Sorbonne Université

Faculté des Sciences et Ingénierie

Parcours Science et Technologie du Logiciel
(STL)

Rapport de projet : Benchmarking de solutions optimistes pour génération de données test à partir de JSON-Schema

Auteurs

Abdelkader Boumessaoud
Zaky Abdellaoui

Encadrant

Mohamed-Amine Baazizi
Lyes Attouche

Table des matières

1 Rapport mi-parcours

strucutre

1. la description du contexte du projet
2. la présentation du cahier des charges (tâches à réaliser)
3. l'explication des tâches déjà réalisées
4. un prévisionnel des tâches restantes, avec retro-planning associé

1.1 contexte du projet

— objectif du projet : caracteriser les limitations des générateurs open-source en terme de classes de schémas traitées correctement et de classe de schémas problématique
— expliquer contexte : JSON Schema et
— probleme de génération d'exemples : pourquoi c'est compliqué
— citer approaches existants et outils en ligne
— dire pouquoi se limiter au Faker et JE

1.2 cahier des charges

1.3 tâches déjà réalisées

— étude et compréhension : syntaxe et sémantique du langage
— prise en main d'un validateur : experimentation sur Test Suite officiel [**?**], familarisation avec sortie standard
— prise en main de générateurs open-source [**?**] et [**?**] : expériementations sur collection X (xx schémas) et Y (yy schémas)
— implantation d'une chaine de traitement : générateur + validateur avec sortie détaillée
— analyse du code guidée par les exemples : une fois une class de schéma provoquant des erreurs identifiée, synthétiser des schémas ayant les mêmes caracteristiqus (opéretueurs, profondeur, etc) et tester sur ces schémas pour confirmer l'hypothese sur une eventuelle limitation par rapport à une class de schémas
— expérimentation à petite échelle : prendre WP ou SP, classer par famille d'approches, extraire les erreurs au format standard, examiner le premier niveau seulement puis tenter de généraliser. Autre possibilité, examiner pour chauqe output les erreurs de façon complete puis corréler au code source pour confirmer certaines hypotheses sur la limitation de l'outil pour certains opérateurs ou agencement d'opérateurs

1.4 prévisionnel des tâches restantes

— experimentations à large échelle : considérer les collections de schémas de plus grande taille, confirmer/infirmer les hypotheses dérivées sur les collections de plus petite taille

2 Questions

- demander si on peut checker le travail fait sur la 3eme lib - discuter de notre plan d'organisation entre le faker et le everything

3 Introduction

3.1 Contexte et Motivation

JSON-Schema continue d'être un outil précieux pour définir et valider la structure des données JSON dans une variété de contextes. Son expressivité et son format standardisé en font un choix idéal pour décrire les API, vérifier la structure des données dans les pipelines d'apprentissage et spécifier les exigences du développement logiciel. L'utilisation de JSON-Schema dans ces domaines a augmenté ces dernières années, ce qui rend important de disposer d'outils efficaces pour générer des instances de test à partir de ces schémas.

De plus, l'essor de l'architecture des microservices et le besoin d'interopérabilité des données ont augmenté la demande de JSON-Schema, car il permet la définition de structures de données partagées entre plusieurs services. Son utilisation a également été étendue à d'autres applications telles que la validation de données dans des bases de données et des systèmes de stockage de données.

Dans l'ensemble, l'utilité de JSON-Schema découle de sa capacité à fournir un moyen standardisé et expressif de définir et de valider la structure des données JSON, ce qui en fait un outil indispensable dans de nombreux domaines du développement logiciel et de la gestion des données.

Mais malgré cela, ses limites sont a dénombrer, que ce soit du point de vue de complexité, manque de normalisation, types de données limités, performance, Absence de transformation des données, rapport d'erreur limité, Spécification incomplète.

3.2 But et Objectives

L'objectif du projet est d'étudier les limitations (théoriques et pratiques) de différentes implémentations visant à générer des instances de test à partir de schémas JSON, en se concentrant principalement sur trois bibliothèques open-source. L'étude utilisera à la fois la retro-ingénierie et l'analyse expérimentale pour évaluer l'efficacité et les limites de ces approches, et pourra s'étendre à d'autres bibliothèques open source jugées applicables.

3.3 Étendue du projet

L'etude de ses limites permettrai de les surmonter, ce qui mènera a l'élargissement de son domaine applicatif, Cela aiderait à garantir la qualité des instances de test générées à partir des schémas d'entrée, conduisant à des résultats plus fiables et plus précis. La possibilité de générer automatiquement des instances de test à partir de schémas améliorera l'efficacité des processus de développement logiciel, facilitant ainsi le test et la validation de nouvelles applications et de nouveaux systèmes.

