Sorbonne Université

Faculté des Sciences et Ingénierie

Master Informatique
Parcours Science et Technologie du Logiciel
(STL)

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

Auteurs

Abdelkader Boumessaoud
Zaky Abdellaoui

Encadrants

Mohamed-Amine Baazizi
Lyes Attouche

Table des matières

1 Contexte du projet

1.1 Objectifs du projet

L'objectif principal de ce projet est de caractériser les limitations des générateurs open-source en termes de classes de schémas JSON traitées correctement et de classes de schémas problématiques. Pour atteindre cet objectif, nous utiliserons des méthodes de rétro-ingénierie et des analyses expérimentales pour évaluer l'efficacité et les limites de trois bibliothèques open-source spécifiques, mais nous pourrons également étendre notre étude à d'autres bibliothèques si nécessaire. L'analyse des résultats nous permettra de caractériser les classes de schémas qui posent des problèmes aux générateurs open-source et d'identifier les domaines dans lesquels ces générateurs peuvent être améliorés.

1.2 Contexte

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.

Malgré cela, ses limites sont nombreuses, que ce soit du point de vue de sa complexité, de son manque de normalisation, des types de données qu'il autorise, de sa performance, son absence de transformation des données,sa gestion limité des erreurs, ou encore sa spécification incomplète.

1.3 Problèmes de génération

La génération d'instances de schémas JSON peut poser problème car il est souvent difficile de s'assurer que les instances que l'on génère respectent réellement les contraintes spécifiées. De surcroît, si le schéma est complexe, le processus peut devenir coûteux en temps et en ressources, en particulier avec une entrée de grande taille. Il est également possible que certaines parties du schéma soient ambiguës ou mal définies, ce qui complique la recherche et la création d'instances satisfaisantes.

1.4 Approches existantes

Nous nous sommes concentrés sur trois bibliothèques, toutes publiées en tant que logiciel libre, déjà identifiées et choisies pour leur compatibilité avec la quasi-totalité des opérateurs de JSON Schema :

JSON Schema Faker Une bibliothèque JavaScript de génération de données fictives pour les documents JSON, qui utilise des générateurs de données aléatoires. Elle se base sur la spécification JSON Schema pour définir le contenu autorisé d'un document JSON, et permet de générer des données basiques ou complexes conformes au schéma donné en entrée.

`json-everything` Une série de projets visant à étendre la fonctionnalité de traitement des données JSON dans l'espace `System.Text.Json` en C#, qui propose une bibliothèque (elle même en C#) pour interroger et gérer les données JSON. Elle est basée sur un générateur de données pour les classes C# et est donc limitée en termes d'expressivité sur la partie de JSON Schema prise en charge.

JSON Generator : Une bibliothèque Java pour la génération de données JSON à partir de schémas JSON. Elle prend en charge une variété de types de données JSON, notamment les

chaînes de caractères, les nombres, les booléens, les tableaux, les objets et les valeurs nulles. Elle permet également la génération des données aléatoires à partir des schémas JSON, ce qui facilite la création de jeux de données de test. JSON Generator prend en charge la norme JSON Schema Draft 7, mais peut être étendue pour prendre en charge d'autres normes de schéma JSON.

1.5 Le choix de JSON Schema Faker et de `json-everything`

En effectuant des tests, nous avons pu observer que le taux de validation des instances générées par JSON Schema Faker et `json-everything` était inférieur à celui de JSON Generator. Nnous n'avons donc pas trouvé pertinent de se concentrer sur ce dernier pour notre étude comparative, et avons choisi de nous concentrer sur les deux autres, pour mener une analyse plus approfondie de leurs fonctionnalités et performances.

2 Cahier des charges

2.1 Objectif

Caractériser les limitations des générateurs open-source en caractérisant des classes de schémas JSON traitées correctement et de classes de schémas problématiques, et identifier les manières dont ces générateurs pourraient être améliorés.

2.2 Méthodes

— Utiliser des méthodes de rétro-ingénierie et des analyses expérimentales pour évaluer lefficacité et les limites des bibliothèques open-source spécifiées, mais nous pourrons également étendre notre étude à dautres bibliothèques si nécessaire.

— Caractériser les classes de schémas qui posent des problèmes aux générateurs open-source.

— Identifier les domaines dans lesquels ces générateurs peuvent être améliorés.

