RFC 8927 : JSON Type Definition

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 7 novembre 2020

Date de publication du RFC : Novembre 2020

https://www.bortzmeyer.org/8927.html

Il existe plusieurs langages pour décrire la structure d'un document JSON. Aucun ne fait l'objet d'un clair consensus. Sans compter les nombreux programmeurs qui ne veulent pas entendre parler d'un schéma formel. Ce nouveau RFC décrit un de ces langages, **JTD**, "*JSON Type Definition*". Une de ses particularités est que le schéma est lui-même écrit en JSON. Son cahier des charges est de faciliter la génération automatique de code à partir du schéma. JTD est plus limité que certains langages de schéma, afin de faciliter l'écriture d'outils JTD.

On l'a dit, il n'existe pas d'accord dans le monde JSON en faveur d'un langage de schéma particulier. La culture de ce monde JSON est même souvent opposée au principe d'un schéma. Beaucoup de programmeurs qui utilisent JSON préfèrent l'agilité, au sens « on envoie ce qu'on veut et le client se débrouille pour le comprendre ». Les mêmes désaccords existent à l'IETF, et c'est pour cela que ce RFC n'est pas sur le chemin des normes, mais a juste l'état « Expérimental ».

JSON est normalisé dans le RFC 8259 [1]. D'innombrables fichiers de données sont disponibles au format JSON, et de très nombreuses API prennent du JSON en entrée et en rendent en sortie. La description des structures de ces requêtes et réponses est typiquement faite en langage informel. C'est par exemple le cas de beaucoup de RFC qui normalisent un format utilisant JSON comme le RFC 9083, les RFC 8620 et RFC 8621, le RFC 7033, etc. Une des raisons pour lesquelles il est difficile de remplacer ces descriptions en langue naturelle par un schéma formel (comme on le fait couramment pour XML, par exemple avec Relax NG) est qu'il n'y a pas d'accord sur le cahier des charges du langage de schéma. JTD ("*JSON Type Definition*") a des exigences bien précises (section 1 du RFC). Avant de comparer JTD à ses concurrents (cf. par exemple l'annexe B), il faut bien comprendre ces exigences, qui influent évidemment sur le langage :

1. Pour voir le RFC de numéro NNN, https://www.ietf.org/rfc/rfcNNN.txt, par exemple https://www.ietf.org/rfc/rfc8259.txt

— Description sans ambiguïté de la structure du document JSON (d'accord, cette exigence est assez banale, pour un langage de schéma…),
— Possibilité de décrire toutes les constructions qu'on trouve dans un document JSON typique,
— Une syntaxe qui doit être facile à lire et à écrire, aussi bien pour les humains que pour les programmes,
— Et, comme indiqué plus haut, tout faire pour faciliter la génération de code à partir du schéma, la principale exigence de JTD ; l'idée est que, si un format utilise JTD, un programme qui analyse ce format soit en bonne partie générable uniquement à partir de la description JTD.

Ainsi, JTD a des entiers sur 8, 16 et 32 bits, qu'un générateur de code peut traduire directement en (le RFC utilise des exemples C++) `int8_t`, `int16_t`, etc, mais pas d'entiers de 64 bits, pourtant admis en JSON mais peu portables (cf. annexe A.1). JTD permet de décrire les propriétés d'un objet (≪ objet ≫, en JSON, désigne un dictionnaire) qu'on peut traduire en `struct` ou `std::map` C++.

Les fans de théorie des langages et de langages formels noteront que JTD n'est pas lui-même spécifié en JTD. Le choix ayant été fait d'un format simple, JTD n'a pas le pouvoir de se décrire lui-même et c'est pour cela que la description de JTD est faite en CDDL ("*Concise Data Definition Language*", RFC 8610).

La syntaxe exacte est spécifiée en section 2, une fois que vous avez (re)lu le RFC 8610. Ainsi, la description de base, en CDDL, d'un membre d'un objet JSON est :

```
properties = (with-properties // with-optional-properties)

with-properties = (
     properties: { * tstr => { schema }},
     ? optionalProperties: { * tstr => { schema }},
     ? additionalProperties: bool,
     shared,
   )
```

Ce qui veut dire en langage naturel que le schéma JTD peut avoir un membre `properties`, lui-même ayant des membres composés d'un nom et d'un schéma. Le schéma peut être, entre autres, un type, ce qui est le cas dans l'exemple ci-dessous. Voici un schéma JTD trivial, en JSON comme il se doit :

```
{
  "properties": {
    "name": {
      "type": "string"
    },
    "ok": {
      "type": "boolean",
      "nullable": true
    },
    "level": {
      "type": "int32"
    }
  }
}
```

Ce schéma accepte le document JSON :

```
{
  "name": "Foobar",
  "ok": false,
  "level": 1
}
```

Ou bien ce document :

```
{
  "name": "Durand",
  "ok": null,
  "level": 42
}
```

(L'élément "*nullable*" peut valoir `null` ; si on veut pouvoir omettre complètement un membre, il faut le déclarer dans `optionalProperties`, pas `properties`.) Par contre, cet autre document n'est pas valide :

```
{
  "name": "Zig",
  "ok": true,
  "level": 0,
  "extra": true
}
```

Car il y a un membre de trop, `extra`. Par défaut, JTD ne le permet pas mais un schéma peut comporter `additionalProperties: true` ce qui les autorisera.