4 L'état de l'art

4.1 Vue d'ensemble de JSON-Schema

JSON-Schema est un format de validation pour les données JSON. Il définit la structure attendue d'un document JSON en spécifiant les types de données pour chaque propriété, ainsi que les règles de validation telles que les valeurs minimales et maximales autorisées et les attributs obligatoires. Les schémas JSON peuvent être utilisés pour valider les données à la volée, ce qui permet de garantir la qualité et la cohérence des données et peuvent également être partagés entre différentes applications pour faciliter la communication de données structurées.

4.1.1 Exemple 1 : Générale

Voici un exemple pour illustrer la structure de base d'un JSON-schema :

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Product",
  "description": "A product from Acme's catalog",
  "type": "object",
  "properties": {
    "productId": {
      "description": "The unique identifier for a product",
      "type": "integer",
      "minimum": 1
    },
    "productName": {
      "description": "Name of the product",
      "type": "string",
      "minLength": 1
    },
    "price": {
      "description": "The price of the product",
      "type": "number",
      "exclusiveMinimum": 0
    },
    "tags": {
      "description": "Tags associated with the product",
      "type": "array",
      "items": {
        "type": "string"
      },
      "minItems": 1,
      "uniqueItems": true
    },
    "dimensions": {
      "description": "The dimensions of the product",
      "type": "object",
      "properties": {
        "length": {
          "type": "number"
        },
        "width": {
          "type": "number"
        },
        "height": {
          "type": "number"
        }
      },
      "required": ["length", "width", "height"]
    }
  },
  "required": ["productId", "productName", "price"]
}
```

Ce schéma décrit un objet JSON "Product" qui représente un produit dans le catalogue d'Acme. Il a les propriétés suivantes :

- "productId" (obligatoire) est un entier unique qui représente l'identifiant du produit, avec une valeur minimale de 1.
- "productName" (obligatoire) est une chaîne de caractères qui représente le nom du produit, avec une longueur minimale de 1 caractère.
- "price" (obligatoire) est un nombre qui représente le prix du produit, avec une valeur minimale ex-

clusive de 0.
- "tags" est un tableau de chaînes de caractères qui représentent les tags associés au produit, avec au moins 1 élément et des éléments uniques.
- "dimensions" est un objet qui représente les dimensions du produit, avec les propriétés "length", "width" et "height", toutes obligatoires et de type nombre.

De plus, les propriétés "productId", "productName" et "price" sont requises pour que l'objet soit considéré comme valide.
Un document JSON validé par ce schéma serait :

```json
{
    "productId": 42,
    "productName": "Super Product",
    "price": 19.99,
    "tags": ["awesome", "must-have"],
    "dimensions":
    {
        "length": 10,
        "width": 5,
        "height": 2
    }
}
```

Un document JSON non validé ce schéma serait :

```json
{
    "productId": 0,
    "productName": "",
    "price": -1,
    "tags": [],
    "dimensions": {}
}
```

Ce document ne serait pas validé car :

- "productId" doit être un entier avec une valeur minimale de 1, mais il est défini sur 0.
- "productName" doit être une chaîne non vide, mais c'est une chaîne vide.
- "price" doit être un nombre non négatif, mais il est défini sur -1.
- "tags" doit avoir au moins 1 élément et tous les éléments doivent être uniques, mais c'est un tableau vide.
- "dimensions" doit avoir les propriétés "longueur", "largeur" et "hauteur", mais c'est un objet vide.

Le schéma requiert que toutes les propriétés "productId", "productName" et "price" soient présentes, mais ce document ne les a pas toutes.

4.1.2 Exemple 2 : Objet

Ce schéma JSON décrit les propriétés et les règles pour un objet de produit :

```json
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Objet de produit",
  "description": "Un sch ma pour d crire un objet de produit",
  "type": "object",
  "properties": {
    "id": {
      "type": "integer",
      "description": "Identifiant unique du produit",
      "minimum": 1
    },
```

```
    "nom": {
      "type": "string",
      "description": "Nom du produit"
    },
    "prix": {
      "type": "number",
      "description": "Prix du produit",
      "minimum": 0
    },
    "disponible": {
      "type": "boolean",
      "description": "Indique si le produit est disponible"
    }
  },
  "patternProperties": {
    "^categories.*$": {
      "type": "string",
      "description": "Cat gories du produit"
    }
  },
  "additionalProperties": true,
  "required": [
    "id",
    "nom",
    "prix"
  ],
  "propertyNames": {
    "pattern": "^[a-zA-Z0-9]+$",
    "description": "Les noms de propri t  ne doivent contenir que des
        caract res alphanum riques"
  }
}
```

- patternProperties définit un modèle pour les propriétés qui doivent correspondre au modèle ĉategories.*$. - additionalProperties est défini sur false, ce qui signifie qu'aucune propriété supplémentaire ne peut être ajoutée à l'objet de produit. - unevaluatedProperties définit un autre schéma pour les propriétés qui ne sont pas incluses dans les propriétés requises. - required définit les propriétés requises pour un objet de produit. - propertyNames définit un modèle pour les noms de propriété.

Un document JSON validé par ce schéma serait :

