RFC 8259 : The JavaScript Object Notation (JSON) Data Interchange Format

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 14 décembre 2017

Date de publication du RFC : Décembre 2017

https://www.bortzmeyer.org/8259.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 7159[1], avec peu de changements), est actuellement le plus à la mode. Comme son concurrent XML, c'est un format textuel, et il permet de représenter des structures de données hiérarchiques.

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 (enfin, approximativement <http://timelessrepo.com/json-isnt-a-javascript-subset>). La norme officielle de JavaScript est à l'ECMA, dans ECMA-262 <http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf>. JSON est dans la section 24.5 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. Les deux normes, ce RFC et la norme ECMA, sont écrites de manière différente mais, en théorie, doivent aboutir au même résultat. ECMA et l'IETF sont censés travailler ensemble pour résoudre les incohérences (aucune des deux organisations n'a, officiellement, le dernier mot).

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/rfc7159.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). Un texte JSON n'est pas forcément un objet ou un tableau, par exemple :

```
"Hello world!"
```

est un texte JSON légal (composé d'une chaîne de caractères en tout et pour tout). Une des conséquences est qu'un lecteur de JSON qui lit au fil de l'eau peut ne pas savoir si le texte est fini ou pas (il ne suffit pas de compter les crochets et accolades). À part les objets, les tableaux et les chaînes de caractères, un texte JSON peut être un nombre, ou bien un littéral, `false`, `true` ou `null`.

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 est obligatoirement UTF-8 dès qu'on envoie du JSON par le réseau (bien des mises en œuvre de JSON ne peuvent en lire aucun autre). Les textes JSON transmis par le réseau ne doivent pas utiliser de BOM.

Lorsqu'on envoie du JSON par le réseau, le type MIME à utiliser est `application/json`.

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, remplacé ensuite par le RFC 7159. Quels changements apporte cette troisième révision (annexe A) ? Elle corrige quelques erreurs, résout quelques incohérences avec le texte ECMA, et donne des avis pratiques aux programmeurs. Les principaux changements :

— Passage d'ECMA-262 seul à ECMA-404 (262 était pour tout JavaScript, 404 pour JSON seul).
— Correction des erreurs, dont une technique, une affirmation trop optimiste sur la compatibilité de JSON avec JavaScript <`https://www.rfc-editor.org/errata/eid3915`>.
— UTF-8 est désormais obligatoire.
— Progression sur le chemin des normes, de « Proposition de norme » à Norme tout court.

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`. Un exemple en Go figure plus loin, pour analyser la liste des stations de la RATP.

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));
```

Voyons maintenant des exemples réels avec divers outils de traitement de JSON. D'abord, les données issues du service de vélos en libre-service Vélo'v. C'est un gros JSON qui contient toutes les données du système. Nous allons programmer en Haskell un programme qui affiche le nombre de vélos libres et le nombre de places disponibles. Il existe plusieurs bibliothèques pour faire du JSON en Haskell mais Aeson <`https://hackage.haskell.org/package/aeson`> semble la plus utilisée. Haskell est un langage statiquement typé, ce qui est loin d'être idéal pour JSON. Il faut déclarer des types correspondant aux structures JSON :

```
data Velov =
  Velov {values :: [Station]} deriving Show

instance FromJSON Velov where
 parseJSON (Object v) =
    Velov <$> (v .: "values")

data Station =
  Station {stands :: Integer,
           bikes :: Integer,
           available :: Integer} deriving Show

data Values = Values [Station]
```

Mais ça ne marche pas : les nombres dans le fichier JSON ont été représentés comme des chaînes de caractères ! (Cela illustre un problème fréquent dans le monde de JSON et de l'"*open data*" : les données sont de qualité technique très variable.) On doit donc les déclarer en `String` :

```
data Station =
  Station {stands :: String,
           bikes :: String,
           available :: String} deriving Show
```

Autre problème, les données contiennent parfois la chaîne de caractères `None`. Il faudra donc filtrer avec la fonction Haskell `filter`. La fonction importante filtre les données, les convertit en entier, et en fait la somme grâce à `foldl` :

```
sumArray a =
  show (foldl (+) 0 (map Main.toInteger (filter (\i -> i /= "None") a)))
```

Le programme complet est (en ligne sur `https://www.bortzmeyer.org/files/velov.hs`). Une fois compilé, testons-le :

