RFC 6570 : URI Template

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 28 mars 2012

Date de publication du RFC : Mars 2012

https://www.bortzmeyer.org/6570.html

Beaucoup d'applications Web utilisent des URI (RFC 3986 [1]) qui peuvent être décrits par un gabarit ("*template*"), avec des variables qui seront incarnées pour produire l'URI final. Ce RFC, issu d'une coopération entre l'IETF et le W3C normalise un langage (déjà largement déployé) pour ces gabarits.

La section 1.1 du RFC fournit quelques exemples d'URI obéissant à un gabarit. Par exemple, des pages personnelles identifiées par le tilde :

```
http://example.com/~fred/
http://example.com/~mark/
```

ou des entrées d'un dictionaire :

```
http://example.com/dictionary/c/cat
http://example.com/dictionary/d/dog
```

ou encore un moteur de recherche :

```
http://example.com/search?q=cat&lang=en
http://example.com/search?q=chien&lang=fr
```

1. Pour voir le RFC de numéro NNN, https://www.ietf.org/rfc/rfcNNN.txt, par exemple https://www.ietf.org/rfc/rfc3986.txt

À chaque fois, le patron de conception sous-jacent est assez clair. Le langage décrit dans ce RFC 6570 permet d'**abstraire** ce patron dans un gabarit d'URI. Les trois exemples cités plus haut pourront être décrits :

```
http://example.com/~{username}/

http://example.com/dictionary/{term:1}/{term}

http://example.com/search{?q,lang}
```

Cela permet d'abstraire un URI, en montrant clairement au lecteur le processus de construction. Cela contribue donc à la création de beaux URL <https://www.bortzmeyer.org/beaux-urls.html>, qui ne sont pas simplement des identificateurs opaques, mais qui peuvent être compris par l'utilisateur du Web.

En dehors de l'intérêt intellectuel, quelle est l'utilité de ce système ? Il permet notamment la création automatique d'URI, à partir de gabarits et de valeurs pour les variables (comme la variable `term` dans le deuxième exemple ci-dessus). Les protocoles fondés sur REST en seront ravis. Ainsi, le gabarit `http://www.example.com/foo{?query,number}` combiné avec les variables {query : "mycelium", number : 100} va donner `http://www.example.com/foo?query=mycelium&number=100`. Si `query` n'était pas défini, cela serait simplement `http://www.example.com/foo?number=100`.

Pour prendre des exemples réels, on peut décrire les URL d'OpenStreetMap comme tirés du gabarit `http://www.openstreetmap.org/?{lon,lat}` et ceux de Wikipédia comme faits à partir de `http://fr.wikipedia.org/wiki/{topic}`.

La section 1.2 explique qu'il existe en fait plusieurs niveaux pour le langage de description de gabarits. Une mise en œuvre donnée s'arrête à un certain niveau, les premiers étant les plus simples. Le **niveau 1** fournit simplement l'expansion des variables (qui sont placées entre accolades), avec échappement. Si `var` vaut "Bonjour le monde", l'expansion de {var} sera `Bonjour%20le%20monde`.

Le **niveau 2** ajoute un opérateur + pour les variables pouvant contenir des caractères spéciaux, et le # pour les identificateurs de fragment. Si `path` vaut "/foo/bar", {path} est expansé en `%2Ffoo%2Fbar` alors que {+path} donnera `/foo/bar`. Et avec la variable `var` valant "Bonjour le monde", l'expansion de {#var} sera `#Bonjour%20le%20monde` permettant de construire des URI avec un identificateur pointant vers un point précis du document. Par exemple, les URL des "*Internet-Drafts*" dans la file d'attente de l'éditeur des RFC ont le gabarit `http://www.rfc-editor.org/queue2.html{#draftname}` avec, par exemple, `draftname` égal `draft-ietf-lisp`.

Au **niveau 3**, cela devient bien plus riche : on a plusieurs variables dans une expression (le gabarit `map?{x,y}` devient l'URI `map?1024,768` si x=1024 et y=768). Et on gagne surtout les paramètres mutiples (séparés par &) si utilisées dans les formulaires Web : le gabarit {?x,y} devient `?x=1024&y=768`.

Enfin, le **niveau 4** ajoute des modificateurs après le nom de la variable. Par exemple, : permet d'indiquer un nombre limité de caractères (comme dans le `term:1` que j'ai utilisé au début, qui indique la première lettre). Et le * indique une variable composite (une liste ou un dictionnaire) qui doit être elle-même expansée. Si la variable `couleurs` vaut la liste {rouge, bleu, vert}, le gabarit (de niveau 1) {couleurs} donnera `rouge,bleu,vert` alors que la gabarit de niveau 4 {?couleurs*} vaudra `?couleurs=rouge&couleurs=bleu&couleurs=vert`.