```

{
  "id": 1,
  "nom": "Ordinateur portable",
  "prix": 500,
  "disponible": true,
  "categories": "informatique, ordinateur",
  "unevaluatedProperties": {
    "note": "Bonne performance pour le prix"
  }
}
```

Ce document

Un document JSON non validé par ce schéma serait :

```
{
  "id": -1,
  "nom": "Ordinateur portable",
  "prix": -500,
  "disponible": true,
```

```
  "categories": ["informatique", "ordinateur"],
  "unevaluatedProperties": {
    "note": 123
  },
  "taille": 200
}
```

Ce document n'est pas valide pour les raisons suivantes :

- La propriété id a une valeur négative, ce qui est en dehors de la plage minimale définie dans le schéma. - La propriété prix a une valeur négative, ce qui est en dehors de la plage minimale définie dans le schéma. - La propriété unevaluatedProperties.note n'est pas une chaîne de caractères, ce qui est en dehors des types définis dans le schéma. - La propriété taille est mal orthographiée dans le document JSON, ce qui signifie qu'elle n'est pas définie dans le schéma et que additionalProperties est défini sur false.

4.1.3 Exemple 3 : Array

Voici un schéma JSON qui définit un tableau représentant des produits :

```
{
    "$schema": "http://json-schema.org/draft-07/schema#",
    "type": "array",
    "items": [
        {
            "type": "string",
            "minLength": 3,
            "maxLength": 10
        },
        {
            "type": "integer",
            "minimum": 0
        }
    ],
    "tuple": [
        {
            "type": "string"
        },
        {
            "type": "integer",
            "minimum": 10
        }
    ],
    "contains": {
        "type": "string",
        "enum": ["apple", "banana", "cherry"]
    },
    "minItems": 2,
    "uniqueItems": true
}
```

Ce schéma définit un tableau qui doit avoir au moins 2 éléments et dont les éléments sont uniques. Les éléments du tableau doivent être soit une chaîne avec une longueur minimale de 3 et une longueur maximale de 10, soit un nombre entier supérieur ou égal à 0. Le tableau doit également contenir une séquence de deux éléments, une chaîne et un nombre entier supérieur ou égal à 10. Enfin, le tableau doit contenir au moins un élément parmi une liste d'options prédéfinies (pomme, banane, cerise).

- Items : Il s'agit d'une propriété qui définit les critères de validation pour les éléments d'un tableau. Il peut s'agir d'un schéma JSON complet ou simplement d'un type de données (comme "integer" ou "string").

- Tuple validation : La propriété "items" peut également être définie comme une liste de schémas JSON, permettant une validation tuple pour le tableau, ce qui signifie que chaque élément du tableau doit respecter le schéma défini à la même position dans la liste.

- Unevaluated items : La propriété "additionalItems" peut être utilisée pour définir comment les éléments supplémentaires dans le tableau doivent être traités si le nombre d'éléments dépasse la longueur de la liste de schémas définis dans "items". Si "additionalItems" est défini comme "false", les éléments supplémentaires seront considérés comme non valides. Si "additionalItems" est défini comme un schéma JSON, les éléments supplémentaires seront validés contre ce schéma.

- Contains : La propriété "contains" peut être utilisée pour définir un schéma JSON qui doit être inclus dans au moins un élément du tableau.

- Length : Les propriétés "minItems" et "maxItems" peuvent être utilisées pour définir une plage minimale et maximale de nombres d'éléments pour le tableau.

- Uniqueness : La propriété "uniqueItems" peut être définie comme "true" pour exiger que tous les éléments du tableau soient uniques.

Un document JSON validé par ce schéma serait :

```
[
    "apple", 10,
    "banana", 20
  ]
```

Un document JSON non validé par ce schéma serait :