```
% curl -s https://download.data.grandlyon.com/ws/rdata/jcd_jcdecaux.jcdvelov/all.json | ./velov
"Stands: 6773"
"Bikes: 2838"
"Available: 3653"
```

Je n'ai pas utilisé les dates contenues dans ce fichier mais on peut noter que, si elles sont exprimées en ISO 8601 (ce n'est hélas pas souvent le cas), c'est sans indication du fuseau horaire (celui en vigueur à Lyon, peut-on supposer).

Un autre exemple de mauvais fichier JSON est donné par Le Monde avec la base des députés français <http://www.lemonde.fr/les-decodeurs/article/2017/06/25/comment-le-monde-a-enquete-sur-le 5150847_4355770.html>. Le fichier <http://s1.lemde.fr/assets-redaction/pol/quisontvosdeputes/ js/donnees.js> est du JavaScript, pas du JSON (il commence par une déclaration JavaScript `var datadep = {...}`) et il contient plusieurs erreurs de syntaxe (des apostrophes qui n'auraient pas dû être échappées).

Voyons maintenant un traitement avec le programme spécialisé dans JSON, jq <https://www. bortzmeyer.org/jq.html>. On va servir du service de tests TLS , dont les résultats sont consultables avec un navigateur Web, mais également téléchargeables en JSON. Par exemple, `https://tls.imirhil.fr/https/www` donne accès aux résultats des tests pour la version HTTPS de ce blog <https://www.bortzmeyer. org/https-blog.html> :

```
% curl -s https://tls.imirhil.fr/https/www.bortzmeyer.org.json| jq '.date'
"2017-07-23T14:10:25.760Z"
```

Notons qu'au moins une clé d'un objet JSON n'est pas nommée uniquement avec des lettres et chiffres, la clé `$oid`. jq n'aime pas cela :

```
% curl -s https://tls.imirhil.fr/https/www.bortzmeyer.org.json| jq '._id.$oid'
jq: error: syntax error, unexpected '$', expecting FORMAT or QQSTRING_START (Unix shell quoting issues?) at <top
._id.$oid
jq: 1 compile error
```

Il faut mettre cette clé entre guillemets :

```
% curl -s https://tls.imirhil.fr/https/bortzmeyer.org.json| jq '."_id"."$oid"'
"596cb76c2525939a3b34120f"
```

Toujours avec jq, les données de la Deutsche Bahn, en . C'est du GeoJSON (RFC 7946), un profil de JSON. Ici, on cherche la gare de Ratisbonne :

```
%  jq '.features | map(select(.properties.geographicalName == "Regensburg Hbf"))' railwayStationNodes.geojso
[
  {
    "type": "Feature",
    "geometry": {
      "type": "Point",
      "coordinates": [
        12.09966625451,
        49.011754555481
      ]
    },
    "properties": {
      "id": "SNode-1492185",
      "formOfNode": "railwayStop",
      "railwayStationCode": "NRH",
      "geographicalName": "Regensburg Hbf",
...
```

Toujours avec jq, on peut s'intéresser aux données officielles états-uniennes en `https://catalog.data.gov/da` `format=JSON`. Prenons les données sur la délinquance à Los Angeles (j'ai bien dit délinquance et pas criminalité, celui qui traduit "*crime*" par crime ne connait pas l'anglais, ni le droit). `https://data.lacity.org/api` est un très gros fichier (805 Mo) et jq n'y arrive pas :

```
% jq .data la-crime.json
error: cannot allocate memory
```

Beaucoup de programmes qui traitent le JSON ont ce problème (un script Python produit un `MemoryError`) : ils chargent tout en mémoire et ne peuvent donc pas traiter des données de grande taille. Il faut donc éviter de produire de trop gros fichiers JSON.

Si vous voulez voir un vrai exemple en Python, il y a mon article sur le traitement de la base des codes postaux <`https://www.bortzmeyer.org/dns-code-postal-lonlat.html`>. Cette base peut évidemment aussi être examinée avec jq. Et c'est l'occasion de voir du GeoJSON :