Des mécanismes similaires à ces gabarits existent dans d'autres langages comme OpenSearch ou WSDL. Par exemple, le gabarit décrivant les URL du moteur de recherche de mon blog `</search>` a la forme `http://www.bortzmeyer.org/search{?pattern}` et, dans dans sa description OpenSearch `<https://www.bortzmeyer.org/opensearch.html>`, on trouve la même information (quoique structurée différemment).

Comme l'explique la section 1.3, cette nouvelle norme vise à unifier le langage de description de gabarits entre les langages, tout en restant compatible avec les anciens systèmes. Normalement, la syntaxe des gabarits d'URI est triviale à analyser, tout en fournissant la puissance expressive nécessaire. Il s'agit non seulement de générer des URI à partir d'un patron, mais aussi de pouvoir capturer l'essence d'un URI, en exprimant clairement comment il est construit.

La section 2 décrit rigoureusement la syntaxe. En théorie, un analyseur qui ne gère que le niveau 1 est acceptable. Toutefois, le RFC recommande que tous les analyseurs comprennent la syntaxe de tous les niveaux (même s'ils ne savent pas faire l'expansion), ne serait-ce que pour produire des messages d'erreurs clairs (≪ *"This feature is only for Level 3 engines and I'm a poor Level 2 program."* ≫ .

Cette section contient donc la liste complète des caractères qui ont une signification spéciale pour le langage de gabarits, comme +,? ou &, ainsi que de ceux qui sont réservés pour les futures extensions (comme = ou @). En revanche, $ et les parenthèses sont exprèssement laissés libres pour d'autres langages qui voudraient décrire une part de l'URI.

Ensuite, la section 3 se consacre aux détails du processus d'expansion qui, partant d'un gabarit et d'un ensemble de valeurs, va produire un URI. Notez qu'une variable qui n'a pas de valeur ne donne pas un 0 ou un `N/A` ou quoi que ce soit de ce genre. Elle n'est pas expansée. Donc, le gabarit `{?x,y}` avec x valant 3 et y n'étant pas défini, devient `?x=3` et pas `?x=3&y=NULL`. Si x n'était pas défini non plus, le résultat serait une chaîne vide (sans même le point d'interrogation).

Est-ce que ces gabarits peuvent poser des problèmes de sécurité ? La section 4 regarde cette question et conclus que cela dépend de qui fournit le gabarit, qui fournit les variables, et comment va être utilisé l'URI résultant (aujourd'hui, les mises en œuvre existantes les utilisent dans des contextes très variés). Par exemple, des variables contenant du code JavaScript peuvent être dangereuses si l'URI résultant est traité par ce langage. Des attaques de type XSS sont donc parfaitement possibles si le programmeur ne fait pas attention.

Pour les programmeurs, l'annexe A contient un ensemble de recommandations sur la mise en œuvre de ce RFC, ainsi qu'un algorithme possible pour cette mise en œuvre. Il existe plusieurs mises en œuvre de ce RFC `<http://code.google.com/p/uri-templates/wiki/Implementations>`. Malheureusement, toutes celles que j'ai essayées semblent limitées au niveau 1 et ne le disent pas clairement.

Le code Python en `<http://code.google.com/p/uri-templates/>` (niveau 1 et un bout du niveau 2) :

```
import uritemplate
import simplejson
import sys

vars = {"foo": "bar/"}

def test_print(template):
    actual = uritemplate.expand(template, vars)
    print actual

test_print("http://www.example.org/thing/{foo}")
test_print("http://www.example.org/thing/{+foo}")
```

On obtient :

```
http://www.example.org/thing/bar%2F
http://www.example.org/thing/bar/
```

Avec le module Perl `URI::Template` (niveau 1), on peut écrire :

```
use URI::Template;
my $template = URI::Template->new( 'http://example.com/{foo}' );
my $uri      = $template->process( foo => 'Hello World/' );
print $uri, "\n";
```

Le code Haskell en <http://hackage.haskell.org/cgi-bin/hackage-scripts/package/uri-template> (niveau 1 seulement) écrit ainsi :

```
module Main(main) where

import Network.URI.Template

testEnv :: TemplateEnv
testEnv =
  addToEnv "foo" "Hello World/" $
    addToEnv "x" "1024" $
     addToEnv "y" "768" $
       newEnv

main :: IO ()
main = do
  putStrLn (expand testEnv "http://example.org/{foo}")
  return ()
```

Donne :

```
% runhaskell Test.hs
http://example.org/Hello%20World%2f
```