```
[
  "est",
  5979550
]
```

Ce document (génère par le faker) ne serait pas valide car : ne respecte pas le contains

4.1.4 Exemple 4 : Composition

Voici un schéma JSON qui utilise la composition de schémas :

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Personne",
  "type": "object",
  "properties": {
    "nom": {
      "type": "string",
      "minLength": 3
    },
    "age": {
      "type": "integer",
      "minimum": 0
    },
    "sexe": {
      "type": "string",
      "enum": ["homme", "femme"]
    }
  },
  "required": ["nom", "age", "sexe"],
  "allOf": [
    {
      "not": {
        "properties": {
          "age": {
```

```
              "maximum": 150
            }
          }
        }
      },
      {
        "anyOf": [
          {
            "properties": {
              "sexe": {
                "const": "homme"
              }
            }
          },
          {
            "properties": {
              "age": {
                "minimum": 18
              }
            }
          }
        ]
      },
      {
        "oneOf": [
          {
            "properties": {
              "nom": {
                "pattern": "^[A-Z][a-z]+$"
              }
            }
          },
          {
            "properties": {
              "nom": {
                "pattern": "^[A-Z][a-z]+ [A-Z][a-z]+$"
              }
            }
          }
        ]
      }
    ]
}
```

Ce schéma définit un objet Personne qui a des propriétés obligatoires nom, age et sexe.

L'opérateur allOf spécifie que toutes les conditions doivent être respectées. La première condition utilise l'opérateur not pour spécifier que l'âge ne doit pas dépasser 150 ans. La deuxième condition utilise anyOf pour spécifier que le sexe doit être "homme" ou que l'âge doit être d'au moins 18 ans. La troisième condition utilise oneOf pour spécifier que le nom doit être soit en format "Prénom Nom" soit en format "Nom".

Un document JSON validé par ce schéma serait :

```

{
  "nom": "John Doe",
  "age": 170,
  "sexe": "homme"
}
```

Cet objet a un nom qui est soit "John Doe" en suivant le pattern

La génération d'instances de tests à partir de JSON-Schema consiste à créer des exemples de données qui respectent la structure définie par le schéma JSON. Cela peut être utile pour les tests automatisés, la validation des entrées utilisateur, etc. Les outils de génération d'instances de tests peuvent utiliser les contraintes définies dans le schéma, telles que les types de données, les tailles minimales et maximales, les valeurs minimales et maximales, les éléments uniques, etc., pour générer des données qui respectent ces contraintes.
Les techniques existent pour générer automatiquement des instances de tests à partir de schémas JSON ne garantissent pas toujours la conformité. Une approche optimiste est de combiner des fragments de schémas pour produire une instance valide avec l'utilisation d'un validateur externe pour corriger les instances non conformes. Son efficacité est dans l'absence de la complexité difficile du problème de génération, qui a une complexité exponentielle prouvée.

4.3 Limitations des approches existantes (non-optimiste)

Il existe plusieurs approches pour générer des instances de test à partir de JSON-Schema :

- Génération aléatoire : en utilisant des bibliothèques telles que jsonschema2popo, on peut générer des instances aléatoires en se basant sur le JSON-Schema, en respectant les contraintes définies dans le schéma.
-Génération basée sur des exemples : en utilisant des bibliothèques telles que json-schema-faker, on peut générer des instances en se basant sur des exemples spécifiés dans le JSON-Schema.
-Génération à partir de modèles : en utilisant des outils tels que QuickType, on peut générer du code pour des instances de test en fonction du JSON-Schema, ce qui peut être utile pour les tests automatisés.

4.3.1 limitations théoriques

4.3.2 limitations pratiques

5 Méthodologie

5.1 json-everything

5.2 json-schema-faker

5.2.1 quelques tests

les instances générés par json-schema-faker (sur interface) seront testé sur jsonschemavalidator

en prenant l'exemple 1 : objet

6 Résultats

6.1 Trouvailles

6.2 Comparaison des bibliothèques

6.3 Limites des approches optimistes

6.3.1 limitations théoriques

6.3.2 limitations pratiques

7 Conclusions

7.1 Résumé des résultats

7.2 Implications

7.3 Travail futur

Références

[1] https ://json-schema.org

[2] Jsongenerator. https ://github.com/jimblackler/jsongenerator

[3] Json schema faker. https ://github.com/json-schema-faker/json-schema-fakeryes

[4] JSON everything. https ://github.com/gregsdennis/json-everything

[5] Lyes Attouche, Mohamed Amine Baazizi, Dario Colazzo, Francesco Falleni, Giorgio Ghelli, Cristiano Landi, Carlo Sartiani, Stefanie Scherzinger : A Tool for JSON Schema Witness Generation. EDBT 2021 : 694-697

[6] Mohamed Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, Stefanie Scherzinger : Not Elimination and Witness Generation for JSON Schema. CoRR abs/2104.14828 (2021)

[7] Pierre Bourhis, Juan L. Reutter, Fernando Suárez, Domagoj Vrgoc