sont de plus en plus utilisées dans l’Internet d’aujourd’hui et sont par exemple celles de l’Internet des Objets (IoT) telles que les systèmes de gestion des bâtiments. Ou encore des nouvelles applications de management des réseaux informatiques d’entreprises. Un des modèles utilisant des communications orientées contenu est le publication/abonnement (publish/subscribe, ou Pub/Sub). Il est déjà largement utilisé dans les réseaux et applications d'entreprise, principalement en raison de son évolutivité et de la prise en charge de la topologie de réseau dynamique.

Cependant, le modèle de communication publication/abonnement offre une topologie de réseau dynamique, une communication orienté contenu et prennent en compte les contraintes (fréquences de connexions, nombres de système destinateur, criticités des ressources) ce qui le permet d’être utilisables par les applications telles que Internet des Objets (IoT), etc.

Dans ce chapitre nous décrirons ce modèle Pub/Sub, son importance pour les systèmes et applications distribués et quelques protocoles qui utilisent.

**Définition et description** :

1. **Définition** :

Pub/Sub est un modèle de communication asynchrone de service à service utilisée dans les architectures sans serveur central [11]. Avec ce modèle on sous-entend un éditeur qui envoie un message et un abonné qui reçoit le message via un courtier de message.

1. **Description** :

La communication Pub/Sub n’a pas besoin de connaître l’adresse source ou de destination. Les sources et les destinateurs sont appelés respectivement les producteurs(éditeurs) et les consommateurs (abonnés) de messages.

Les producteurs communiquent avec les abonnés de manière asynchrone à travers un courtier en diffusant des catégories de messages sans tenir compte de la façon dont ces messages seront traités. Pub/Sub diffuse ensuite des événements à tous les services qui y sont intéressés. Cette manière d’intégration augmente la flexibilité et la stabilité du système dans son ensemble car on n’a pas besoin des appels de procédures ni attendre que les abonnés reçoivent les données [12]. C’est un modèle dynamique et découplé (les expéditeurs de messages et les destinataires peuvent agir à des intervalles de temps différents) d’où son adaptation aux applications distribués à grande échelle et peut être implémenté comme pattern de mémoire partagée [13]. Ce paradigme est fonctionnel et flexible en application distribué avec la communication centrée sur le contenu.

Subscriber

Publisher

**Figure 2.1** : présentation de Pub/Sub

La communication Pub/Sub met en relation trois entités dont le producteur (*Producer*), le consommateur (Consumer) et le courtier (*broker*).

1. **Les producteurs** :

Récoltent les informations et les publient au courtier, au lieu de destiner les messages à un consommateur il les associe à une catégorie de message.

1. **Les consommateurs** :

Un consommateur envoie une suscription événementielle à une catégorie de message et toutes informations qui répondent aux contraintes de cet abonnement sont transmises par le courtier et plusieurs consommateurs peuvent s’intéresser aux mêmes informations.

1. **Le courtier** :

Il est l’entité qui assure la livraison des messages aux consommateurs au moment opportun. Il peut posséder une petite mémoire de stockage temporaire des messages. Un exemple de courtier utilisant le protocole Pub/Sub est AMQP est RabbitMQ [14].

1. **Les catégories de Pub/Sub** :

Pub/Sub est généralement divisé en deux catégories : à base de sujet et à base de contenu. Il existe une troisième catégorie basée sur le type d’événement, il est considéré comme étant une propriété dynamique des événements avec des variantes sécurités [15].

1. **Pub/Sub centré sur le sujet** :

Dans cette catégorie les éditeurs sont tenus d'étiqueter (identifiés par mots-clés) chaque événement avec un nom de sujet ; les abonnés s'abonnent aux événements sous un sujet particulier. Chaque sujet est considéré comme un service événementiel propre, identifié par un nom unique et chaque événement appartient à l’un d’un ensemble fixe de sujet [15]. Après l’abonnement à un sujet, le consommateur serait notifié de tout événement des producteurs de ce sujet ou thème. L’abonnement à un sujet T peut être consulté en tant que membre d'un groupe T, et la publication d'un événement sur le sujet T se traduit donc par une diffusion d’événement parmi les membres de T [15]. C’est un modèle utilisé par Cloud Pub/Sub [16].

1. **Pub/Sub centré sur le contenu** :

Dans les systèmes basés sur le contenu, chaque abonné définit une souscription selon la structure interne des événements et chaque producteur de données attribue un nom unique et sémantiquement significatif à chaque paquet de données qu'il génère. C’est un système utilisé dans le routage statique car même si les intérêts changes dynamiquement la technique de routage reste inchangée. Les consommateurs s'abonnent à des événements sélectifs en spécifiant des filtres d’abonnement. Ces filtres définissent des contraintes, généralement sous la forme de paires nom-valeur de propriétés et d'opérateurs de comparaison de base (=, <, …), qui identifient [15]. NDN-PS (Named Data Network Pub/Sub) est un protocole qui utilise cette manière de distribution de données [17].

1. **Pourquoi Pub/Sub** :

Pub/Sub est un modèle centré courtier dans lequel il n’y a pas de saturation de plage d’adressage. Il est adapté aux systèmes distribués et aux systèmes ayant en communication plusieurs milliers d’autres appareils fixes ou mobiles comme Internet of Things (IoT). C’est une architecture pour un réseau qui ne cesse de se développer et qui accueille chaque jour de nouvel utilisateur qui échange des contenus continuellement, car elle offre à ses consommateurs la possibilité d’abonnement aux données permettant aux applications grand public (consommateurs) de recevoir efficacement des mises à jour [17]. Il est extensible, à faible consommation d’énergie et est sans serveur central ce qui assure la disponibilité des données et la tolérance aux pannes des équipements.

1. **Les protocoles qui utilisent Pub/Sub** :

Les protocoles les plus populaires utilisant le modèle de communication publication/abonnement des messages sont :

1. **Le protocole MQTT** :

MQTT (Message Queuing Telemetry Transport) [18] est un protocole léger, ouvert et simple pour transport de messagerie de publication/abonnement et Client/server. Il est conçu pour être facile à mettre en œuvre et est utilisable dans des environnements contraints tels que l’internet des objets. Il est basé sur TCP/IP, idéal pour la transmission des données utilisant une faible bande passante et est à faible consommation énergétique.

La principale caractéristique de MQTT est sa légèreté, ce protocole ne requiert que des ressources minimales et peut donc être utilisé sur de petits microcontrôleurs.

Son objectif est de faire dialoguer des équipements qui ne disposent que peu de ressources pour assurer une connexion permanente. Pour Fabien Pereira Vaz (gestionnaire technique de vente de Paessler AG), MQTT se démarque par sa souplesse et sa simplicité de mise en œuvre, en plus d'assurer une transmission de données bidirectionnelle. Le protocole prend ainsi de l'importance dans les technologies opérationnelles de l'industrie et c’est l’un des protocoles le plus utilisé dans l’internet des objets. Il utilise un courtier pour le relais des messages. Son processus se divise en quatre étapes distinctes : connexion, authentification, communication, terminaison et les accords définit pour garantir la livraison sont : plus d’une fois, moins d’une fois et exactement une fois.

1. **NDN-PS** :

NDN-PS (Named Data Network Pub/Sub) [17] est une nouvelle architecture qui fournit une communication centrée sur les données au niveau de la couche réseau. NDN implémente un modèle de communication demande-réponse asynchrone qui dissocie naturellement les producteurs et les consommateurs de données. Il définit deux types de paquets de couche réseau : intérêt et donnée. Chaque producteur de données attribue un nom unique et sémantiquement significatif à chaque paquet de données qu'il génère. Chaque consommateur émet un paquet d'intérêt avec un nom de données ou un préfixe de nom, qui est transmis en fonction du nom (préfixe). Pour chaque intérêt reçu, les producteurs NDN utilisent des stratégies de transfert pour décider où transmettre l'intérêt en tenant compte des politiques d'utilisation de la table de transfert et de la mesure des décisions de transfert précédentes.

1. **Pub/Sub sur Google Cloud** :

C’est un ensemble d’applications implémentées comme un ensemble de service connecté par la messagerie asynchrone. Le producteur crée le message et l’envoie pour un topic laquelle est appelé « ressource », le message est stocké jusqu’à l’acquittement de tous les consommateurs. Pour retirer ce message, une application consommatrice doit créer un abonnement pour ce topic, et Pub/Sub Cloud livre le message à tous ce qu’y sont abonnés. L’acquittement des messages permet la non livraison des messages plus d’une fois. Cette communication peut s’effectuée entre un producteur et plusieurs consommateurs, plusieurs producteurs et un seul consommateur et plusieurs producteurs et plusieurs consommateurs. Une application productrice peut-être n’importe quelle application capable d’envoyer une requête http aux APIs du Google. Quant aux applications consommatrices, elles doivent être non seulement capables d’envoyer une requête https, mais aussi accepter la requête POST de https [16]. C’est un modèle idéal pour un environnement où il y a beaucoup d’échanges de messages.

