RFC 7807 : Problem Details for HTTP APIs

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 1 avril 2016

Date de publication du RFC : Mars 2016

https://www.bortzmeyer.org/7807.html

Lorsqu'on fait une requête HTTP, on récupère un code à trois chiffres qui indique notamment si tout s'est bien passé (si le code commence par 2, c'est bon, s'il commence par 4 ou 5, c'est qu'il y a un problème). Ces codes ne sont pas toujours assez fins et bien des API de services REST (donc reposant sur HTTP) voudraient des précisions. Plutôt que de créer un nouveau code, ce RFC propose un mécanisme qui permet d'envoyer du JSON normalisé indiquant tous les détails sur le problème survenu.

Les codes de statut de HTTP sont définis dans la section 6 du RFC 7231 [1]. Parmi les plus célèbres, on note 200 (qui veut dire que tout s'est bien passé) ou 404 (qui indique que le serveur n'a pas trouvé la ressource demandée). Si le serveur veut fournir des détails, il envoie traditionnellement de l'HTML dans le corps de sa réponse. Le navigateur Web peut alors l'afficher. Mais si la requête n'était pas envoyée par un humain derrière son navigateur, si elle venait d'un client REST ? HTML ne convient alors pas et il faut passer à une information analysable par une machine. C'est ce que fait ce RFC, qui définit deux formats, un en JSON (RFC 8259) et un en XML. Le client REST va donc avoir un résumé simple (le code de statut) et tous les détails nécessaires s'il le souhaite. « Comprenant » l'erreur exacte, le client pourra même, dans certains cas, la corriger.

Le RFC utilise surtout des exemples avec l'erreur 403 "*Forbidden*" mais j'ai préféré me servir de 402 "*Payment required*". Ce code n'a jamais été clairement documenté <https://medium.com/@humphd/402-payment-required-95bc72f06fcd> (il est marqué « réservé pour un usage futur » dans le RFC 7231, section 6.5.3) et c'est sans doute pour cela que notre RFC ne l'utilise pas, mais je le trouve plus rigolo. Voici par exemple une page Web payante </faut-vraiment-payer.html> :

```
% wget https://www.bortzmeyer.org/faut-vraiment-payer.html
--2016-03-26 15:54:00--  https://www.bortzmeyer.org/faut-vraiment-payer.html
Resolving www.bortzmeyer.org (www.bortzmeyer.org)... 2001:4b98:dc0:41:216:3eff:fece:1902, 2605:4500:2:245b::42, 
Connecting to www.bortzmeyer.org (www.bortzmeyer.org)|2001:4b98:dc0:41:216:3eff:fece:1902|:443... connected.
HTTP request sent, awaiting response... 402 Payment Required
2016-03-26 15:54:00 ERROR 402: Payment Required.
```

1. Pour voir le RFC de numéro NNN, https://www.ietf.org/rfc/rfcNNN.txt, par exemple https://www.ietf.org/rfc/rfc7231.txt

On peut envisager plein de choses dans cette réponse analysable par une machine, comme un URI unique pour ce problème donné, qui pourrait être, par exemple, transmis au support pour faciliter la communication. Ou bien un URI d'un service REST de paiement permettant d'envoyer l'argent souhaité. Mais, naturellement, l'utilisation de cette réponse est facultative : parfois, le code de statut est suffisant et il n'y a rien à ajouter (c'est souvent le cas du 404), et parfois il vaut mieux utiliser un format spécifique à l'application utilisée (c'est d'ailleurs le cas pour toutes les API développées avant ce RFC). Ceci dit, pour les nouvelles applications, le mécanisme décrit dans ce RFC peut être très utile, pour doter toutes les applications d'un mécanisme commun de signalement des erreurs et problèmes.