2.3 Livrables

— Un rapport détaillé présentant les résultats de l'analyse, les conclusions et les recommandations.

— Les codes sources utilisés pour l'analyse et les résultats obtenus.

— Des exemples de schémas JSON problématiques et des solutions proposées pour les traiter.

2.4 Contraintes

— Les analyses doivent être effectuées sur des schémas JSON de différentes tailles et complexités.

— Les résultats doivent être fiables et reproductibles.

— Les codes sources et les résultats doivent être documentés et organisés de manière à faciliter la compréhension et la réutilisation.

3 Tâches déjà réalisées

3.1 Étude et compréhension de JSON Schema

L'étude et la compréhension de JSON Schema ont été des étapes essentielles de notre projet. JSON Schema est une spécification qui permet de définir la structure, le type et les contraintes de validation pour des documents JSON. Elle fournit un ensemble de mots-clés pour décrire la forme attendue des données, comme le type, le format, les propriétés requises et facultatives, les valeurs par défaut, etc.

Nous avons étudié en détail les différents mots-clés disponibles dans JSON Schema, ainsi que leur syntaxe et leur sémantique. Nous avons également examiné des exemples concrets de schémas JSON pour mieux comprendre leur utilisation pratique.

Nous avons appris que JSON Schema est un outil puissant pour valider et documenter la structure des documents JSON. Grâce à JSON Schema, il est possible de garantir la qualité des données échangées entre différents systèmes, en s'assurant que les documents JSON respectent des contraintes spécifiques. Nous avons également constaté que JSON Schema peut être utilisé pour générer des données de test conformes à un schéma donné, ce qui peut être très utile pour tester des applications qui manipulent des données JSON.

En conclusion, notre étude approfondie de JSON Schema nous a permis de mieux comprendre les fonctionnalités et les avantages de cette spécification, ainsi que son importance pour notre projet.

3.2 Prise en main d'un validateur JSON Schema

Nous avons examiné plusieurs options disponibles pour les validateurs JSON Schema et avons finalement choisi jschon (une bibliothèque Python) pour nos besoins.

Nous avons d'abord étudié la documentation et les exemples fournis par le validateur pour comprendre son utilisation et ses fonctionnalités. Ensuite, nous avons mis en place un environnement de test pour valider nos schémas JSON à l'aide du validateur.

Nous avons également exploré les fonctionnalités avancées du validateur, telles que les extensions et les validateurs personnalisés, pour répondre à des besoins spécifiques. Nous avons finalement intégré le validateur à notre pipeline de validation pour garantir que tous nos schémas JSON soient valides avant de les utiliser dans notre application.

3.3 Prise en main de générateurs open-source

Nous avons réalisé une étude approfondie de deux générateurs open-source de données JSON : JSON Schema Faker et `json-everything` . Nous avons pris en main ces outils pour comprendre leur fonctionnement et leur syntaxe, puis les avons utilisés pour générer des données JSON simples et complexes.

Nous avons constaté que ces générateurs étaient capables de générer des données cohérentes et réalistes en respectant les contraintes définies par les schémas JSON. Cependant, nous avons également identifié certaines limites en termes de complexité et de diversité des données générées.

Nous avons notamment remarqué que la génération de données imbriquées ou de structures de données plus complexes pouvait poser des difficultés, et que les options de personnalisation des générateurs étaient parfois limitées.

Malgré ces limitations, nous considérons que l'utilisation de générateurs open-source comme JSON Schema Faker et `json-everything` peut être une solution pratique et efficace pour la génération de données JSON.

3.4 Implantation d'une chaine de traitement

Cette chaîne de traitement est une pipeline qui se compose de trois étapes principales.

La première étape est la génération d'instances JSON Schema pour un ensemble de datasets de JSON Schema. Pour ce faire, nous avons utilisé des générateurs open-source tels que (JSON Schema Faker / `json-everything`).

Une fois que nous avons généré les instances, la deuxième étape consiste à valider ces instances par rapport aux schémas JSON correspondants. Nous avons utilisé un validateur JSON Schema jschon.

Enfin, la troisième étape consiste à classer ces schémas selon l'erreur de validation de leurs instances générées. Cette classification nous permet d'analyser les schémas ayant des erreurs similaires ensemble, ce qui facilite l'analyse du code du générateur. Nous avons utilisé l'IDE WebStorm de la suite JetBrains pour cette étape.