1. **Le protocole AMQP** :

AMQP (Advanced Message Queuing Protocol) [19] est un protocole réseau ouvert (open source) pour la messagerie. Il agit au niveau de la couche application autorisant un échange de messages entre deux parties avec utilisation d’une file d’attente (Queuing). Il est standard et peut être utilisé pour la mise en place d’une topologie d’échange de type publication/abonnement. L’objectif principale de ce protocole est de mettre en place un serveur de messagerie orienté message en utilisant la fille d’attente et l’interopérabilité pour permettre une production/consommation sans un problème de langage de programmation voire de système d’exploitation. Il est efficace dans l’interconnexion des objets hétérogènes. AMQP est plus adapté aux situations exigeantes : une faible connexion, la fiabilité de la communication et une importante sécurité [20]. Il est utilisé dans activeMQ (broker open source populaire multi-protocole dont AMQP) par la fondation Apache pour fournir son implémentation dans plusieurs langages tels que : C, C++, java... et d’autre plateforme comme IBM (MQLight) [20].

1. **Pub/Sub et Client/serveur** :

Pub/Sub est une architecture basée sur un composant central courtier (peut être un serveur). C’est un modèle dans lequel un consommateur n’a pas besoin de connaître les producteurs de donnée et un producteur n’a pas non plus besoin de connaître un consommateur de donnée.

Le modèle client/serveur est un modèle de communication d’internet à base de requête/réponse. Un client demande des informations ou des services et le serveur répond par les données ou services demandés. Le serveur doit connaître l’adresse du client pour pouvoir communiquer avec lui et inversement.

Pub/Sub

consumer

producer

requête

serveur

client

consumer

réponse

consumer

producer

**Figure 2.2** : comparaison Pub/Sub et client/serveur

**Quelques briefs description de cette comparaison** :

|  |  |
| --- | --- |
| **Pub/Sub** | **Client/server** |
| Centré sur le contenu | Centré sur l’adresse IP |
| Routage basé sur le contenu | Routage basé sur IP |
| Élimination du DNS | Impossible de fonctionner sans DNS |
| Données distribuées | Partages de données |
| Pas de serveur central pour les informations | Un serveur central est utilisé |
| QoS plusieurs vers plusieurs | QoS un client et un serveur |
| Communication asynchrone | Communication synchrone |
| Les deux entités peuvent débuter la communication. | Client débute la communication |

1. **Protocole à implémenter AMQP** :

Parmi les protocoles cités nous implémentons AMQP car c’est un protocole qui assure l’échange des messages chiffrés et assure l’interopérabilité entre les applications. Il utilise l’architecture publication/abonnement utilisé dans les systèmes de messageries ainsi que pour la gestion des périphériques IoT**.**

Une communication AMQP est un transfert de message pair-à-pair entre les nœuds d’un réseau. Elle est constituée d’un ensemble de conteneurs qui peuvent contenir plusieurs nœuds. Un conteneur peut être un broker ou une application cliente. Un nœud peut être un producteur ou un consommateur responsable d’un stockage ou d’une livraison de message.

Une communication entre deux conteneurs nécessite un établissement de connexion. Les messages envoyés lors d’une connexion sont fragmentés en un ou plusieurs trames. Une connexion est alors constituée de l’envoie d’une séquence de trame ordonnée car la réception de la dernière trame implique que les premières trames sont déjà reçues.

1. **Architecture AMQP** :
2. **Les entités de communication** :
3. **Nœuds** : les nœuds sont les entités responsables de production, de stockage, de livraison ou de consommation des messages. Il peut être un producteur, un consommateur ou une queue.

* **Source ou producteur** : est un nœud qui crée et envoie les messages.
* **Cible ou consommateur** : est un nœud qui récupère et traite les messages.
* **File d’attente (queue)** : est la destination ciblée pour le stockage des messages. Elle peut être plusieurs, chacune ayant un identifiant unique qui la différencie des autres et est réalisée à l’aide d’une mémoire. Elle permet de distribuer les messages aux différents consommateurs. Un message est stocké dans une file d’attente à condition que les règles de comparaison entre l’identifiant de la file d’attente et celui du message soient respecté.

Broker

Queue1

**Exchange**

Queue2

Queue3

**Figure 2.3** : broker, le gestionnaire des queues

1. **Conteneurs** : est un ensemble de nœud ou une application client avec ces producteurs et/ou ces consommateurs, ou un broker avec ces queues. Le placement des nœuds dans un conteneur peut être plat ou hiérarchique [20] :

**Topologie hiérarchique des nœuds**

**Topologie plate des nœuds**

b1

a

Node path /a

Conteneur

Conteneur

Node path a

a

b2

b

/

b

b3

c

Node path c

b4

c

Node path /b/b4

d

d

**Figure 2.4** : différentes topologies des nœuds dans un conteneurs

1. **Unité de communication AMQP** :

Une trame est l’unité de base échangée de AMQP et chaque trame est composée de trois parties distinctes :

Header

Body

Extended header

**Figure 2.5** : paquet AMQP

* Frame header (8 bits) : il est de taille fixe et contient la version protocole AMQP utilisé.
* Extended header (taille variable) : Le traitement de cette zone dépend du type de trame car il contient des données supplémentaires.
* Frame body (taille variable) : c’est le corps de la trame, il contient les données à transmettre et peut avoir différentes formes telles que : *open/close* (la trame d’établissement ou de fin de connexion), *begin/end* (établissement ou finalisation d’une session), *attach/dettach* (concerne les liens), *transfert* (pour envoyer un message), *disposition* (utilisé pour informer le récepteur des changements d’état de livraison).

1. **Connexion entre pairs** :

Tout connexion AMQP commence par les échanges TCP, ensuite chaque pair doit envoyer un paquet *open* qui indique ses limites et ses capacités pour établir les deux Chaines de connexion unidirectionnelle appelé chacun un lien AMQP. Cet échange TCP concerne les informations sur la version du protocole et de la sécurité. Ce lien peut être par exemple producteur-queue-consommateur. Ces deux liens de sens opposé constituent une connexion bidirectionnelle avec le control de flow d’où une session.

Session

conteneur

conteneur

Sessi sessionon

lien

lien

**Figure 2.6** : Session AMQP

La session est éphémère c’est-à-dire qu’après la déconnexion les pairs doivent refaire le même processus [19].

Pour la fermeture de la connexion les deux points de communication doivent envoyer un paquet *close* avec un code de fermeture de session. Pour cela chaque pair doit attendre avec un délai raisonnable pour la réception de ce paquet pour éviter toute erreur car c’est le dernier paquet envoyé.

1. **Contrôle de flow** :

Le point de terminaison de session attribue à chaque trame sortante un identifiant de transfert implicite à partir d'une séquence. Chaque point de terminaison de session maintient l'état suivant pour gérer les trames de transfert entrantes et sortantes. Le contrôle de flow détermine l’identifiant du prochain paquet entrant attendu et sortant, le nombre de paquet entrant et sortant tout en respectant la taille de la fenêtre (la taille maximale de paquet recevable par le consommateur) ce qui facilite la communication avec une faible bande passante. Cette connexion étant l’envoie d’une séquence ordonnée de trame, les deux entités vérifient le numéro de séquence car le contrôle de flow est basé sur le nombre de paquet échangé. Les deux points de communication incrémentent l’identifiant du paquet précédemment reçu comme l’identifiant du prochain paquet à envoyer et peut aussi décrémente la taille de la fenêtre.

Il exige aussi le contrôle de flow des liens car il possède un état qui indique si le lien est disponible pour l’envoie des messages et pour le mettre à jour l’envoyeur doit le modifier pour le receveur [19].

1. **Etat de la connexion et de la réception des messages** :
2. **Etat de la connexion** :

Toute connexion AMQP commence par les échanges TCP,

Il s’agit des étapes principales indiquées par les paquets de comment la communication se passe entre les pairs :

* Open/Close : cette étape implique l’envoie des paquets d’initiation ou de fermeture de la connexion ce qui veut dire que la connexion est soit ouverte ou fermée.
* Begin/End : pour l’établissement ou la fermeture de la session les pairs s’envoi ces paquets. A ce stade la session est ouverte ou fermée et la session fait partie de la connexion TCP établie.
* Attach/Detach : indique l’attachement ou le détachement d’un lien a une session.
* Send/Receive : indique l’envoi des paquets et leurs réceptions sur les deux liens établis.

1. **Etat de la réception des messages** :

Les messages reçus peuvent être de différente forme tel que :

* *Accepted* : le message est bien reçu et est traité par le consommateur
* *Rejected* : les messages invalides que les consommateurs ne peuvent pas traiter donc ils sont rejetés.
* *Released* : les messages invalides qui doivent être re-envoyé pour qu’ils puissent être traités.
* *Modified* : les messages non traités car ils ont été modifiés.
* *Received* : cet état indique à l’éditeur de re-envoyer le message mais il indique à la cible que le message ne sera pas re-envoyer [19].

