RFC 7159 : The JSON Data Interchange Format

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 3 mars 2014

Date de publication du RFC : Mars 2014

https://www.bortzmeyer.org/7159.html

Il existe une pléthore de langages pour décrire des données structurées <https://www.bortzmeyer.org/data-formats.html>. JSON, normalisé dans ce RFC (qui succède au RFC 4627 [1]), est actuellement le plus à la mode. JSON se veut plus léger que XML. Comme son concurrent XML, c'est un format textuel, et il permet de représenter des structures de données hiérarchiques. Ce RFC est l'ancienne norme JSON, remplacée depuis par le RFC 8259.

À noter que JSON doit son origine, et son nom complet ("*JavaScript Object Notation*") au langage de programmation JavaScript, dont il est un sous-ensemble. La norme officielle de JavaScript est à l'ECMA actuellement la version 5.1 du document ECMA-262 <http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf>. JSON est dans la section 15.12 de ce document mais est aussi dans ECMA-404 <http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf>, qui lui est réservé. Il était prévu une publication commune ECMA/IETF mais elle n'a finalement pas eu lieu. Contrairement à JavaScript, JSON n'est pas un langage de programmation, seulement un langage de description de données, et il ne peut donc pas servir de véhicule pour du code méchant (sauf si on fait des bêtises comme de soumettre du texte JSON à `eval()`, cf. section 12 et erratum #3607 <http://www.rfc-editor.org/errata_search.php?eid=3607> qui donne des détails sur cette vulnérabilité).

Voici un exemple, tiré du RFC, d'un objet exprimé en JSON :

1. Pour voir le RFC de numéro NNN, https://www.ietf.org/rfc/rfcNNN.txt, par exemple https://www.ietf.org/rfc/rfc4627.txt

```
{
    "Image": {
        "Width":  800,
        "Height": 600,
        "Title":  "View from 15th Floor",
        "Thumbnail": {
            "Url":    "http://www.example.com/image/481989943",
            "Height": 125,
            "Width":  "100"
        },
        "IDs": [116, 943, 234, 38793]
      }
 }
```

Les détails de syntaxe sont dans la section 2 du RFC. Cet objet d'exemple a un seul champ, `Image`, qui est un autre objet (entre { et }) et qui a plusieurs champs. (Les objets sont appelés dictionnaires ou "*maps*" dans d'autres langages.) L'ordre des éléments de l'objet n'est pas significatif (certains analyseurs JSON le conservent, d'autres pas). Un de ces champs, `IDs`, a pour valeur un tableau (entre [et]). Les éléments d'un tableau ne sont pas forcément du même type (section 5, qui précise un point qui n'était pas clair dans le précédent RFC). Autrefois, un texte JSON était forcément un objet ou un tableau. Ce n'est plus le cas aujourd'hui donc, par exemple :

```
"Hello world!"
```

est un texte JSON légal (composé d'une chaîne de caractères en tout et pour tout). Ce point a d'ailleurs suscité de vigoureuses protestations car un lecteur de JSON qui lit au fil de l'eau peut ne pas savoir si le texte est fini ou pas (avant, il suffisait de compter les crochets et accolades).

Et quel encodage utiliser pour les textes JSON (section 8) ? Le RFC 4627 était presque muet à ce sujet. Cette question est désormais plus développée. Le jeu de caractères est toujours Unicode et l'encodage par défaut est UTF-8. UTF-16 et UTF-32 sont permis mais UTF-8 est le seul qui assure l'interopérabilité, bien des mises en œuvre de JSON ne peuvent en lire aucun autre. Les textes JSON ne doivent pas utiliser de BOM et le RFC ne dit pas comment le récepteur sait quel est l'encodage utilisé, on suppose qu'il doit être indiqué par ailleurs (par exemple dans l'en-tête `Content-Type:` en HTTP).

Notre RFC ne spécifie pas un comportement particulier à adopter lorsqu'une chaîne de caractères n'est pas légale pour l'encodage choisi. Cela peut varier selon l'implémentation.

Autre problème classique d'Unicode, la comparaison de chaînes de caractères. Ces comparaisons doivent se faire selon les caractères Unicode et pas selon les octets (il y a plusieurs façons de représenter la même chaîne de caractères, par exemple `foo*bar` et `foo\u002Abar` sont la même chaîne).

JSON est donc un format simple, il n'a même pas la possibilité de commentaires <http://stackoverflow.com/questions/244777/can-i-comment-a-json-file> dans le fichier... Voir sur ce sujet une intéressante compilation <http://blog.getify.com/2010/06/json-comments/>.