Ce document JSON ne sera pas accepté non plus :

```
{
  "name": "Invalid",
  "ok": true
}
```

Car la propriété `level` ne peut pas être absente.

Un exemple plus détaillé, et pour un cas réel, figure dans l'annexe C du RFC, en utilisant le langage du RFC 7071.

JTD ne permet pas de vrai mécanisme d'extension mais on peut toujours ajouter un membre `metadata` dont la valeur est un objet JSON quelconque, et qui sert à définir des « extensions » non portables.

Jouons d'ailleurs un peu avec une mise en œuvre de JTD. Vous en trouverez plusieurs ici <`https://github.com/jsontypedef`>, pour divers langages de programmation. Essayons avec celui en Python. D'abord, installer le paquetage :

```
% git clone https://github.com/jsontypedef/json-typedef-python.git
% cd json-typedef-python
% python setup.py build
% python setup.py install --user
```

(Oui, on aurait pu utiliser `pip install jtd` à la place.) Le paquetage n'est pas livré avec un script exécutable, on en crée un en suivant la documentation <https://jtd.readthedocs.io/>. Il est simple :

```
#!/usr/bin/env python3

import sys
import json

import jtd

if len(sys.argv) != 3:
    raise Exception("Usage: %s schema json-file" % sys.argv[0])

textSchema = open(sys.argv[1], 'r').read()
textJsonData = open(sys.argv[2], 'r').read()

schema = jtd.Schema.from_dict(json.loads(textSchema))
jsonData = json.loads(textJsonData)

result = jtd.validate(schema=schema, instance=jsonData)
print(result)
```

Si le fichier JSON correspond au schéma, il affichera un tableau vide, sinon un tableau contenant la liste des erreurs :

```
% ./jtd.py myschema.json mydoc1.json
[]
```

(`myschema.json` contient le schéma d'exemple plus haut et `mydoc1.json` est le premier exemple JSON.) Si, par contre, le fichier JSON est invalide :

```
 % ./jtd.py myschema.json mydoc3.json
[ValidationError(instance_path=['extra'], schema_path=[])]
```

(`mydoc3.json` était l'exemple avec le membre supplémentaire, `extra`.)

Une particularité de JTD est de normaliser le mécanisme de signalement d'erreurs. Les erreurs doivent être formatées en JSON (évidemment…) avec un membre `instancePath` qui est un pointeur JSON (RFC 6901) indiquant la partie invalide du document, et un membre `schemaPath`, également un pointeur, qui indique la partie du schéma qui invalidait cette partie du document (cf. le message d'erreur ci-dessus, peu convivial mais normalisé).

JTD est spécifié en CDDL donc on peut tester ses schémas avec les outils CDDL, ici un outil en Ruby :

```
% gem install cddl --user
```

Ensuite, on peut valider ses schémas :

```
% cddl jtd.cddl validate myschema.json
%
```

Si le schéma a une erreur (ici, j'ai utilisé le type `char`, qui n'existe pas) :

```
% cddl jtd.cddl validate wrongschema.json
CDDL validation failure (nil for {"properties"=>{"name"=>{"type"=>"char"}, "ok"=>{"type"=>"boolean", "nullable"=
["char", [:text, "timestamp"], nil]
["char", [:text, "timestamp"], null]
```

(Oui, les messages d'erreur de l'outil `cddl` sont horribles.)

Et avec l'exemple de l'annexe C, le reputon du RFC 7071 :

```
% cddl jtd.cddl validate reputon.json
%
```

C'est parfait, le schéma du RFC est correct, validons le fichier JSON tiré de la section 6.4 du RFC 7071 :

```
% ./jtd.py reputon.json r1.json
[]
```

Si jamais il y a une erreur (ici, on a enlevé le membre `rating`) :

```
% ./jtd.py reputon.json r1.json
[ValidationError(instance_path=['reputons', '0'], schema_path=['properties', 'reputons', 'elements', 'properties
```

Une intéressante annexe B fait une comparaison de JTD avec CDDL. Par exemple, le schéma CDDL :

```
root = "PENDING" / "DONE" / "CANCELED"
```

accepterait les mêmes documents que le schéma JTD :

```
{ "enum": ["PENDING", "DONE", "CANCELED"]}
```

Et celui-ci, en CDDL (où le point d'interrogation indique un terme facultatif) :

```
root = { a: bool, b: number, ? c: tstr, ? d: tdate }
```

reviendrait à ce schéma JTD :

```
{
  "properties": {
    "a": {
      "type": "boolean"
    },
    "b": {
      "type": "float32"
    }
  },
  "optionalProperties": {
    "c": {
      "type": "string"
    },
    "d": {
      "type": "timestamp"
    }
  }
}
```

Merci à Ulysse Carion pour sa relecture.