Un générateur de code pour *OpenPS*

1. **Introduction** :

*OpenPS* est une spécification open source pour les APIs qui utilisent le modèle publication/abonnement. Elle n’a pas encore de générateur de code clients spécifique afin de simplifier l’implémentation pour ces APIs. Pour ça nous mettons en place une approche afin de s'assurer que ces APIs répondent aux exigences définies dans leurs spécifications *OpenPS*. Cette approche consiste en un générateur de code automatique pour une plate-forme cible avec un langage de programmation cible. Pour cela nous nous servons du générateur *OpenAPI*. Et avions deux choix, soit utilisé le générateur OpenAPI, soit la personnalisé. Le générateur *OpenAPI* est un générateur open source qui génère le code client et implémentation serveur en utilisant comme entré un fichier de spécification *OpenAPI* écrit en YAML ou JSON. Ce générateur consiste au minimum en quatre (4) ensemble 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 but de ce chapitre, est de générer un code d’implémentation client d’une API locale et ces *dependencies* avec générateur personnalisé de *OpenAPI* tout en respectant les manière de communication publication/abonnement.

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

*OpenAPI* Codegen est un projet open source qui permet de générer automatiquement des bibliothèques clientes d'API, génération de SDK, 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.Pour l’utiliser à ses propres fin, *OpenAPI* Codegen doit être importer localement.

1. **Templates files** : *Mustache*

Les fichiers templates est un ensemble de fichier contenant 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 [4]. Il décrit les manières d’affichage. Il est implémenté dans plusieurs autre langage de programmation [5]. Mustache se sert des accolades pour indiquer les champs. Ces accolades sont à l’origine son nom car en leurs rotant à 90° ils ressemblent à des moustaches. 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 4.1** : 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 [4]. 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 4.2** : condition (if\_else) en *Mustache*

1. **Une boucle** :

Une boucle en 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 dépends 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 4.3** : 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 [6]. 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**:

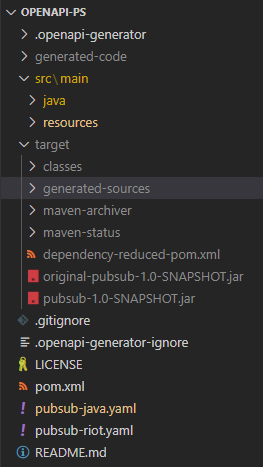
C’est un fichier de configuration OpenAPI codegen approuvé par un ensemble de développeurs sur internet. Il contient le nom des différent fichier *OpenAPI* codegen pour la génération de code en différent langage.

1. **Config file** :

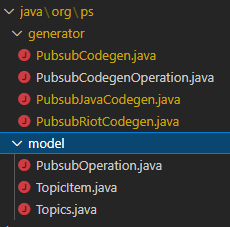
Ce sont fichiers YAML de OpenAPI contenant des exemples réels décrits. Ce qui permet aux utilisateurs de testes la sortie de leur code.

1. **Architecture du générateur OpenPS** :

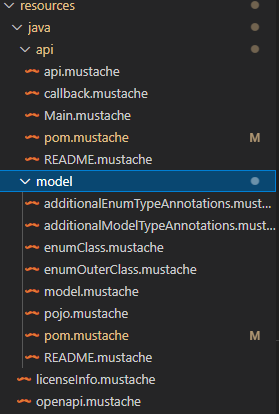
OpenPS est une spécification open source qui n’a pas encore de générateur de code agréé. Pour générer les codes clients pour les APIs Pub/Sub, nous utilisons le générateur de la spécification *OpenAPI* afin de décrire une spécification *OpenPS* qui sera acceptée par le générateur *OpenAPI* localement. 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.



1. **Codegen** :



1. **Templates et POM** :



1. **Un cas d’utilisation** :

Nous avons mis en place une spécification *OpenPS* acceptée le générateur OpenAPI. 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 la communication 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.

1. **YML utilisé en entrée** :

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.

servers:

  - url: http://multiverse.com:4534

    description: Main AMQP broker.

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'

  '/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'

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

    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

    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

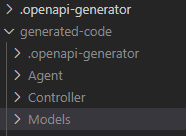
              items:

                type: array

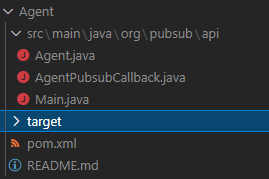
                items:

                  type: string

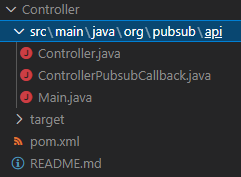
1. **Code généré** :



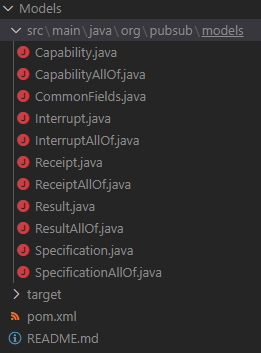
1. **Agent** :



1. **Controller** :



1. **Model** :



1. **Conclusion** :

Le générateur de code *OpenAPI-PS* est un générateur le local qui générant le code localhost. Plus précisément le code d’implémentation de RabbitMQ, qui est un broker local de communication publication/abonnement.

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

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

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

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

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

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