RFC 9651 : Structured Field Values for HTTP

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 6 octobre 2024

Date de publication du RFC : Septembre 2024

https://www.bortzmeyer.org/9651.html

Plusieurs en-têtes HTTP sont **structurés**, c'est-à-dire que le contenu n'est pas juste une suite de caractères mais est composé d'éléments qu'un logiciel peut analyser. C'est par exemple le cas de `Accept-Language:` ou de `Content-Disposition:`. Mais chaque en-tête ainsi structuré a sa propre syntaxe, sans rien en commun avec les autres en-têtes structurés, ce qui en rend l'analyse pénible. Ce nouveau RFC (qui remplace le RFC 8941 [1]) propose donc des types de données et des algorithmes que les futurs en-têtes qui seront définis pourront utiliser pour standardiser un peu l'analyse d'en-têtes HTTP. Les en-têtes structurés anciens ne sont pas changés, pour préserver la compatibilité. De nombreux RFC utilisent déjà cette syntaxe (RFC 9209, RFC 9211, etc).

Imaginez : vous êtes un concepteur ou une conceptrice d'une extension au protocole HTTP qui va nécessiter la définition d'un nouvel en-tête. La norme HTTP, le RFC 9110, section 16.3.2, vous guide en expliquant ce à quoi il faut penser quand on conçoit un en-tête HTTP. Mais même avec ce guide, les pièges sont nombreux. Et, une fois votre en-tête spécifié, il vous faudra attendre que tous les logiciels, serveurs, clients, et autres (comme Varnish, pour lequel travaille un des auteurs du RFC) soient mis à jour, ce qui sera d'autant plus long que le nouvel en-tête aura sa syntaxe spécifique, avec ses amusantes particularités. Amusantes pour tout le monde, sauf pour le programmeur ou la programmeuse qui devra écrire l'analyse.

La solution est donc que les futurs en-têtes structurés réutilisent les éléments fournis par notre RFC, ainsi que son modèle abstrait, et la sérialisation proposée pour envoyer le résultat sur le réseau. Le RFC recommande d'appliquer strictement ces règles, le but étant de favoriser l'interopérabilité, au contraire du classique principe de robustesse <https://www.bortzmeyer.org/principe-robustesse.html>, qui mène trop souvent à du code compliqué (et donc dangereux) car voulant traiter tous les cas et toutes les déviations. L'idée est que s'il y a la moindre erreur dans un en-tête structuré, celui-ci doit être ignoré complètement.

1. Pour voir le RFC de numéro NNN, `https://www.ietf.org/rfc/rfcNNN.txt`, par exemple `https://www.ietf.org/rfc/rfc8941.txt`

La syntaxe est malheureusement spécifiée sous forme d'algorithmes d'analyse. L'annexe C fournit toutefois aussi une grammaire en ABNF (RFC 5234).

Voici un exemple fictif d'en-tête structuré très simple (tellement simple que, si tous étaient comme lui, on n'aurait sans doute pas eu besoin de ce RFC) : `Foo-Example:` est défini comme ne prenant qu'un élément comme valeur, un entier compris entre 0 et 10. Voici à quoi il ressemblera en HTTP :

```
Foo-Example: 3
```

Il accepte des paramètres après un point-virgule, un seul paramètre est défini, `foourl` dont la valeur est un URI. Cela pourrait donner :

```
Foo-Example: 2; foourl="https://foo.example.com/"
```

Donc, ces solutions pour les en-têtes structurés ne serviront que pour les **futurs** en-têtes, pas encore définis, et qui seront ajoutés au registre IANA <https://www.iana.org/assignments/http-fields/http-fields.xml#field-names>. Imaginons donc que vous soyez en train de mettre au point une norme qui inclut, entre autres, un en-tête HTTP structuré. Que devez-vous mettre dans le texte de votre norme ? La section 2 de notre RFC vous le dit :

- Inclure une référence à ce RFC 9651.
- J'ai parlé jusqu'à présent d'en-tête structuré mais en fait ce RFC 9651 s'applique aussi aux pieds ("*trailers*") (RFC 9110, section 6.5 ; en dépit de leur nom, les pieds sont des en-têtes eux aussi). La norme devra préciser si elle s'applique seulement aux en-têtes classiques ou bien aussi aux pieds.
- Spécifier si la valeur sera une liste, un dictionnaire ou juste un élément (ces termes sont définis en section 3). Dans l'exemple ci-dessus `Foo-Example:`, la valeur était un élément, de type entier.
- Penser à définir la sémantique.
- Et ajouter les règles supplémentaires sur la valeur du champ, s'il y en a. Ainsi, l'exemple avec `Foo-Example:` imposait une valeur entière entre 0 et 10.