1. [**La qualité de services (QoS)**](https://www.ibm.com/docs/en/ibm-mq/9.2?topic=reliability-message-delivery-amqp#q123795___qos) :

AMPQ est efficace, portable, multiport et réputé par sa fiabilité de livraison [21]. Il fonctionne bien dans les environnements multi client et obéit à des règles strictes, garantissant l'interopérabilité des clients de différents fournisseurs. Ce protocole offre trois manières de communication producteur/consommateur grâce au principe de réglage automatique (*setled*) de ces entités de communication. Ces trois manières sont :

* ***at-most-once*** : un seul envoi avec la possibilité qu'il soit manqué ; si un producteur choisit ce réglage il supprime sa copie du message avant que les files en acquitte. De même si un consommateur choisit cette option la file n’attend pas l’acquittement de leur part avant de le jeter.

consommateur

producteur

Transfert (delivery\_tag,setled, …)

**Figure 2.7**: QoS “at-most-once”

* ***at-least-once*** : livraison garantie avec la possibilité de duplication de message, contrairement au premier le producteur et la file d’attente attend respectivement l’accusé de réception de la file d’attente et le consommateur.

consommateur

producteur

Transfert (delivery\_tagsetled , …)

ACK (setled, …)

**Figure 2.8**: QoS “at-least-once”

* ***exactly-once*** : livraison unique garantie. Car avant l’envoie du message le producteur et le consommateur s’envoient quelques messages d’acquittement [21].

1. **Fonctionnement** :

AMQP est un protocole de communication entre les applications à travers un réseau. Il permet une communication asynchrone car l’émetteur et le consommateur n’ont pas à agir au même rythme et le consommateur n’est pas non plus obligé de traiter l’information et d’en accuser la réception. Il s’appuie sur le broker pour la distribution des messages aux différentes destinations. Les messages étant stockés dans une file d’attente, le producteur peut ainsi travailler sans avoir de période d’inactivité.

Les messages sont distribués dans les différentes files d’attentes en utilisant les identifiant de ces files et le *binding* des messages.

Broker

Queue

Echange

Consommateur

Producteur

Queue

Queue

**Figure 2.9** : éléments intermédiaires de AMQP

**Exemple d’application** :

Cet exemple met en place un producteur, un consommateur et un serveur RabbitMQ avec le langage Java en localhost.

**RabbitMQ** : est un server (ici broker = gestionnaire de queues) localhost de simulation du protocole AMQP qui écoute par défaut sur le port 15672 pour les versions 3.x.

Publie Consomme

**Producteur**

**Consommateur**

**RabbitMQ**

Vérifie

**Figure 2.10** : élément intermédiaire de RabbitMQ

Apres avoir lancé le programme producteur (**Prod**), il produit un message et ce message sera stocké dans la queue définie et visible dans **RabbitMQ**. Le programme consommateur (**Cons**) va ensuite vérifier la présence d’un message dans la même queue et le consomme (affiche le message trouvé) s’il y en a.

**NB** : Une exécution continue de **Prod** produit plusieurs messages qui seront stockés dans la même queue.

1. **Transmission des messages aux consommateurs** :

Après la production des messages, chaque message est transmis au broker par les producteurs et le broker à son tour livre les messages aux consommateurs. Le broker AMQP est constitué de l’*exchange* et des queues. Le transfert de message entre *l’exchange* et les queues s’effectue par la correspondance entre la *routing\_key* du message et le *binding\_key* des files d’attente.

Le protocole AMQP comporte quatre types *d’exchange*, ces quatre types deroutages sont :

* Direct exchange ou unicast : le message est livré à une seule file d’attente si la *routing\_key* correspond au *binding\_key* donc un seul destinateur. C’est une communication one-to-one.
* Fanout exchange ou broadcast : c’est une communication one-to-many car ce type d’exchange ne fournit pas de *routing\_key* donc le message est livré à toutes les files d’attentes.
* Topic exchange ou multicast : c’est comme direct exchange sauf que la comparaison entre les *keys* ne concerne qu’une partie des keys. C’est possible d’avoir une communication *many-to-many*.
* Header exchange : utilise l’en-tête des messages à la place de la routing\_key. Les informations que contient cet entête seront comparées au *binding\_key*.

Par défaut, c’est-à-dire default exchange, le *binding\_key* d’une queue est utilisé comme le *routing\_key* ce qui stockage directement les messages dans cette queue.

Si un ou plusieurs consommateurs s’abonnent à une queue pour un type de message. Le message leurs serait livré une fois son stockage dans la file d’attente.

1. **Inconvénient :**

Ce protocole n’est pas compatible à ses versions précédentes car actuellement il n’existe que deux versions qui indépendantes aux niveaux de broker, *binding* et *exchange* [22].

1. **Sécurité**:

AMQP est un protocole d’encodage binaire qui utilise un protocole de sécurisation SSL/TLS et de l’authentification SASL. Chacun de ces deux protocoles consiste en quelque échange de frame header suivies des négociations de TLS ou de SASL. Cette négociation est un échange de certificat de sécurité pour les échanges sur réseau informatique.

SSL/TLS permet de crypter les informations sur les différentes chaînes pour assurer la confidentialité des données [19].

SASL est un mécanisme d’authentification qui vérifie les différents accès aux informations et peut être par exemple (nom d’utilisateur et mot de passe).

1. **Conclusion** :

Les services de communication de nouvelle génération doivent pouvoir coopérer pour répondre à des besoins spécifiques tout en gardant leur autonomie. Les réseaux sont mis en place notamment dans le but de transférer des données d’un système à un autre ou de fournir des ressources partagées des serveurs, des bases de données ou d’une imprimante. Ceci nécessite de maîtriser leurs architectures.

Pub/Sub est une norme qui préconise comment les équipements devraient communiquer entre eux tout en permettant aux systèmes ouverts capable de respecter son architecture d’être apte à échanger des informations avec d'autres équipements hétérogènes et qui sont issus de constructeurs différents. Les protocoles de cette architecture fourni une connexion logique qui est à la base de ces échanges entre ces systèmes informatiques indépendants afin de répondre aux contraintes des applications et systèmes actuelles.

Dans le prochain chapitre, nous présenterons une spécification pour les APIs utilisant le modèle Pub/Sub. Plus précisément le protocole AMQP pour la génération des codes clients pouvant implémenter ces APIs.

# Chapitre 3 : la spécification *OpenPS*

1. **Introduction** :

Dans une architecture logicielle, une API a pour fonction de simplifier la vie du développeur en lui offrant des entrées et sorties de communication (Input/Output) standardisées et découplées de leur implémentation applicative. Ce rôle d’interaction pour assurer ces implémentations pose de problème aujourd’hui dans les architectures distribuées et plus complexe tel que publication/abonnement et même dans les IoT etc.

Les interfaces de programmation (API) étant une des parties intégrantes des systèmes distribués [23]. Pour simplifier leurs utilisations, un certain nombre de documentations sur les normes des API ont été proposées ; pour les services Web basés sur REST, La spécification OpenAPI [24] a été adopté. Les APIs basées sur http sont courantes sur le Web, résident entre le serveur et les clients, ne répondent qu’aux requête bien formulées, et fournissent des URLs vers les ressources. Néanmoins, La conception d’API pour l’architectures pub/sub comme on le fait pour des APIs d’architectures centralisées (client/serveur) serait surement une erreur de conception d’architecture logicielle. Car on passe d’une logique applicative centralisée à une logique applicative distribuée où la fonction de l’API s’en trouve transformée. Cette erreur est pourtant fréquente car on continue de la gérer au niveau de la conception d’application alors qu’elle devrait être gérée au niveau de l’interface de communication qu’est l’API [25] pour rendre l’interconnexion transparente.

Comparer aux applications web, IoT est encore nouveau et de nombreuses APIs IOT sont disponibles. Leur principal protocole est devenu un des protocole Pub/Sub (MQTT) qui leur permettra probablement une forte croissance dans le monde des APIs. Car leurs principaux fournisseurs comme Amazon, Azure et IBM fournissent à la fois des API http et MQTT pour accéder à leurs services [27].

A ce jour, étant donné l’existence de quelques APIs de type pub/sub [26], aucune spécification, ni outil d'installations ni test en ligne simples disponibles pour apprendre à utiliser les API IOT n’a encore été définie pour uniformiser la description des APIs pub/sub à la manière du modèle Web alors que, le développement d’application IoT ne cesse de croître. Ces APIs Pub/Sub bien qu'il existe plusieurs standards pour les systèmes pub/sub [28,29], communiquent à travers des canaux, fournissent des abonnements topics permettant aux clients la réception des informations à tout moment et intéressantes dans le monde des IoT. Cela implique que chaque projet d'API est unique et dépend des cas d'usages spécifiques. Pour le partage des bonnes pratiques d’implémentation, des ressources, et l’interconnexion d’application ou API, nécessite de mettre en place des règles communes qui seront respecter par chaque API d’où la bienvenue d’une spécification.

Dans ce chapitre nous décrivons une spécification open source pub/sub appelée *OpenPS* similaire à celle de *OpenAPI* permettant d’uniformiser les APIs, en utilisant le modèle pub/sub afin de minimiser l’intervention des développeurs dans la communication des applications.