```
% jq '.features | map(select(.properties.nom_de_la_commune == "LE TRAIT"))' laposte_hexasmal.geojson
[
  {
    "type": "Feature",
    "geometry": {
      "type": "Point",
      "coordinates": [
        0.820017087099,
        49.4836816397
      ]
    },
    "properties": {
      "nom_de_la_commune": "LE TRAIT",
      "libell_d_acheminement": "LE TRAIT",
      "code_postal": "76580",
      "coordonnees_gps": [
        49.4836816397,
        0.820017087099
      ],
      "code_commune_insee": "76709"
    }
  }
]
```

J'avais promis plus haut un exemple écrit en Go. On va utiliser la liste des positions géographiques des stations RATP, en https://data.ratp.fr/explore/dataset/positions-geographiques-des-stations-name&disjunctive.code_postal&disjunctive.departement. Le programme Go (en ligne sur https://www.bortzmeyer.org/files/read-ratp.go) va afficher le nombre de stations et la liste :

```
% ./read-ratp < positions-geographiques-des-stations-du-reseau-ratp.json
26560 stations
Achères-Ville
Alésia
Concorde
...
```

Comme déjà indiqué, c'est plus délicat en Go que dans un langage très dynamique comme Python. Il faut construire à l'avance des structures de données :

```
type StationFields struct {
Fields Station
}

type Station struct {
Stop_Id   int
Stop_Name string
}
```

Et toute violation du « schéma » des données par le fichier JSON (quelque chose qui arrive souvent dans la nature) plantera le programme.

Si on veut beaucoup de fichiers JSON, le service de données ouvertes officielles data.gouv.fr permet de sélectionner des données par format. Ainsi, donnera tous les fichiers en JSON. Prenons au hasard les frayères du centre de la France, https://www.data.gouv.fr/fr/datasets/points-de-frayere-des-especes- Il est encodé en ISO-8859-1, ce qui est explicitement interdit par le RFC. Bref, il faut encore rappeler qu'on trouve de tout dans le monde JSON et que l'analyse de fichiers réalisés par d'autres amène parfois des surprises.

On peut aussi traiter du JSON dans PostgreSQL. Bien sûr, il est toujours possible (et sans doute parfois plus avantageux) d'analyser le JSON avec une des bibliothèques présentées plus haut, et de mettre les données dans une base PostgreSQL. Mais on peut aussi mettre le JSON directement dans PostgreSQL et ce SGBD fournit un type de données JSON <https://www.postgresql.org/docs/current/static/datatype-json.html> et quelques fonctions permettant de l'analyser <https://www.postgresql.org/docs/current/static/functions-json.html>. Pour les données, on va utiliser les centres de santé en Bolivie, en http://geo.gob.bo/geoserver/web/?wicket:bookmarkablePage=:org. On crée la table :

```
CREATE TABLE centers (
 ID serial NOT NULL PRIMARY KEY,
 info json NOT NULL
);
```

Si on importe le fichier JSON bêtement dans PostgreSQL (psql -c "copy centers(info) from stdin" mydb < centro-salud.json), on récupère un seul enregistrement. Il faut donc éclater le fichier JSON en plusieurs lignes. On peut utiliser les extensions à SQL de PostgreSQL pour cela <https://stackoverflow.com/questions/39224382/how-can-i-import-a-json-file-into-postgresql#answer-39224859>, mais j'ai préféré me servir de jq :

```
%  jq --compact-output '.features | .[]' centro-salud.json |  psql -c "copy centers(info) from stdin" mydb
COPY 50
```

On peut alors faire des requêtes dans le JSON, avec l'opérateur `->`. Ici, le nom des centres (en jq, on aurait écrit `.properties.nombre`) :

```
mydb=> SELECT info->'properties'->'nombre' AS Nom FROM centers;
                   nom
-----------------------------------------
 "P.S. ARABATE"
 "INSTITUTO PSICOPEDAGOGICO"
 "HOSPITAL GINECO OBSTETRICO"
 "HOSPITAL GASTROENTEROLOGICO"
 "C.S. VILLA ROSARIO EL TEJAR"
 "C.S. BARRIO JAPON"
 "C.S. SAN ANTONIO ALTO (CHQ)"
 "C.S. SAN JOSE (CHQ)"
 "C.S. SAN ROQUE"
...
```

Bon, sinon, 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 <https://www.json.fr/>.