Le premier RFC décrivant JSON était le RFC 4627. Quels changements apporte cette première révision (section 1.3 et annexe A) ? Au début du travail, l'IETF était partagée entre des ultras voulant réparer tous les problèmes de la spécification et les conservateurs voulant changer le moins de choses possible. Finalement, rien de crucial n'a été modifié, quelques bogues <http://www.rfc-editor.org/errata_search.php?rfc=4627&rec_status=15&presentation=table> ont été corrigées et des points flous du premier RFC ont été précisés (quelques uns ont déjà été présentés plus haut). Cela a quand même posé quelques problèmes philosophiques. Ainsi, le RFC 4627 disait dans sa section 2.2 « "*The names within an object SHOULD be unique*" ». Et s'ils ne l'étaient pas, que se passait-il ? Si un producteur de JSON, ignorant l'avertissement, envoie :

```
{
"name": "Jean Dupont",
"city": "Paris",
"city": "London"
}
```

Que va faire le récepteur ? Planter ? Prendre la première ville ? La dernière ? Un ensemble de toutes les villes ? Une solution possible aurait été de transformer le *"SHOULD"* en *"MUST"* (cf. RFC 2119) et donc de déclarer cet objet JSON illégal. Mais cela aurait rendu illégaux des tas d'objets JSON circulant sur l'Internet et qui n'étaient que déconseillés. Le choix a donc été de décréter (section 4) que l'objet respectant la règle « noms uniques » serait **interopérable** (tous les récepteurs en feront la même interprétation) alors que celui qui viole cette règle (comme l'exemple avec deux `city` plus haut) aurait un comportement imprévisible. Un programme mettant en œuvre l'analyse de JSON a donc le droit de choisir parmi tous les comportements mentionnés plus haut.

Autres changements, un avertissement (section 6) sur la précision des nombres (on peut écrire 3.14159265358979323846 mais on n'a aucune garantie que le récepteur pourra conserver une telle précision).

Voici un exemple d'un programme Python pour écrire un objet Python en JSON (on notera que la syntaxe de Python et celle de JavaScript sont très proches) :

```
import json

objekt = {u'Image': {u'Width': 800,
                     u'Title': u'View from Smith\'s, 15th Floor, "Nice"',
                     u'Thumbnail': {u'Url':
                                    u'http://www.example.com/image/481989943',
                                    u'Width': u'100', u'Height': 125},
                     u'IDs': [116, 943, 234, 38793],
                     u'Height': 600}} # Example from RFC 4627, lightly modified

print json.dumps(objekt)
```

Et un programme pour lire du JSON et le charger dans un objet Python :

```
import json

# One backslash for Python, one for JSON
objekt = json.loads("""
{
      "Image": {
          "Width":  800,
          "Height": 600,
          "Title":  "View from Smith's, 15th Floor, \\\"Nice\\\"",
          "Thumbnail": {
              "Url":    "http://www.example.com/image/481989943",
              "Height": 125,
              "Width":  "100"
          },
          "IDs": [116, 943, 234, 38793]
        }
   }
""") # Example from RFC 4267, lightly modified

print objekt
print ""
print objekt["Image"]["Title"]
```

Le code ci-dessus est très simple car Python (comme Perl ou Ruby ou, bien sûr, JavaScript) a un typage complètement dynamique. Dans les langages où le typage est plus statique, c'est moins facile et on devra souvent utiliser des méthodes dont certains programmeurs se méfient, comme des conversions de types à l'exécution. Si vous voulez le faire en Go, il existe un bon article d'introduction <http://blog.golang.org/2011/01/json-and-go.html> au paquetage standard `json`.

Pour Java, qui a le même « problème » que Go, il existe une quantité impressionnante de bibliothèques différentes pour faire du JSON (on trouve en ligne plusieurs tentatives de comparaison <http://www.rojotek.com/blog/2009/05/07/a-review-of-5-java-json-libraries/>). J'ai utilisé JSON Simple <http://code.google.com/p/json-simple/>. Lire un texte JSON ressemble à :

```
import org.json.simple.*;
...
Object obj=JSONValue.parse(args[0]);
if (obj == null) { // May be use JSONParser instead, it raises an exception when there is a problem
    System.err.println("Invalid JSON text");
    System.exit(1);
} else {
    System.out.println(obj);
}

JSONObject obj2=(JSONObject)obj; // java.lang.ClassCastException if not a JSON object
System.out.println(obj2.get("foo")); // Displays member named "foo"
```

Et le produire :

```
JSONObject obj3=new JSONObject();
obj3.put("name","foo");
obj3.put("num",new Integer(100));
obj3.put("balance",new Double(1000.21));
obj3.put("is_vip",new Boolean(true));
```

JSON dispose d'une page Web officielle <http://json.org>, où vous trouverez plein d'informations. Pour tester dynamiquement vos textes JSON, il y a ce service <http://www.json.fr/>. Il y a aussi des gens critiques, voir par exemple, cette analyse détaillée <http://seriot.ch/parsing_json.php> (avec beaucoup de références au RFC).