1. **Définition** :

La spécification *OpenPS* définie une interface standard pour les APIs en se basant sur le modèle Publish/Subscribe. Elle assure l’interopérabilité et la lisibilité des codes par les producteurs, consommateurs ainsi que pour les machines. Elle se sert de quelques objets de la spécification *OpenAPI* et d’autres nouvel objet pour la description et la mise en place des APIs REST.

Avec les outils de documentation et de génération de code, une spécification *OpenPS* permet de générer le code du producteur et du consommateur dans divers langages de programmation.

1. **Constitution de la spécification *OpenPS***:

Tout comme la spécification *OpenAPI*, la spécification *OpenPS* est constituée d’un ensemble d’objet suivant une structure hiérarchique. Une spécification *OpenPS* est composé de sept (9) objet racine. Parmi ces objets la présence de quatre (4) sont exigé pour un minimum description d’un document *OpenPS*.

protocol

string

broker

object

info

object

openps

string

tags

object

security

object

components

object

topic

object

externalsDocs

object

**Figure 3.1** : différents objets de *OpenPS*

1. **openps** :

C’est une chaîne de caractère décrivant la version utilisée par les documents *OpenPS* permettant aussi leurs interprétations par les outils de cette spécification et fait partie des objets exigé par la spécification. Cette spécification est à la version « 1.0.0 »

1. **info** :

C’est un objet dont sa présence dans la spécification est obligatoire. Il fournit les informations sur l’entête de l’API, il est constitué d’autre objet dont les plus important sont :

* **title** : le nom de l’API.
* **version** : version du document *OpenPS* « 1.0.0 ».

1. **broker** :

C’est un objet exigé par la spécification appelé aussi courtier qui offre les services Publish/Subscribe. Sa valeur fournie des informations des entités cibles. Il exige la présence de certains objets conteneurs tels que :

* **name** : est une unique chaîne de caractère qui représente le nom du broker.
* **hostname** : représente le nom d’hôte du broker cible (ici localhost).
* **port** : un nombre entier qui désigne le port du broker cible.

1. **protocol** :

Il précise le protocole à implémenter pour la communication Publish/Subscribe par exemple le protocole AMQP.

1. **topics** :

Cet objet décrit le nom des topics disponible ainsi que les opérations de publication/abonnement pour l’API. Il correspond à l’objet « Path » de la spécification OpenAPI. Il exige que le nom d’un topic commence par un slash (/) sur lequel il n’y a que deux opérations disponibles et un objet message exprimant le format de l’information échangée sur ce topic. Il est le dernier objet exigé par cette spécification. Ces trois composantes constitutives sont les suivants :

1. **publish** :

Il définit l’opération publish (de publication) d’un message sur un topic. C’est un objet qui décrit les résumés sur un topic à publier, l’entité chargé de publier ce topic ainsi que les qualités de service pour la distribution de ce topic. Il définit ces informations à travers d’autres objets qu’il supporte. Ces objets sont présentés ci-dessous :

* **tags** : c’est un tableau de chaine de caractère dont l’item i correspond à la description de l’opération de catégorie i.
* **summary** : c’est une chaine de caractère désignant un petit résumé de ce que fait l’opération publish.
* **description** : c’est la description de l’opération publish en chaine de caractère.
* **entities** : définis le points de terminaison ou l’entité qui publie les messages.
* **QoS** : c’est une chaine indiquant comment le message est distribué ; les valeurs autorisées sont : *none, at-most-once, at-least-once, exactly-once*.

1. **subscribe** :

Il définit l’opération subscribe (abonnement ou récupération d’un message) sur un topic. C’est un objet qui décrit les résumés de l’abonnement sur un topic et l’entités chargé de récupérer les messages sur ce topic. Il fournit ces informations à travers les mêmes objets que publish mais ne prend pas en charge la qualité de services QoS.

1. **message** :

Il contient trois (3) composants permettant de définir le format et le contenu d’un message.

* ***description*** : un petit résumé en chaîne de caractère décrivant le message.
* ***content*** : c’est un tableau *map*(*string*, *object*) permettant de définir le contenu d’un message
* ***Required*** : un booléen qui indique si le corps du message est exigé, il a la valeur false par défaut.

1. **components** : c’est un objet conteneur qui permet à la description d’éviter la duplication du code des opérations qui ont des paramètres en commun, et tout en le référençant dans l’opération en question avec la propriété $*ref* qui a un chemin comme valeur depuis l’objet racine. Il correspond au même objet « *components »* de la spécification *OpenAPI*.
2. **security** : il respectera toutes les règles de celle de *OpenAPI* mais pas encore défini.
3. **tags** : le même que la spécification *OpenAPI*.
4. **externalsDocs** : le même que la spécification *OpenAPI*.
5. **Les langages utilisés** :

Comme la spécifications *OpenAPI*, la spécification *OpenPS* supporte deux langages de description des APIs qui sont : JSON et YAML.

1. **L’outil utilisé par *OpenPS* :**

La spécification *OpenPS* utilise les mêmes outils que la spécification *OpenAPI*, il s’agit de Swagger UI, Swagger Editor et Swagger Codegen. *OpenPS* modifie les fichiers Codegen pour subvenir à ses fins et utilise la manière d’ajout des extensions de la spécification OpenAPI (x-) pour définir les différentes opérations. Avec Swagger Editor et Swagger UI nous décrivons, éditons et générons nos documents APIs et avec Swagger Codegen générons des clients permettant d’interagir avec notre API.

1. **La syntaxe de la spécification *OpenPS*** :

Cette syntaxe indique comment les objets de cette spécification se combines et comment spécifier les valeurs des différentes propriétés.

Les objets qui n’ont pas encore été définis tels que *security* ne seront pas présenté dans la syntaxe ci-dessous de cette spécification.

openps: 1.0.0

info:

  version: 1.0.0

  title: Multiverse Telemetry API

  description: A simplified version of the Controller-Agent pub-sub API for telemetry.

brokers:

  - name: broker1

    description: First AMQP broker.

    hotsname: 'broker1.multiverse-nms.com'

    port: 5672

protocol: 'AMQP 1.0'

topics:

  '/capability':

    publish:

      entity: agent

      description: Agent advertizes its capabilities.

      qos: 'none'

    subscribe:

      entity: controller

      description: Controller collects agents measurement capabilities.

    message:

      required: true

      content:

        application/json:

          schema:

            $ref: '#/components/schemas/Capability'

components:

  schemas:

    Capability:

      description: A capability provided by the agent.

      allOf:

        - type: object

          required:

            - capability

          properties:

            capability:

              description: Verb of the capability message.

              type: string

              enum:

                - measure

externalDocs: ‘C:\Users\OASC\Desktop\Projet\chapitres\openps- specification.doc’

**Figure 3.2** : extrait de la spécification *OpenPS*

1. **Conclusion** :

Les API connectent les machines et les humains. Ce sont les éléments constitutifs de l'écosystème numérique connecté du 21e siècle, travaillant en arrière-plan pour tout créer, des jeux vidéo aux appareils médicaux etc.

La spécification *OpenPS* crée un cadre commun pour l'ensemble du cycle de vie de l'API afin de définir les fonctionnalités des service RESTful, y compris leurs ressources. *OpenPS* est lisible à la fois pour l'homme et pour la machine. Elle agit comme le modèle de l'ensemble du développement et de la livraison des APIs permettant de communiquer la valeur et la fonctionnalité des APIs, à la fois aux parties prenantes internes et aux consommateurs externes. Elle est open source offrant avec ses outils plus de compréhensibilité pour aider les développeurs à prendre plus d’avantage de toutes les capacités d’une API.

Dans le prochain chapitre, nous décrirons l’architecture de la spécification *OpenPS*, nous présenterons quelques outils utilisés pour la génération automatique de code. En utilisant comme entré (input) un document de la spécification *OpenPS* (en YAML) et un cas d’utilisation de *OpenPS*.

# **Chapitre 4 : Un générateur de code pour *OpenPS***

1. **Introduction** :

La génération de code est une opération consistant à générer automatiquement le code. Son but est de minimiser les risques d'erreurs de programmation. Il permet la production de code source (répétitif) afin d’aider le programmeur de se concentrer sur l'écriture de code. Actuellement il existe plusieurs générateurs de code et l’un des plus connu sur le web est le générateur de code OpenAPI. Ce générateur peut générer différents codes en différents langage de programmation mais il exige comme entrée un document de définition d’une API et plus précisément un document de la spécification *OpenAPI* que nous avons présentés dans le chapitre 1 de ce mémoire.

