RFC 6902 : JavaScript Object Notation (JSON) Patch

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 3 avril 2013

Date de publication du RFC : Avril 2013

https://www.bortzmeyer.org/6902.html

Le format de données structurées JSON, normalisé dans le RFC 8259 [1], a pris une importance de plus en plus grande et est désormais utilisé dans bien des contextes. Cela entraine l'apparition de normes auxiliaires, spécifiant comment faire telle ou telle opération sur des fichiers JSON. Ainsi, le RFC 6901 indique comment désigner une **partie** d'un document JSON et ce RFC 6902, sorti en même temps, indique comment exprimer les **modifications** ("*patches*") à un document JSON.

Logiquement, le "*patch*" est lui-même un fichier JSON (de type `application/json-patch`, cf. la section 6 de notre RFC) et il indique les opérations à faire (ajouter, remplacer, supprimer) à un endroit donné du document (identifié par un pointeur JSON, ceux du RFC 6901). Voici un exemple de "*patch*" :

```
[
     { "op": "add", "path": "/baz", "value": "qux" }
]
```

Il va ajouter (opération `add`) un membre nommé `baz` et de valeur `qux`. Si le document originel était :

```
{ "foo": "bar"}
```

Celui résultant de l'application du "*patch*" sera :

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

```
{
     "baz": "qux",
     "foo": "bar"
   }
```

À noter que ces "*patches*" travaillent sur le modèle de donnéees JSON, pas directement sur le texte (autrement, on utiliserait le traditionnel format diff), et peuvent donc s'appliquer sur des données qui ont un modèle similaire, quelle que soit la syntaxe qu'ils avaient. Dans les exemples de code Python à la fin, le "*patch*" est appliqué à des variables Python issues d'une analyse d'un fichier JSON, mais qui pourraient avoir une autre origine. (Au passage, je rappelle l'existence d'un format équivalent pour XML, le "*XML patch*" du RFC 5261.)

Le "*patch*" peut être envoyé par divers moyens, mais l'un des plus courants sera sans doute la méthode `PATCH` de HTTP (RFC 5789). Voici un exemple d'une requête HTTP avec un "*patch*" :

```
PATCH /my/data HTTP/1.1
Host: example.org
Content-Length: 326
Content-Type: application/json-patch
If-Match: "abc123"

[
     { "op": "test", "path": "/a/b/c", "value": "foo" },
     { "op": "remove", "path": "/a/b/c" },
     { "op": "add", "path": "/a/b/c", "value": [ "foo", "bar" ] },
     { "op": "replace", "path": "/a/b/c", "value": 42 },
     { "op": "move", "from": "/a/b/c", "path": "/a/b/d" },
     { "op": "copy", "from": "/a/b/d", "path": "/a/b/e" }
]
```

Un "*patch*" JSON a quelle forme (section 3) ? C'est un tableau dont chaque object qui le compose est une opération, à appliquer dans l'ordre du tableau (dans le premier exemple ci-dessus, le tableau ne comportait qu'une seule opération). L'entité à laquelle on applique le "*patch*" est donc transformée successivement et chaque opération s'applique au résultat de l'opération précédente.

Quelles sont les opérations possibles (section 4) ? Le membre `op` indique l'opération, `path` la cible et `value` la nouvelle valeur. `op` peut être :

— `add` : si la cible est dans un tableau, on ajoute la valeur à ce tableau (si le pointeur pointe vers la fin du tableau, avec la valeur `-`, on ajoute après le dernier élément). Si la cible est un membre inexistant d'un objet, on l'ajoute. S'il existe, on le remplace (ce point a été vigoureusement critiqué au sein du groupe de travail, où plusieurs personnes regrettaient qu'on mélange les sémantiques de `add` et de `replace` ; parmi les propositions alternatives, il y avait eu de nommer cette opération `set` ou `put` plutôt qu'`add`).
— `remove` : on retire l'élément pointé.
— `replace` : on remplace la valeur pointée.
— `move` : on déplace l'élément (ce qui nécessite un paramètre supplémentaire, `from`).
— `copy` : on copie la valeur pointée vers un nouvel endroit.
— `test` : teste si une valeur donnée est présente à l'endroit indiqué. Cela sert lorsqu'on veut des opérations conditonnelles. Les opérations étant appliquées dans l'ordre, et la première qui échoue stoppant tout le "*patch*", un `test` peut servir à faire dépendre les opérations suivantes de l'état du document JSON.