Une spécification d'un en-tête structuré ne peut que rajouter des contraintes à ce que prévoit ce RFC 9651. S'il en retirait, on ne pourrait plus utiliser du code générique pour analyser tous les en-têtes structurés.

Rappelez-vous que notre RFC est strict : si une erreur est présente dans l'en-tête, il est ignoré. Ainsi, s'il était spécifié que la valeur est un élément de type entier et qu'on trouve une chaîne de caractères, on ignore l'en-tête. Idem dans l'exemple ci-dessus si on reçoit `Foo-Example: 42`, la valeur excessive mène au rejet de l'en-tête.

Les valeurs peuvent inclure des paramètres (comme le `foourl` donné en exemple plus haut), et le RFC recommande d'ignorer les paramètres inconnus, afin de permettre d'étendre leur nombre sans tout casser.

On a vu qu'une des plaies du Web était le laxisme trop grand dans l'analyse des données reçues (c'est particulièrement net pour HTML). Mais on rencontre aussi des cas contraires, des systèmes (par exemple les pare-feux applicatifs) qui, trop fragiles, chouinent lorsqu'ils rencontrent des cas imprévus, parce que leurs auteurs avaient mal lu le RFC. Cela peut mener à l'ossification, l'impossibilité de faire évoluer l'Internet parce que des nouveautés, pourtant prévues dès l'origine, sont refusées. Une solution récente est le **graissage**, la variation délibérée des messages pour utiliser toutes les possibilités du protocole. (Un exemple pour TLS est décrit dans le RFC 8701.) Cette technique est recommandée par notre RFC.

La section 3 du RFC décrit ensuite les types qui sont les briques de base avec lesquelles on va pouvoir définir les en-têtes structurés. La valeur d'un en-tête peut être une liste, un dictionnaire ou un élément. Les listes et les dictionnaires peuvent à leur tour contenir des listes. Une liste est une suite de termes qui, dans le cas de HTTP, sont séparés par des virgules :

```
ListOfStrings-Example: "foo", "bar", "It was the best of times."
```

Et si les éléments d'une liste sont eux-mêmes des listes, on met ces listes internes entre parenthèses (et notez la liste vide à la fin, et l'espace comme séparateur) :

```
ListOfListsOfStrings-Example: ("foo" "bar"), ("baz"), ("bat" "one"), ()
```

Un en-tête structuré dont la valeur est une liste a toujours une valeur, la liste vide, même si l'en-tête est absent.

Un dictionnaire est une suite de termes nom=valeur. En HTTP, cela donnera :

```
Dictionary-Example: en="Applepie", fr="Tarte aux pommes"
```

Et nous avons déjà vu les éléments simples dans le premier exemple. Les éléments peuvent être de type entier, chaîne de caractères, booléen, identificateur, valeur binaire, date, et un dernier type plus exotique (lisez le RFC pour en savoir plus). L'exemple avec `Foo-Example:` utilisait un entier. Les exemples avec listes et dictionnaires se servaient de chaînes de caractères. Ces chaînes sont encadrées de guillemets (**pas** d'apostrophes). Compte-tenu des analyseurs HTTP existants, les chaînes de caractères ordinaires doivent être en ASCII (RFC 20). Si on veut envoyer de l'Unicode, il faut utiliser un autre type, "*Display String*". Quant aux booléens, notez qu'il faut écrire 1 et 0 (pas `true` et `false`), et préfixé d'un point d'interrogation.

Toutes ces valeurs peuvent prendre des paramètres, qui sont eux-mêmes une suite de couples clé=valeur, après un point-virgule qui les sépare de la valeur principale (ici, on a deux paramètres) :

```
ListOfParameters: abc;a=1;b=2
```

J'ai dit au début que ce RFC définit un modèle abstrait, et une sérialisation concrète pour HTTP. La section 4 du RFC spécifie cette sérialisation, et son inverse, l'analyse des en-têtes (pour les programmeur·ses seulement).