La spécification *OpenAPI*, tout comme la spécification *OpenPS*, est une spécification open source pour les APIs qui utilisent respectivement un modèle de communication web ou asynchrone. Contrairement à *OpenAPI*, la spécification *OpenPS* présentée dans le chapitre 3, n’a pas encore de générateur de code clients spécifique afin de simplifier l’implémentation des APIs. Pour cela nous mettons en place une approche afin de s'assurer que ces APIs répondent aux exigences définies dans leurs spécifications *OpenPS*. Cette approche consiste à mettre en œuvre un générateur de code automatique pour une plate-forme cible avec un langage de programmation cible. Pour cela nous nous servons du générateur *OpenAPI* comme référence principale. Nous avions deux méthodes, soit utiliser le générateur OpenAPI, soit le personnaliser. Le générateur de codes *OpenAPI* est un générateur open source qui génère le code client et l’implémentation serveur en utilisant comme entrée un fichier de spécification en format YAML ou JSON. Ce générateur consiste au minimum en quatre (4) ensembles de fichiers : Codegen file, SPI registration, template et config file. Il permet la génération automatique de code d’implémentations API. Cette génération de code permet de respecter les modèles de base de communication décrits par les templates. Le code obtenu après une génération de code peut être aussi bien du code source prêt à être modifier ou compilé ou même exécutable.

Le but de ce chapitre, est de présenter l’approche que nous avons adopté pour générer un code d’implémentation client d’une API locale et ces *dependencies* avec générateur personnalisé de *OpenAPI* tout en respectant la modèle de communication publication/abonnement.

Dans ce qui suit, nous décrirons les fichiers constitutifs de ce générateur de code.

1. **Codegen file** : *OpenAPI* codegen

*OpenAPI* Codegen [30] est un projet open source qui permet de générer automatiquement des bibliothèques clientes d'API, génération de SDK (*Software* *Development* *Kit*), simulation des services et de la documentation à partir d'une spécification *OpenAPI* [31].

Codegen est une classe java qui contrôle les objets racines de la spécification *OpenAPI.* Ce fichier permet de créer les opérations, le modèle de fichier (différent lien ou racine d’importation d’une classe dans une autre) et les fonctions prédéfinies dont le contenu doit uniquement être modifié (preProcessor, etc.).

*OpenAPI* Codegen est un outil apporté parmi d’autre (Swagger Editor, UI) pour la génération de code d'API intégrées, conçues pour les équipes d'API travaillant avec la spécification Swagger (*OpenAPI*). En tant qu’outil de générateur de code limité, son intention est de générer et tester les codes dans un ensemble de modèle et de langage. C’est un outil pour faciliter l’unification et la génération des frameworks. Avant la génération du code, codegen interprète les codes et les instructions dans les fichiers templates et dans les fichiers POM (*Project Object Model*). Les informations que contiennent ces templates sont appelées des règles [32] qui décrivent l’objectif et celles dans les POM permettent la description du projet généré et les librairies utilisées.

C’est un des projets approuvés par des meilleurs développeurs [33].Pour l’utiliser à ses propres fins, *OpenAPI* Codegen doit être importé localement.

1. **Templates files** : *Mustache*

Les fichiers templates contiennent des textes qui définissent une part de code source généré. Ils contiennent les codes sources pour le langage cible combiné avec des tokens (variables ou nom d’objet qui peut être utilisé dans les templates) appelés par le codegen. Ces fichiers templates permettent de décrire un modèle et ont des extensions « .*mustache* ».

« Mustache » est une spécification pour les langages de modélisation [34]. Il décrit les manières d’affichage et import des exception (dans le cas où le code généré aura besoin). Il est implémenté dans plusieurs autres langages de programmation [35]. Mustache se sert des accolades pour indiquer les champs. Ces accolades sont à l’origine de son nom car en leurs rotant à 90° (dans le sens contraire de l’aiguille d’une montre) il ressemble à la moustache.

90°

**Figure 4.1** : obtention du logo de « Mustache »

Ces champs contiennent des variables appelés des entrées qui seront remplacés par leurs valeurs (si elles existent) lors de la génération de code.

Fichier.mustache Résultat

Bonjour UMMTO

Bonjour {{universite}}

Entrée

{

" universite ": "UMMTO"

}

**Figure 4.2** : fonctionnement du générateur avec les templates *Mustache*

C’est un langage moins logique car il n’effectue pas les opérations (+,-,\*,/,> …), les conditions ni les boucles [34]. Il est cependant possible d'afficher ou de masquer sous condition un bloc de texte à travers une section.

1. **Une section Mustache avec la condition (**si et sinon**)** :

Une section commence par {{#nomSection}} et se termine par {{/nomSection}} et la section inverse est faite par le même nom que la section : {{^nomSection}} et se termine par {{/nomSection}}.

Fichier.mustache

Bonjour

{{#maCondition}}

département informatique

{{/maCondition}}

{{^maCondition}}

Tizi-Ouzou

{{/maCondition}}

Résultat

Bonjour Tizi-Ouzou

Entrée

{

"maCondition": false

}

**Figure 4.3** : condition (if\_else) en *Mustache*

1. **Une boucle** :

Une boucle en langage mustache est faite avec les tableaux c’est-à-dire que mustache se comporte comme une boucle quand l’entrée est un tableau. Le nombre d’itération de la boucle dépend du nombre d’entrée (indice) du tableau.

Fichier.mustache Résultat

**U**

**M**

**M**

**T**

**O**

{{#universite}}

{{.}}

{{/ universite}}

Entrée

{

" universite " : ["U", "M", "M", "T", "O"]

}

**Figure 4.4** : boucle en *Mustache*

1. **Inclusion des fichiers**:

En mustache un fichier template peut charger d’autres fichiers templates en écrivant comme suit {> nom\_template2}}

1. **POM files** :

POM est un acronyme de « Project Object Model » qui représente Maven fondamental pour le bon fonctionnement des projet java local [36]. Comme le code à générer est une implémentation java qui dépend de quelque librairie Maven. Chaque projet ou sous-projet Maven est configuré par un POM qui contient les informations nécessaires à Maven. Il contient en général le nom du projet, numéro de version, dépendances vers d'autres projets, bibliothèques nécessaires à la compilation, noms des contributeurs, des ressources comme images etc. Le POM se présente par un fichier « pom.xml » à la racine d’un projet java. Sa représentation XML est traduite par Maven en une structure de données qui représente le modèle du projet. Il s'appuie sur des repositories de librairies (jar) locaux ou accessibles via HTTP ; à sa première exécution, Maven télécharge les différents plugins dont il a besoin et les installe dans le répertoire « maven/repository » situé dans le répertoire de travail de l'utilisateur. Grâce à la définition du projet dans le fichier pom.xml Maven peut gérer le cycle de vie du projet (compilation, test, packaging, installation...). Pour un fichier POM l’exigence minimal sont les suivantes :

1. Project : qui est la balise racine. Il est suivi en générale de quatre (4) liens http de Maven apache :

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0

https://maven.apache.org/xsd/maven-4.0.0.xsd">

**Figure 4.5** : les liens des dépôts **Maven**

1. ***modelVersion*** : qui doit être 4.0.0.
2. ***groupId*** : l’identifiant du groupe du projet.
3. ***artifactId*** : qui implique le nom du projet.
4. ***Version*** : la version du projet en générale 1.0.0 pour la première fois. Elle indique si le projet est à terme ou en cours du développement en ajoutant « SNAPSHOOT ».
5. **SPI registration**:

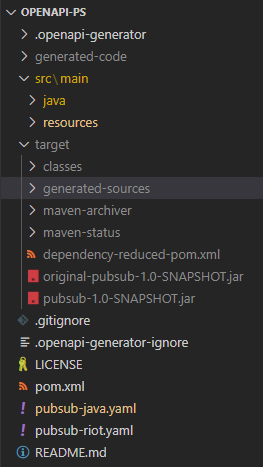
C’est un fichier de configuration OpenAPI codegen approuvé par un ensemble de développeurs sur internet. Il contient le nom des différent fichier *OpenAPI* codegen pour la génération de code en différents langages.

1. **Config file** :

Ce sont des fichiers YAML de *OpenAPI* contenant des exemples réels. Ce qui permet aux utilisateurs de testes la sortie de leur code.

1. **Architecture du générateur *OpenPS*** :

*OpenPS* est une spécification open source qui n’a pas encore de générateur de code agréé. Pour générer les codes clients pour les APIs Pub/Sub, nous utilisons le générateur de la spécification *OpenAPI* afin de décrire une spécification *OpenPS* qui sera acceptée par le générateur *OpenAPI* localement. L’objectif est de permettre la génération automatique de code d’implémentation pour un broker cible pour les producteurs et consommateurs. Ces entités doivent communiquer en utilisant des fonctions pub/sub générer (publish et subscribe). En adoptant un modèle de base de communication décrit par les templates et charger les librairies nécessaires pour cette communication.



**Figure 4.6** : Architecture ***OPENAPI-PS***

Ce générateur de code est disponible sur « ***GitHub »*** à l’adresse « <https://github.com/amar-ox/openapi-ps> »

Les fichiers et dossiers racines sont :

1. ***Generated-code***: c’est le dossier indiqué par la commande lors de la génération de code. Il contient l’ensemble du code généré.
2. ***Src\main***: ce dossier contient l’ensemble des codes des entités, templates et le codegen. Il contient deux dossiers qui sont : java/org/ps et ressource.