Grâce à cette chaîne de traitement, nous avons pu générer des données de manière automatique et efficace, et nous avons également pu identifier des erreurs dans les schémas JSON générés. Cela nous a permis d'analyser les schémas à problèmes et d'améliorer les générateurs open-source utilisés.

3.5 Collections de schémas utilisés

On s'est permis d'expérimenter sur quatre datasets regroupant 8064 schémas

Snowplow : totalisant 420 schémas.
WashingtonPost : totalisant 125 schémas.
Kubernetes : totalisant 1092 schémas.
GitHub : totalisant 6427 schémas.

3.6 Analyse du code guidée par les exemples

Elle nous a permis de mieux comprendre les générateurs open-source que nous avons utilisés. Nous avons cherché des exemples concrets de schémas JSON pour lesquels ces générateurs produisent des résultats différents, ce qui nous a permis d'identifier les parties du code qui sont responsables de ces différences.

Nous avons également étudié en détail la documentation de chaque générateur, afin de mieux comprendre leur fonctionnement interne et leur architecture logicielle. Nous avons ainsi pu identifier les algorithmes et les structures de données clés utilisés par chaque générateur, ainsi que les méthodes de validation et de génération de données.

Cette approche d'analyse du code guidée par les exemples nous a permis d'obtenir une compréhension approfondie des générateurs open-source que nous avons étudiés, ainsi que de leurs limites et de leurs avantages. Elle nous a également permis de formuler des recommandations pour améliorer la qualité des données générées, en identifiant les parties du code qui pourraient être améliorées ou optimisées.

De plus, une fois une classe de schéma provoquant des erreurs identifiée, nous avons synthétisé des schémas ayant les mêmes caractéristiques (opérateurs, profondeur, etc.) et testé sur ces schémas pour confirmer l'hypothèse sur une éventuelle limitation par rapport à une classe de schéma.

3.7 Expérimentation à petite échelle

L'approche que nous avons utilisée consistait à se concentrer sur certains schémas d'un dataset, tel que WashingtonPost ou SnowPlow, de manière individuelle et dont l'instance générée était invalide, et à analyser l'erreur de validation. Nous avons pu ensuite extrait les erreurs au format standard, en examinant le premier niveau seulement.

Une autre approche consistait à examiner pour chaque output les erreurs de façon complète, puis à corréler ces erreurs au code source pour confirmer certaines hypothèses sur la limitation de l'outil pour certains opérateurs ou agencements d'opérateurs.

Nous avons ensuite analysé en détail les règles de validation du schéma et les propriétés et contraintes définies. Nous avons également examiné le code source du générateur open-source correspondant pour comprendre comment il générait les données et quelles étaient les éventuelles erreurs ou limitations du générateur. Nous avons ainsi pu identifier les parties du code qui étaient responsables des erreurs de validation.

Pour améliorer la génération de données, nous avons proposé des modifications des règles de génération ou des contraintes du schéma. Enfin, nous avons validé notre approche en générant à nouveau des instances pour le même schéma, en utilisant les modifications que nous avions proposées. Nous avons ainsi pu constater une amélioration significative du taux de validation pour ce schéma.

4 Prévision des tâches restantes

4.1 Expérimentations à large échelle

Dans le cadre de l'expérimentation à grande échelle, l'idée principale serait d'utiliser les résultats de la classification effectuée par la pipeline pour travailler sur un grand ensemble de schémas, en considérant des collections de schémas de plus grande taille. Nous pourrions ainsi analyser ceux qui ont des erreurs de validation similaires et utiliser cette information pour valider la performance et la fiabilité des générateurs open-source que nous utilisons, tout en confirmant ou infirmant les hypothèses dérivées sur les collections de plus petite taille.

Cette expérimentation nous permettrait également de mesurer la qualité des schémas et des instances générées par ces générateurs sur un plus grand nombre de schémas, ce qui serait utile pour déterminer leur pertinence dans des projets plus vastes nécessitant une utilisation intensive de schémas JSON et de générateurs de données.

Références

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

[2] JSON Generator https://github.com/jimblackler/jsongenerator

[3] JSON Schema Faker https://github.com/json-schema-faker/json-schema-faker

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

[5] jschon (Validation) https://jschon.dev/

[6] 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

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

[8] Pierre Bourhis, Juan L. Reutter, Fernando Suárez, Domagoj Vrgoc, *JSON : Data model, Query languages and Schema specification*, PODS1