Mais à quel endroit applique-t-on cette opération ? C'est indiqué par le membre `path` qui est un pointeur JSON (cf. RFC 6901). Enfin, `value` indique la valeur, pour les opérations qui en ont besoin (par exemple `add` mais évidemment pas `remove`).

La section 5 spécifie ce qui se passe en cas d'erreurs : le traitement s'arrête au premier problème. Pour le cas de la méthode HTTP `PATCH`, voir la section 2.2 du RFC 5789. `PATCH` étant atomique, dans ce cas, le document ne sera pas du tout modifié.

Quelles mises en œuvre existent ? Une liste est disponible en ligne <http://trac.tools.ietf.org/wg/appsawg/trac/wiki/JsonPatch>, ainsi que des jeux de test <https://github.com/json-patch/json-patch-tests>. Je vais ici essayer celle en Python, python-json-patch <https://github.com/stefankoegl/python-json-patch>. D'abord, il faut installer python-json-pointer et python-json-patch :

```
% git clone https://github.com/stefankoegl/python-json-pointer.git
% cd python-json-pointer
% python setup.py build
% sudo python setup.py install

% git clone https://github.com/stefankoegl/python-json-patch.git
% cd python-json-patch
% python setup.py build
% sudo python setup.py install
```

Ensuite, on écrit un programme Python qui utilise cette bibliothèque. On va le faire simple, il prend sur la ligne de commande deux arguments, un qui donne le nom du fichier JSON original et un qui donne le nom du fichier contenant le patch. Le résultat sera écrit sur la sortie standard :

```
#!/usr/bin/env python

import jsonpatch
import json
import sys

if len(sys.argv) != 3:
    raise Exception("Usage: patch.py original patchfile")

old = json.loads(open(sys.argv[1]).read())
patch = json.loads(open(sys.argv[2]).read())

new = jsonpatch.apply_patch(old, patch)

print json.dumps(new)
```

Maintenant, on peut écrire un fichier JSON de test, `test.json`, qui décrit ce RFC :

```
% cat test.json
{
  "Title": "JSON Patch",
  "Number": "NOT PUBLISHED YET",
  "Authors": [
       "P. Bryan"
   ]
}
```

On va maintenant essayer avec différents fichiers `patch.json` contenant des patches (l'annexe A contient plein d'autres exemples). Un ajout à un tableau :

```
% cat patch.json
[
     { "op": "add", "path": "/Authors/0", "value": "M. Nottingham" }
]

% python patch.py test.json patch.json
{"Title": "JSON Patch", "Number": "NOT PUBLISHED YET", "Authors": ["M. Nottingham", "P. Bryan"]}
```

Un changement d'une valeur :

```
% cat patch.json
[
     { "op": "replace", "path": "/Number", "value": "6902" }
]

% python patch.py test.json patch.json
{"Title": "JSON Patch", "Number": "6902", "Authors": ["P. Bryan"]}
```

Un ajout à un objet :

```
% cat patch.json
[
     { "op": "add", "path": "/Status", "value": "standard" }
]

% python patch.py test.json patch.json
{"Status": "standard", "Title": "JSON Patch", "Number": "NOT PUBLISHED YET", "Authors": ["P. Bryan"]}
```

Une suppression d'un champ d'un objet :

```
% cat patch.json
[
     { "op": "remove", "path": "/Authors" }
]

% python patch.py test.json patch.json
{"Title": "JSON Patch", "Number": "NOT PUBLISHED YET"}
```

Si vous aimez les controverses, un des débats les plus animés, plus d'un an avant la sortie du RFC, avait été de savoir si l'opération d'ajout devait se noter `add` (comme cela a finalement été choisi) ou bien simplement +. Le format diff traditionnel utilise des mnémoniques plutôt que des noms et cela peut accroître la lisibilité (cela dépend des goûts).

Merci à Fil pour sa relecture attentive qui a trouvé plusieurs problèmes techniques.