Le modèle de données utilisé est celui de JSON et le type MIME est `application/problem+json`. Comme je l'ai dit plus haut, il y a aussi une version XML mais la référence est le JSON. La section 3 du RFC décrit ce modèle. Commençons par un exemple, une API qui demandait entre autres d'indiquer un âge et à qui on a envoyé un nombre négatif :

```
HTTP/1.1 400 Bad Request
Content-Type: application/problem+json
Content-Language: en

{
   "type": "https://example.net/validation-error",
   "title": "Your request parameters didn't validate.",
   "invalid-params": [ {
                         "name": "age",
                         "reason": "must be a positive integer"
                       }
                     ]
}
```

Ce message d'erreur est de type `https://example.net/validation-error`, une explication en langue naturelle est donnée par le membre `title`, et la liste des paramètres invalides est donnée dans le membre (une extension à la norme) `invalid-params`.

Quels sont les membres possibles de l'objet JSON renvoyé ?

— `type` est un URI (il est donc unique) servant d'identificateur au problème, par exemple lorsqu'on va écrire au support. Il est recommandé qu'il soit déréférençable (c'est-à-dire qu'on puisse le visiter avec un navigateur Web et obtenir des informations sur le problème en question). S'il est absent, sa valeur par défaut est le très vide `about:blank` (RFC 6694, section 3).

— `title` est un titre conçu pour des humains, par exemple pour les messages d'erreurs présentés à l'utilisateur. Étant en langue naturelle, il est, contrairement au `type`, ambigu. Il peut être adapté à la langue de l'utilisateur, via l'habituelle négociation de contenu de HTTP. `detail` (absent dans l'exemple ci-dessus) le complète éventuellement. Le RFC précise que `detail` est fait pour l'utilisateur, pas pour le programmeur qui a conçu le service. Il devrait donc contenir des informations aidant l'utilisateur à corriger sa requête, pas des informations de débogage (pas de pile d'appels Java, par exemple...) D'ailleurs, envoyer ces informations de débogage poserait un problème de sécurité (cf. section 5).

— `instance` (absent dans l'exemple) est un URI qui, contrairement à `type`, n'identifie pas la classe du problème mais une instance particulière. Si `http://example.com/not-enough-credit` indique la classe « pas assez d'argent », `https://example.com/account/12345/msgs/abc` va indiquer le problème d'argent d'un compte particulier.

Notez que `type` et `instance` peuvent être des URI relatifs.

Voici maintenant un exemple sur mon blog (c'est conçu comme une application REST donc les résultats sont toujours en JSON et, en effet, ce n'est pas en HTTPS, ce serait intolérable en production) :

```
% curl -v https://www.bortzmeyer.org/apps/payme
...
< HTTP/1.1 402 Payment required
...
< Content-Length: 253
< Content-Type: application/problem+json
<
{
  "type": "http://errors.bortzmeyer.org/nopay",
  "detail": "Bitcoin address 1HtNJ6ZFUc9yu9u2qAwB4tGdGwPQasQGax, Ethereum address 0xbe1f2ac71a9703275a4d3ea01a34
  "title": "You must pay"
}

% curl -v https://www.bortzmeyer.org/apps/payme\?pay=30
...
< HTTP/1.1 200 OK
< Content-Length: 36
< Content-Type: application/json
<
{
  "title": "OK, 30 credits paid"
}
```

Le code Python WSGI correspondant est :

```
def payme(start_response, environ):
    form = cgi.parse_qs(environ['QUERY_STRING'])
    response_headers = []
    amount = 0
    if form.has_key("pay"):
        try:
            amount = int(form["pay"][0])
        except ValueError: # Bad syntax
            amount = 0
    if amount > 0:
        status = '200 OK'
        response_headers.append(('Content-type', 'application/json'))
        output = json.dumps({"title": "OK, %i credits paid" % amount}, indent=2)
    else:
        status = '402 Payment required'
        response_headers.append(('Content-type', 'application/problem+json'))
        output = json.dumps({"type": "http://errors.bortzmeyer.org/nopay",
                             "title": "You must pay",
                             "detail": "Bitcoin address 1HtNJ6ZFUc9yu9u2qAwB4tGdGwPQasQGax, Ethereum address 0xb
                             indent=2)
    response_headers.append(('Content-Length', str(len(output))))
    start_response(status, response_headers)
    return [output]
```

Si vous n'avez pas ce que vous voulez dans les membres prévus, vous pouvez étendre l'objet JSON. Les clients doivent donc ignorer les membres inconnus. C'est le cas du `invalid-params` dans l'exemple, qui n'est pas un membre standard.