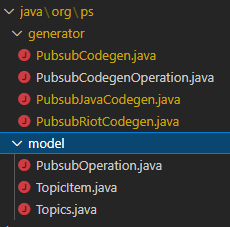
* ***Java/org/ps*** : contient les différents codegen. C’est dans ce fichier que le codegen des langages cible doivent être placés. Ici on a le codegen de java et riot.
* ***Ressource*** : il contient l’ensemble des templates et le POM des entités qui seront générés.

1. ***Target***: il contient les même information que Src/main plus les fichier de configuration Maven.
2. **Pubsub-java.yaml** et **Pubsub-riot.yaml**  : c’est le fichier YAML de description de l’API utilisée. Le codegen s’intéresse à ce fichier pour générer le code d’implémentation. Ici on a le code d’implémentation **java** et **riot**.
3. **Licence et POM** : le fichier licence contient la version, la définition et la politique du serveur « apache ».

Le fichier POM contient la définition et les configurations Maven du générateur de code ***OPENAPI-PS***. Ce fichier gère la racine du projet et le lien depuis lequel on peut importer une classe ou un projet généré dans un autre.

1. **Codegen** : c’est le fichier codegen *OpenAPI* personnalisé et les deux autres codegen java et riot et gestionnaire des opérations dans le dossier « *generator* ». Ces fichiers permettent de vérifier la présence des objets exigés dans la spécification *OpenAPI* et crée les classes (fichier java) nécessaires en se basant sur les templates.

Le fichier model contient les fichier contrôlant les objets racines de OpenAPI.

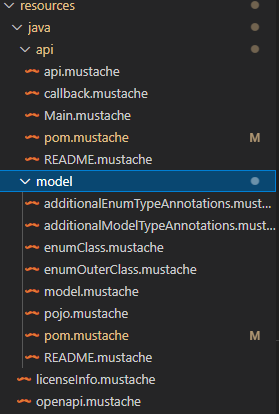


**Figure 4.7** : Différent codegen de ***OPENAPI-PS***

1. **Templates et POM** :

Le dossier ressource contient deux dossiers qui sont « api » et « model » décrit comme suit :

* **Api** : les fichiers du dossier « api » permet de définir les différentes entités. Chaque fichier de ce dossier est une classe qui serait créé avec ses fonction dans chaque entité.
* **Model** : les fichiers que contient model permettent de décrire les topics, leurs propriétés et énuméré les valeurs autorisées par les propriétés. Il contient aussi les différentes annotations permettant l’exécution du projet même si toutes propriétés facultatives n’ont pas de valeurs.
* Chacun de ces deux dossiers contient un fichier POM qui définit le projet et charge toutes les bibliothèques (*dependencies*) et autre projet dont ces projets ou entités dépendent.



**Figure 4.8** : les fichiers **Mustache**

1. **Présentation d’un fichier Mustache :** *callback.mustache* est un fichier décrit en mustache qui représente une classe java dans chaque entité contenant des fonctions callback du broker utilisé.

{{>licenseInfo}}

package {{package}};

{{#imports}}

import {{import}};

{{/imports}}

public class {{classname}}PubsubCallback {

    public {{classname}}PubsubCallback() {}

    {{#operations}}

    {{#operation}}

    {{#isSubscribe}}

    public void {{operationId}}{{#allParams}}Callback({{{dataType}}} {{paramName}}{{/allParams}}) {

        // Not implemented

    }

    {{/isSubscribe}}

    {{/operation}}

    {{/operations}}

}

**Figure 4.9**: contenu du fichier *callback.mustache*

1. **Présentation d’un fichier POM** : le projet Models décrit sans les *dependencies*.

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0

https://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>

   <groupId>{{groupId}}.models</groupId>

   <artifactId>Models</artifactId>

   <version>{{version}}</version>

   <packaging>jar</packaging>

   <name>{{appName}}</name>

   <properties>

      <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

      <java.version>1.8</java.version>

      <maven-compiler-plugin.version>3.8.1</maven-compiler-plugin.version>

    </properties>

<dependencies>

</dependencies>

</project>

**Figure 4.10** : un fichier **POM** décrit en *Mustache*

1. **Un cas d’utilisation** :

Nous avons mis en place une spécification *OpenPS* en utilisant la spécification *OpenAPI* et qui est acceptée par le générateur *OpenAPI* personnalisé. Le but est de générer un code d’implémentation du broker RabbitMQ en langage java avec des fonction publish et subscribe de deux entités Agent et Controller. Le code généré doit posséder les outils nécessaire (librairies) c’est-à-dire doit être exécutable. Pour cela nous avons installés **Erlang** (opt\_win) qui permet le bon fonctionnement de RabbitMQ. Par la suite nous avons installés **RabbitMQ** avec le plugin d’interface utilisateur « rabbitmq-plugins enable rabbitmq\_management » à travers la ligne de commande RabbitMQ. Cette interface d’utilisateur peut être visualiser en localhost au port 15672 avec « *guest*» comme password et username. Elle permet de voir après l’exécution du code généré les files d’attentes avec les différentes abonnement et publications sur les topics existants.

1. **Les outils utilisés** :
2. **RabbitMQ** : c’est le broker local utilisé qui permet la création et gestion des files d’attentes. Son implémentation en java nécessite quelque librairie (voir figure 4.21) et de gestion d’exception java.
3. **Erlang (OTP\_win)** : Erlang est un langage de programmation utilisé pour l’évolutivité dans les systèmes temps-réel et qui demande une grande disponibilité de données.

**OTP** est un ensemble de bibliothèque d’Erlang permettant qui possède sa propre base de données distribués et permettant d’interfacer les applications de différent langage et **win** pour indiquer les version Windows [37]. Il permet le bon fonctionnement de **RabbitMQ** dans notre implémentation.

1. **Visual studio code** : est un environnement de développement intégré (IDE) extensible, complet et gratuit pour créer des applications modernes [38]. C’est l’environnement où est installé le générateur de code ***OPENAPI-PS***. Il nous permet la génération de code après l’installation des Maven.
2. **Eclipse** : Eclipse est un environnement de développement (IDE) Java, qui peut également être utilisé pour d'autres langages de programmation, dont le C/C++ et le PHP en utilisant des plugins.

Eclipse nécessite une machine virtuelle Java (JRE) pour fonctionner et utilise un kit de développement (JDK) pour compiler.

Nous l’utilisons pour tester le code généré par ***OPENAPI-PS***.

1. **Le fichier YAML utilisé comme entrée** :

La spécification *OpenAPI* utilisée comme entrée est la spécification **pubsub-java.yaml** à la racine de l’architecture ***OPENAPI-PS***. Cette spécification concerne deux entités un « agent » qui peut être un programme serveur et un « controller » qui est supposé être un client. La spécification « **pubsub-java.yaml »** est choisie en indiquant le nom du fichier YAML dans la commande de génération de code.

Ce fichier YAML peut être divisé en quatre (4) parties : l’entête de l’API, la description du serveur, la description des « *topics* » et le « *component* ».

1. **Entête de l’API** :

Cette partie est équivalente à l’entête API de *OpenAPI*. Elle concerne les objets racines « openapi » et « info ». Elle décrit la version de la spécification, la version de l’API et le nom de l’API (le titre).

openapi: 3.0.0

info:

  version: 1.0.0

  title: Multiverse Telemetry API

  description: A simplified version of the Controller-Agent pub-sub API for telemetry.

**Figure 4.11** : entête de la description de l’API pub/sub

1. **La description du serveur** :

Il s’agit du serveur utilisé. Comme c’est la description d’une API asynchrone de publication/abonnement c’est un broker et l’URL de sa description.

servers:

  - url: http://multiverse.com:4534

    description: Main AMQP broker.

**Figure 4.12** : description du serveur utilisé

1. **La description des « *topics* »** :

Cette partie est l’objet « paths » car c’est la spécification *OpenAPI*. Elle peut être divisé en deux parties : « /capability » et les autres « topics » comme « /specification », « receipt », etc.

* /capability : il s’agit des capacités du serveurs pour débuter la communication. Il est publié par le serveur décrit (agent) et un client (controller) peut l’avoir par abonnement sur ce *topic*.

paths: # topics

  '/capability':

    servers:

      - url: http://capability.multiverse:4534

        description: Capability AMQP broker.

    x-ps-publish:

      entities:

        - agent

      description: Agent advertizes its capabilities.

      qos: 'once'

    x-ps-subscribe:

      entities:

        - controller

      description: Controller collects agents measurement capabilities.

    x-ps-content:

      application/json:

        schema:

          $ref: '#/components/schemas/Capability'

**Figure 4.13** : Description ***OpenAPI*** du *topic* de capacité serveur

* **Les autres « *topics* »** : ces « *topics* » sont des opérations qu’un client (controller) peut publier et la description de la réponse et l’acquittement (agent) que le serveur publie.

Les opérations sont : la « /*specification* » qui indique les tâches que le client (controller) veut effectuer et « interrupt » sert d’interrompre une tâche qui s’exécute sur le serveur (agent) par la demande du client (controller).