Ah, et si vous êtes fana de formats structurés, ne manquez pas l'annexe A du RFC, qui répond à la question que vous vous posez certainement depuis plusieurs paragraphes : pourquoi ne pas avoir tout simplement décidé que les en-têtes structurés auraient des valeurs en JSON (RFC 8259), ce qui évitait d'écrire un nouveau RFC, et permettait de profiter du code JSON existant ? Le RFC estime que le fait que les chaînes de caractères JSON soient de l'Unicode est trop risqué, par exemple pour l'interopérabilité. Autre problème de JSON, les structures de données peuvent être emboîtées indéfiniment, ce qui nécessiterait des analyseurs dont la consommation mémoire ne peut pas être connue et limitée. (Notre RFC permet des listes dans les listes mais cela s'arrête là : on ne peut pas poursuivre l'emboîtement.)

Une partie des problèmes avec JSON pourrait se résoudre en se limitant à un profil restreint de JSON, par exemple en utilisant le RFC 7493 comme point de départ. Mais, dans ce cas, l'argument « on a déjà un format normalisé, et mis en œuvre partout » tomberait.

Enfin, l'annexe A note également qu'il y avait des considérations d'ordre plutôt esthétiques contre JSON dans les en-têtes HTTP.

Toujours pour les programmeur·ses, l'annexe B du RFC donne quelques conseils pour les auteur·es de bibliothèques mettant en œuvre l'analyse d'en-têtes structurés. Pour les aider à suivre ces conseils, une suite de tests est disponible <https://github.com/httpwg/structured-field-tests>. Quant à une liste de mises en œuvre du RFC, vous pouvez regarder celle-ci <https://github.com/httpwg/wiki/wiki/Structured-Fields>.

Actuellement, il y a dans le registre des en-têtes <https://www.iana.org/assignments/http-fields/http-fields.xml#field-names> plusieurs en-têtes structurés, comme le `Accept-CH:` du RFC 8942 (sa valeur est une liste d'identificateurs), le `Cache-Status:` du RFC 9211 ou le `Client-Cert:` du RFC 9440.

Si vous voulez un exposé synthétique sur ces en-têtes structurés, je vous recommande cet article par un des auteurs du RFC <https://www.fastly.com/blog/improve-http-structured-headers>.

Voyons maintenant un peu de pratique avec une des mises en œuvre citées plus haut, http_sfv <https://github.com/mnot/http_sfv>, une bibliothèque Python. Une fois installée :

```
% python3
>>> import http_sfv
>>> my_item=http_sfv.Item()
>>> my_item.parse(b"2")
>>> print(my_item)
2
```

On a analysé la valeur « 2 » en déclarant que cette valeur devait être un élément et, pas de surprise, ça marche. Avec une valeur syntaxiquement incorrecte :

```
>>> my_item.parse(b"2, 3")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/stephane/.local/lib/python3.12/site-packages/http_sfv/util.py", line 61, in parse
    raise ValueError("Trailing text after parsed value")
ValueError: Trailing text after parsed value
```

Et avec un paramètre (il sera accessible après l'analyse, via le dictionnaire Python `params`) :

```
>>> my_item.parse(b"2; foourl=\"https://foo.example.com/\"")
>>> print(my_item.params['foourl'])
https://foo.example.com/
```

Avec une liste :

```
>>> my_list=http_sfv.List()
>>> my_list.parse(b"\"foo\", \"bar\", \"It was the best of times.\"")
>>> print(my_list)
"foo", "bar", "It was the best of times."
```

Et avec un dictionnaire :

```
>>> my_dict=http_sfv.Dictionary()
>>> my_dict.parse(b"en=\"Applepie\", fr=\"Tarte aux pommes\"")
>>> print(my_dict)
en="Applepie", fr="Tarte aux pommes"
>>> print(my_dict["fr"])
"Tarte aux pommes"
```

L'annexe D résume les principaux changements depuis le RFC 8941. Rien de trop crucial, à part le fait que l'ABNF est reléguée à une annexe. Sinon, il y a deux nouveaux types de base, pour les dates et les chaines de caractères en Unicode ("*Display Strings*").