Bien, maintenant, vous êtes programmeur dans une "*start-up*", vous créez un nouveau service qui a une API, un nom de domaine en `.io`, un "*business plan*" pipeau et vous vous demandez si vous devez utiliser ce RFC et comment. La section 4 du RFC fournit quelques conseils. D'abord, je le répète, ce n'est pas un outil de débogage pour vous, ne serait-ce que pour des raisons de sécurité (cf. section 5). C'est

un outil pour aider vos utilisateurs. Ensuite, si le problème est un problème classique et standard, il est inutile de se servir de ce RFC. Si l'utilisateur demande une ressource qui n'existe pas, le traditionnel et générique 404 (RFC 7231, section 6.5.4) convient parfaitement et je ne vois pas de raison d'ajouter des détails analysables par une machine (dans une page HTML d'erreur, c'est différent, on peut fournir des conseils aux visiteurs, mais rappelez-vous que ce RFC est pour les API, quand le client est un programme).

D'autre part, une application peut avoir de très bonnes raisons d'utiliser un format à elle pour décrire en détail les problèmes. (Sans compter les applications existantes qui ne vont évidemment pas modifier la définition de leur API juste pour coller à ce RFC.)

En revanche, une application nouvelle, qui n'a pas de format d'erreur établi, a tout intérêt à utiliser le cadre de ce RFC plutôt que de réinventer la roue. Dans ce cas, vous allez devoir définir :

— L'URI qui servira de `type` (le seul type prédéfini est `about:blank`),
— Le code de statut HTTP qui l'accompagne,
— Les éventuelles extensions (comme le membre `invalid-params` plus haut).

Ces extensions peuvent utiliser les liens de HTTP (RFC 8288).

Le format principal décrit par ce RFC utilise JSON. Mais, comme il y a des goûts différents, il y a aussi une variante XML, décrite dans l'annexe A. Elle est spécifiée en Relax NG. Le modèle de données étant le même, cela donne à peu près :

```
start = problem

problem =
  element problem {
    (  element  type            { xsd:anyURI }?
     & element  title           { xsd:string }?
     & element  detail          { xsd:string }?
     & element  status          { xsd:positiveInteger }?
     & element  instance        { xsd:anyURI }? ),
    anyNsElement
  }
```

Et le résultat serait :

```
HTTP/1.1 400 Bad Request
Content-Type: application/problem+xml
Content-Language: en

<?xml version="1.0" encoding="UTF-8"?>
<problem xmlns="urn:ietf:rfc:XXXX">
   <type>https://example.net/validation-error</type>
   <title>Your request parameters didn't validate.</title>
   <invalid-params><param><name>age</name><reason>must be a positive integer</reason></param></invalid-param
</problem>
```

On a presque fini, quelques petits mots sur la sécurité en section 5 : attention à ne pas laisser fuiter de l'information qui pourrait aider un attaquant à préparer son attaque. Il faut notamment se méfier des détails de mise en œuvre (du genre afficher la requête SQL qui a échoué...)

Les deux nouveaux types MIME, `application/problem+json` et `application/problem+xml` figurent désormais dans le registre IANA <https://www.iana.org/assignments/media-types/application/problem+json>.

Les développeurs d'API n'ont pas attendu ce RFC pour renvoyer des messages d'erreurs structurés, utilisant d'autres schémas (voici, par exemple, les erreurs possibles de l'API Github <https://developer.github.com/v3/#client-errors>). Un concurrent sérieux à ce RFC est, par exemple, qui a son propre mécanisme de signalement d'erreur <http://jsonapi.org/format/#errors>.