La réponse du serveur (agent) est le « /result » et l’acquittement est le « receipt ».

  '/specification':

    x-ps-publish:

      entities:

        - controller

      description: Controller publishes a request towards agents.

      qos: 'at-least-once'

    x-ps-subscribe:

      entities:

        - agent

      description: Agents listen to incoming measurement requests.

    x-ps-content:

      application/json:

        schema:

          $ref: '#/components/schemas/Specification'

  '/result':

    x-ps-publish:

      entities:

        - agent

      description: Agents publish measurement results.

      qos: 'once'

    x-ps-subscribe:

      entities:

        - controller

      description: Controller receives measurement results.

    x-ps-content:

      application/json:

        schema:

          $ref: '#/components/schemas/Result'

  '/interrupt':

    x-ps-publish:

      entities:

        - controller

      description: Controller stops active mesurements.

      qos: 'at-least-once'

    x-ps-subscribe:

      entities:

        - agent

      description: Agents receive interrupt to stop an active measurement.

    x-ps-content:

      application/json:

        schema:

          $ref: '#/components/schemas/Interrupt'

  '/receipt':

    x-ps-publish:

      entities:

        - agent

      description: Agents sends a receipt for measurement or an interrupt.

      qos: 'at-least-once'

    x-ps-subscribe:

      entities:

        - controller

      description: Controller receives a receipt for a measurement or interrupt.

    x-ps-content:

      application/json:

        schema:

          $ref: '#/components/schemas/Receipt'

**Figure 4.14** : description ***OpenAPI*** des autres topics client ou serveur

1. ***Component*** : cette partie décrie les propriétés communes des différents topics ainsi que les propriétés propres à chaque topic. C’est cette partie qui décrit un exemple de valeur qui peut être prise par une propriété et sont référencé dans les topics correspondants par « **$ref** » de *OpenAPI*.

La partie du component décrivant les propriétés communes est :

components:

  schemas:

    CommonFields:

      type: object

      required:

        - name

        - type

        - operationId

        - when

        - ts

        - target

        - resultColumns

      properties:

        name:

          type: string

          description: Name of the capability. Must be unique and constant over related messages.

        type:

          type: string

          description: Type of the measurement (e.g., cpu, memory). Must be constant over related messages.

        operationId:

          type: string

          description: Hash that identifies a measurement (i.e., specification, recipt, result).

        when:

          type: string

          description: Format <start> ... <stop> / <period>

        ts:

          type: string

          format: date-time

          description: Timestamp of the message creation time.

        target:

          type: string

          description: Identifies agent or group of agents.

        resultColumns:

          description: List of possible result (e.g., cpu usage, available memory).

          type: array

          items:

            type: string

**Figure 4.15** : description ***OpenAPI*** des propriétés communes entre les différents topics

Pour les parties propres à chaque *topic*, chaque *topic* contient une énumération à son propre nom de valeur « MESURE ». Il fait référence à la parties communes (décrit ci-dessus) avec « ***$ref*** » et exige par « ***required*** » que chaque *topic* ait son nom dans sa description.

    Capability:

      description: A capability provided by the agent.

      allOf:

        - $ref: '#/components/schemas/CommonFields'

        - type: object

          required:

            - capability

          properties:

            capability:

              description: Verb of the capability message.

              type: string

              enum:

                - MEASURE

    Specification:

      description: A request that corresponds to the capability.

      allOf:

        - $ref: '#/components/schemas/CommonFields'

        - type: object

          required:

            - specification

          properties:

            specification:

              description: Verb of the specification message.

              type: string

              enum:

                - MEASURE

    Interrupt:

      description: Stop an ative specification.

      allOf:

        - $ref: '#/components/schemas/CommonFields'

        - type: object

          required:

            - interrupt

          properties:

            interrupt:

              description: Verb of the interrupt message.

              type: string

              enum:

                - MEASURE

**Figure 4.16** : description ***OpenAPI*** des propriétés non communes entre les différents topics

Pour le *topic* d’acquittement « ***Receipt*** », il doit indiquer le nom ainsi que le type d’erreur rencontré par le serveur (agent). S’il y en a aux moins une ou « empty » pour indiquer qu’il n’y a pas eu d’erreur car c’est un tableau. Ce tableau nous permet ainsi d’informer le client comme les APIs http.

    Receipt:

      description: Response to either a specification or an interrupt.

      allOf:

        - $ref: '#/components/schemas/CommonFields'

        - type: object

          required:

            - receipt

            - errors

          properties:

            receipt:

              description: Verb of the receipt message.

              type: string

              enum:

                - MEASURE

            errors:

              description: List of errors if any. Empty otherwise.

              type: array

              items:

                type: string

    Result:

      description: Result values as measured by the agent.

      allOf:

        - $ref: '#/components/schemas/CommonFields'

        - type: object

          required:

            - result

            - resultValues

          properties:

            result:

              description: Verb of the result message.

              type: string

              enum:

                - MEASURE

            resultValues:

              description: Actual values associated to results measured.

              type: array

              items:

                type: array

                items:

                  type: string

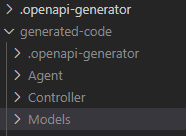
**Figure 4.17** : description ***OpenAPI*** des propriétés de ***Result*** et particularité de ***Receipt***

1. **La génération du code** :

Comme indiqué dans la partie « cas d’utilisation » de ce chapitre, pour la génération de code certaines commandes doivent être exécutée. Elles sont aux nombres de deux qui sont :

* **« mvn clean package -DskipTests** » permet d’installer le Maven sur la machine, supprime l’ancienne compilation java et fait une nouvelle.
* « **java -jar .\target\pubsub-1.0-SNAPSHOT.jar generate -g pubsub-java -i pubsub-java.yaml -o generated-code** » permet la génération du code à partir du fichier YAML indiqué après « -i » et met le code généré dans le dossier indiqué après « -o »

1. **Code généré** : le code généré après l’exécution des commandes se trouve dans le dossier « **generated-code »**, qui lui contient les projets Agent, Controller et Models.

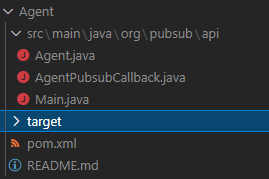


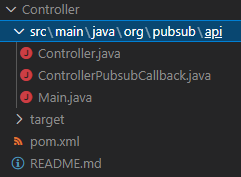
**Figure 4.18** : architecture du dossier contenant le code généré

1. **Agent et Controller** : ces projets contiennent deux (2) classes java, une classe ***main*** et un fichier **POM** chacun. Ces deux classes offrent les différentes fonctions de publication et abonnement aux différents topics aux deux entités **Agent** et **Controller**.

Les classes ***main*** permettent de lancer ces entités (**Agent** et **Controller)**.

Le fichier **POM** décrit ces projets à la manière de Maven, charge les *dependencies* (voir figure 4.21) et le projet Model décrit dessous.

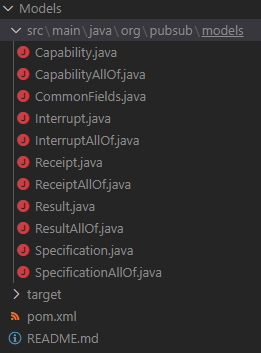




**Figure 4.19** : architecture projet **Agent** et **Controller**

1. **Model** : ce projet contient trois (3) types de fichiers java et un fichier racine **POM**.

* **CommonFields** : cette classe contient les propriétés communes comme indiqué dans la spécification (voir figure 4.13).
* Les fichiers comme « CapabilityAllOf », « InterrupAllOf », etc. contiennent les propriétés exigées et les valeurs énumérés de chaque « *topic* » (voir figure 4.16 et 4.17).
* Les fichiers du nom des *topics* comme « Capability », « Interrupt » etc. décrit les *topics* (voir figure 4.13 et 4.14).
* Le fichier POM contient la description du projet ***Models*** et les ***dependencies*** (voir figure 4.22).



