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 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 [1] 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 [2].

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 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 [3] qui décrivent l’objectif et celles dans les POM permettent la description du projet généré et les librairies utilisées.

C’est un projet approuvé par des meilleurs développeurs [4].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 [5]. Il décrit les manières d’affichage. Il est implémenté dans plusieurs autres langages de programmation [6]. 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 [5]. 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 [7]. 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 du projet. 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 la balise racine.
2. ***modelVersion*** : qui doit être 4.0.0.
3. ***groupId*** : l’identifiant du groupe du projet.
4. ***artifactId*** : qui implique le nom du projet.
5. ***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 ».
6. **SPI registration**:

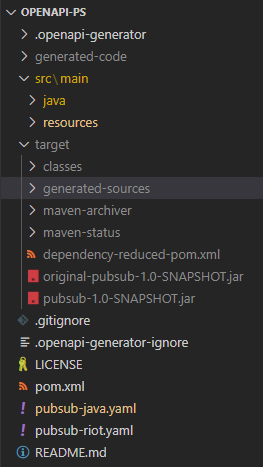
C’est un fichier de configuration OpenAPI codegen approuvé par un ensemble de développeurs sur internet. Il contient le nom des différent fichier *OpenAPI* codegen pour la génération de code en différents langages.

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 4.5** : 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 la l’ensemble des codes des entités, templates et le codegen. Il contient deux dossiers qui sont : java/org/ps et ressource.

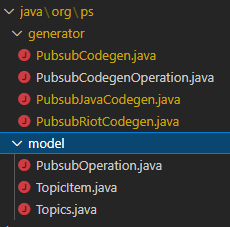
* ***Java/org/ps*** : contient les différents codegen. C’est dans ce fichier que le codegen des langages cible doivent être placés. Ici on a le codegen de java et riot.
* ***Ressource*** : il contient l’ensemble des templates et le POM des entités qui seront générés.

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 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 ***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 dossier « *generator* ». Ces fichiers permettent de vérifier la présence des objets exigés dans la spécification OpenAPI et crée les classes nécessaires en se basant sur les templates.

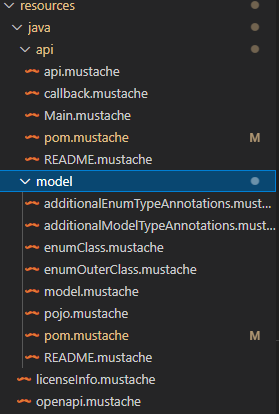
Le fichier model contient les fichier contrôlant les objets racines de OpenAPI.



**Figure 4.6** : 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 4.7** : fichier **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** :
3. **Erlang (OTP)** : Erlang est un langage de programmation utilisé l’évolutivité pour 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 [8]. Il permet le bon fonctionnement de **RabbitMQ** dans notre implémentation.

1. **Visual studio code** : : Un environnement de développement intégré (IDE) extensible, complet et gratuit pour créer des applications modernes [9]. 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 utilisons cet environnement pour tester le code généré par ***OPENAPI-PS***.

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

La spécification OpenAPI utilisé 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 décrie 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 4.8** : 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 4.9** : description 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 abonnement.

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 4.10** : Description ***OpenAPI*** du *topic* de capacité serveur

* **Les autres « *topics* »** : ces « *topics* » sont des opérations qu’un client (controller) peut faire (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 4.11** : description ***OpenAPI*** des autres topics client ou serveur

1. ***Component*** : cette partie décrie les propriétés communes permet de décrire les différents topics ainsi que les propriétés propres à chaque topic. C’est cette partie qui décrie 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 4.12** : description ***OpenAPI*** des propriétés communes entre les différents topics

Pour les parties propres à chaque *topic*, chaque *topic* contient une énumération à son nom de valeur « MESURE ». Il fait référence aux parties communes 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 4.13** : description ***OpenAPI*** des propriétés non communes entre les différents topics

Pour le *topic* d’acquittement « ***Receipt*** », il doit indiquer le nom ainsi te type d’erreur rencontré par le serveur (agent). S’il y en a aux moins une ou « empty » pour indiqur 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

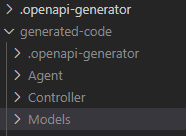
**Figure 4.14** : description ***OpenAPI*** des propriétés de ***Result*** et particularité de ***Receipt***

1. **La génération du code** :

Comme indiqué dans la partie « cas d’utilisation » de ce chapitre, pour la génération 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 faire 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.

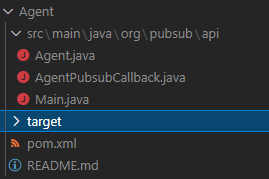


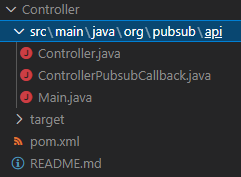
**Figure 4.15** : architecture du dossier contenant le code généré

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 la *dependecy* de RabbitMQ et le projet Model décrit dessous.

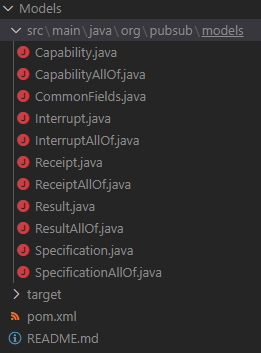




**Figure 4.15** : architecture projet **Agent** et **Controller**

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

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



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>5.13.1</version>

      </dependency>

      <dependency>

          <groupId>org.slf4j</groupId>

          <artifactId>slf4j-api</artifactId>

          <version>1.7.32</version>

      </dependency>

      <dependency>

          <groupId>com.walterjwhite.java.dependencies</groupId>

          <artifactId>slf4j-simple</artifactId>

          <version>0.0.17</version>

          <type>pom</type>

          <scope>test</scope>

      </dependency>

</dependencies>

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>

**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 [10].

La génération automatique de code créée des templates, identifie et mette à jour les dependencies et rendre la programmation plus rapide [11]. 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.

[1] <https://github.com/OpenAPITools/openapi-generator/tree/master/modules/openapi-generator/src/main/java/org/openapitools/codegen>

[2] <https://github.com/OpenAPITools/openapi-generator/tree/master/bin/configs>

[3] <https://swagger.io/tools/swagger-codegen/>

[4] <https://documentation.help/CodeGen/IDH_Topic10.htm>

[5] <https://www.tsmean.com/articles/mustache/the-ultimate-mustache-tutorial/>

[6] <https://mustache.github.io/>

[7] <https://maven.apache.org/guides/introduction/introduction-to-the-pom.html>

[8] <https://www.erlang.org/>

[9] <https://visualstudio.microsoft.com/fr/vs/community/>

[10] <https://www.memoireonline.com/05/12/5885/m_Generation-automatique-du-code-java--partir-d-un-modele-de-classe-UML19.html>

[11] <https://fr.bonitasoft.com/actualites/la-generation-automatique-de-code-les-avantages-et-les-inconvenients>