**Figure 4.20** : architecture du projet ***Models***

1. **Les *dependencies* utilisées dans les *POM*** :
2. **Agent et Controller** : Dans la POM du projet Agent et Controller, contient les *dependencies* suivantes :

    <dependencies>

      <dependency>

        <groupId>org.pubsub.models</groupId>

        <artifactId>Model</artifactId>

        <version>1.0.0</version>

      </dependency>

      <dependency>

        <groupId>com.rabbitmq</groupId>

        <artifactId>amqp-client</artifactId>

        <version>5.13.0</version>

      </dependency>

      <dependency>

          <groupId>org.slf4j</groupId>

          <artifactId>slf4j-api</artifactId>

          <version>1.7.5</version>

      </dependency>

      <dependency>

          <groupId>org.slf4j</groupId>

          <artifactId>slf4j-log4j12</artifactId>

          <version>1.7.5</version>

      </dependency>

    </dependencies>

dependencies>

**Figure 4.21** : *dependencies* dans le **POM** **Agent** et **Controller**

1. **Models** : les différentes dependencies du projet Models sont les suivantes :

<dependencies>

        <dependency>

            <groupId>com.fasterxml.jackson.core</groupId>

            <artifactId>jackson-core</artifactId>

            <version>2.9.6</version>

        </dependency>

        <dependency>

            <groupId>com.fasterxml.jackson.core</groupId>

            <artifactId>jackson-annotations</artifactId>

            <version>2.9.6</version>

        </dependency>

        <dependency>

            <groupId>com.fasterxml.jackson.core</groupId>

            <artifactId>jackson-databind</artifactId>

            <version>2.9.6</version>

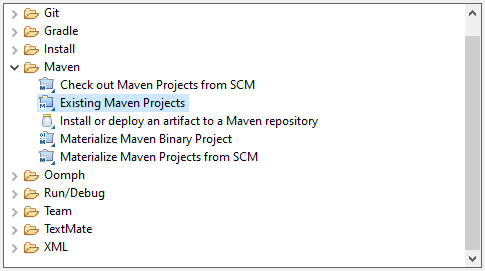
        </dependency>

</dependencies>

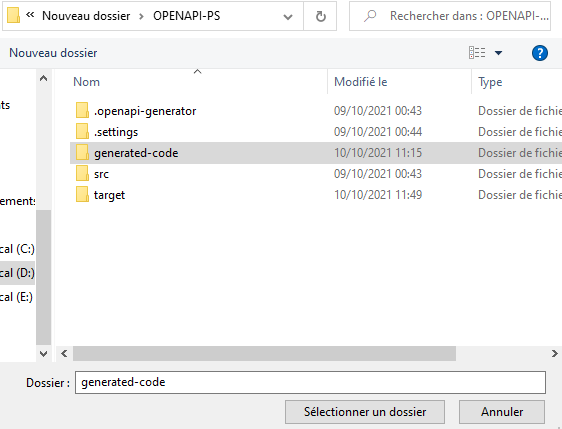
**Figure 4.22** : *dependencies* dans le **POM** du ***Models***

1. **L’exécution du projet** :

Après que nous avons le projet dans l’Eclipse en tant que projet Maven et nous avons pointé vers le dossier ***geneted-code*** pour importer les projets du code générés

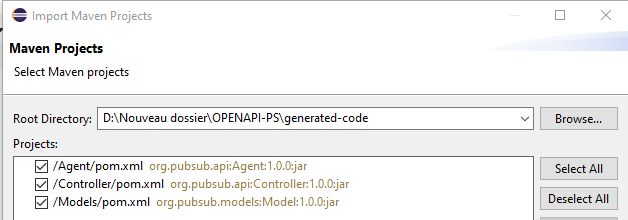


**Figure 4.23** : import d’un projet Maven existant dans Eclipse



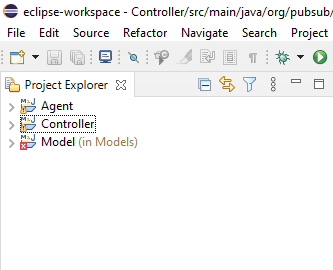
**Figure 4.24** : spécification du dossier contenant les projets

Après la sélection du dossier et avant de compléter l’import, cette figure montre comment chaque projet Maven est définie avec leurs fichiers correspondant :



**Figure 4.25** : définition des projets en fonction de leurs POM

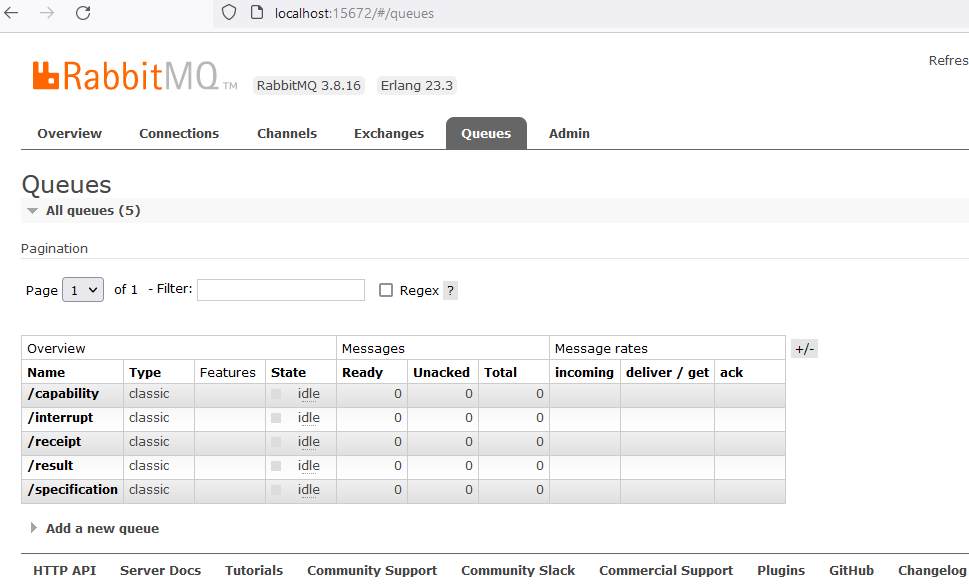
A la fin nous obtenons :



**Figure 4.26** : import des trois (3) projets reçus

Nous avons un petit problème le projet Model à cause des énumérations des valeurs précis par chaque *topic* comme le précise la spécification. Néanmoins ce problème n’empêche pas l’exécution du projet.

Le résultat est visible dans le RabbitMQ en visualisant le port 15672 dans le navigateur :



**Figure 4.27** : le résultat de l’exécution dans le navigateur

1. **Conclusion** :

Le générateur de code *OpenAPI-PS* est un générateur local qui, génère le code local. Plus précisément le code d’implémentation de RabbitMQ, qui est un des brokers locaux de communication publication/abonnement. Cette génération de code est faite à partir d’un programmes informatiques écrits dans un langage de programmation de haut niveau (java). Il prend en compte les règles de modélisation pour produire automatiquement le code source selon les conditions et les règles du développement [39].

La génération automatique de code créée des templates, identifie et mette à jour les dependencies et rendre la programmation plus rapide [40]. Sa mise en place est un travail complexe qui nécessite une programmation dynamique car un petit changement peut induire des erreurs dans le code généré. Mais son utilisation continue d’évoluer au sein des langages de programmation, des IDE. Il permet d’automatiser le développement d’API dans un système pour qu’il corresponde à un autre. Néanmoins un générateur de code doit réduire les templates au strict nécessaire et doit être organiser de manière efficace pour une maintenance facile.

# **Conclusion générale**

Au cours de ce travail, nous avons mis en évidence la compatibilité de Pub/Sub (détaillé dans le chapitre 2) avec les files d’attentes qui peuvent avoir un grand impact sur la communication distribué. Et nous avons aussi comparés une spécification d’architecture client/serveur (*OpenAPI*) à une spécification d’architecture sans serveur (*OpenPS*) utilisant le broker. La spécification *OpenAPI* étant considéré comme légère pour le développement d’API client/serveur, elle n’exige pas la réécriture d’une API existante, elle n’exige pas non plus la liaison d’un logiciel à un service. Car le service décrit peut ne pas appartenir au descripteur et tous les services ne peuvent pas être décrits par *OpenAPI* [41]. Cette spécification n'est pas destinée à couvrir tous les styles d'API web vu l’existence d’autre style de communication comme Pub/Sub et d’autre spécification pour les API [42]. Cependant, la spécification *OpenPS* peut aussi décrire les capacités des services web pour les protocoles de messagerie asynchrone. *OpenPS* utilise un broker gestionnaire des files d’attentes. Ces filles d’attentes permettent une communication asynchrone, ce qui signifie que les points de terminaison qui produisent et consomment les messages interagissent avec la file d'attente et non les uns avec les autres. En séparant les deux entités avec les files d’attentes on aura davantage de flexibilité et une réduction de collision [43].

Bien que *OpenAPI* et son générateur de code soient un outil presque complet et bien qu’il soit le meilleur choix pour les fournisseurs d’API, ses exigences de sécurité comme *api\_key* dans les requêtes et les modèles de chemin URL (Uniform Ressource Locator) ne sont pas compatible avec le Cloud Endpoints. Pour les Cloud Endpoints la précision de port dans la documentation *OpenAPI* ou *OpenPS* et l’écriture de « *parameter* » de type tableau n’est pas non plus accepté [44]. Ces deux spécifications n’acceptent pas des référencements récursifs par exemple un *schema* qui fait référence à lui-même. En considérant que le générateur *OpenAPI* n’est pas tout à fait sécuriser, il ne doit pas être un espoir de générateur de code pour *OpenPS*. Car lors de la génération du code, les fichiers créés (par la fonction File.createTempFile("codegen-", "-tmp")) ne sont pas contrôlés ce qui veut dire que des ajouts ou des suppressions peuvent être faites par un instru [45]. Donc avec le manque d’outil de génération de code pour *OpenPS*, cette spécification n’est pas encore au point mais la description des APIs Pub/Sub avec celle de OpenAPI ne sera pas aussi complet qu’avec *OpenPS*.