

Une application bidon avec REST, SQL et Unicode, juste comme exemple

Stéphane Bortzmeyer

<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 25 Juillet 2007. Dernière mise à jour le 31 Juillet 2007

<http://www.bortzmeyer.org/rest-sql-unicode-exemple.html>

Je décris ici une application Web qui ne sert à rien mais qui montre l'usage de plusieurs techniques qui sont apparemment souvent demandées : SQL, REST (sur lequel j'ai déjà écrit un article (<http://www.bortzmeyer.org/programmation-rest.html>)), Unicode, etc, le tout en Python.

Cette application m'a servi à tester différentes techniques que je voulais expérimenter, pour apprendre ou bien pour déployer dans d'autres applications.

Commençons par la décrire : l'application reçoit, via le Web, en suivant le protocole REST (elle a aussi une interface par un formulaire HTML) des chaînes de caractères en Unicode et elle les enregistre dans une base de données. On peut ensuite les extraire, les décomposer en caractères, etc.

Commençons par la base de données. Elle utilise PostgreSQL qui permet de stocker et de récupérer de l'Unicode (<http://www.bortzmeyer.org/postgresql-unicode.html>). Le schéma est constitué d'une seule table, `Words` (en fait, des chaînes de caractères, elles ne sont pas limitées à un seul mot). Cette table comprend une colonne `word` qui va stocker notre chaîne de caractères. Vous pouvez voir le schéma complet dans le fichier (en ligne sur <http://www.bortzmeyer.org/files/create-registry-unicode.sql>). Il tire profit des "triggers" de PostgreSQL pour que la colonne `updated` soit automatiquement mise à jour, libérant le programmeur de l'application de ce souci (et diminuant le risque de bogues).

Maintenant, l'application elle-même. Elle prend la forme d'un seul script Python, (en ligne sur <http://www.bortzmeyer.org/files/rest-registry.py>). Ce script est conçu pour utiliser le protocole CGI. Certes, ce protocole est vieux et a des défauts, mais il fonctionne avec tous les serveurs HTTP et il est très simple. On pourrait plutôt utiliser un module Apache écrit avec `mod_python` ou bien passer par SCGI mais ce sera pour une autre version.

Pour toute application REST, les deux premières questions à se poser sont :

- Quel est le schéma de nommage des URI ?

– Quels sont les méthodes (les verbes) HTTP utilisées ?

Ici, l'URI sera du type `PREFIX/STRING` où `PREFIX` dépendra de l'hébergement (par exemple, il vaudra `http://www.example.net/rest-registry`) et `STRING` sera la chaîne gérée. Si elle est vide, on supposera que l'action s'applique à tout le registre (dans un esprit similaire à celui de APP où l'absence d'une entrée signifie qu'on parle de toute la collection).

L'URI est donc choisi par le client HTTP à la création, comme avec WebDAV (mais contrairement à APP). On dit que le client **contrôle l'espace de nommage**.

Le fait d'utiliser l'URI pour stocker la chaîne qui nous intéresse entraîne des limites pour l'application (relevées par Kim-Minh Kaplan) : on ne peut pas enregistrer de chaîne qui contienne un caractère illégal pour un URL (comme le point d'interrogation ou le dièse).

Les verbes utilisés seront les classiques méthodes HTTP, GET pour récupérer une chaîne ou bien tout le registre, POST pour mettre à jour une chaîne ou bien le registre, DELETE pour détruire une chaîne (on ne peut pas détruire tout le registre), PUT pour enregistrer une nouvelle chaîne. On notera que POST appliqué à tout le registre permet la création d'une nouvelle chaîne, puisque créer une entrée dans le registre est une modification de celui-ci. Les codes de retour, à trois chiffres, sont également tirés de la norme HTTP ? le RFC 2616¹. Par exemple, nous renverrons 501 pour les méthodes non encore mise en œuvre.

Comme il est toujours préférable d'écrire le code qui utilise un service avant le service lui-même, voici des exemples d'utilisation avec l'excellent logiciel curl. Ils nous serviront de spécification.

Enregistrement d'une nouvelle chaîne (%20 est l'encodage de l'espace, qui est normalement interdit dans les URL) :

```
% curl -v --request PUT http://www.example.net/rest/registry/Pierre%20Louÿs
...
> PUT /rest/registry/Pierre%20Louÿs HTTP/1.1
User-Agent: curl/7.13.2 (i386-pc-linux-gnu) libcurl/7.13.2 OpenSSL/0.9.7e zlib/1.2.2 libidn/0.5.13
Host: www.example.net
Accept: */*

< HTTP/1.1 200 OK
< Date: Wed, 25 Jul 2007 18:49:12 GMT
< Server: Apache/2.0.58 (Gentoo) mod_python/3.2.10 Python/2.4.3 mod_ssl/2.0.58 OpenSSL/0.9.8d PHP/5.2.2-p11
< X-Script: registry running with Python 2.4.4
...
```

Si la chaîne existe déjà, nous décidons de renvoyer un code 403 (interdiction).

Récupération d'une chaîne existante :

¹Pour voir le RFC de numéro NNN, <http://www.ietf.org/rfc/rfcNNN.txt>, par exemple <http://www.ietf.org/rfc/rfc2616.txt>

```
% curl -v --request GET http://www.example.net/rest/registry/café
...
> GET /rest/registry/café HTTP/1.1
User-Agent: curl/7.13.2 (i386-pc-linux-gnu) libcurl/7.13.2 OpenSSL/0.9.7e zlib/1.2.2 libidn/0.5.13
Accept: */*

< HTTP/1.1 200 OK
< Date: Wed, 25 Jul 2007 16:26:27 GMT
< Server: Apache/2.0.58 (Gentoo) mod_python/3.2.10 Python/2.4.3 mod_ssl/2.0.58 OpenSSL/0.9.8d PHP/5.2.2-pl1-gent
< X-Script: registry running with Python 2.4.4
< Content-Type: text/html; charset=UTF-8
```

On note que le script ne fait malheureusement pas, à l'heure actuelle, de négociation de contenu (<http://www.bortzmeyer.org/negotiation-contenu-http.html>). Envoyer du `text/plain` serait plus logique dans le cas de `curl`.

Naturellement, une tentative de récupérer par `GET` une chaîne qui n'existe pas doit se traduire par un code de retour `404`.

Détruisons désormais une chaîne (actuellement, il n'y a aucun mécanisme d'autorisation, n'importe qui peut détruire) :

```
% curl -v --request DELETE http://www.example.net/rest/registry/ça%20va
> DELETE /rest/registry/ça%20va HTTP/1.1
User-Agent: curl/7.13.2 (i386-pc-linux-gnu) libcurl/7.13.2 OpenSSL/0.9.7e zlib/1.2.2 libidn/0.5.13
...

< HTTP/1.1 200 OK
< Date: Wed, 25 Jul 2007 18:54:27 GMT
< Server: Apache/2.0.58 (Gentoo) mod_python/3.2.10 Python/2.4.3 mod_ssl/2.0.58 OpenSSL/0.9.8d PHP/5.2.2-pl1-gent
< X-Script: registry running with Python 2.4.4
```

Naturellement, à la place de `curl`, on aurait pu utiliser n'importe quel client HTTP capable d'envoyer des méthodes autre que `GET` ou bien écrire un programme en s'appuyant sur les nombreuses bibliothèques existantes qui permettent de développer un client HTTP spécifique assez facilement.

Voyons maintenant le code qui va mettre en œuvre ces opérations. Son intégralité est disponible en (en ligne sur <http://www.bortzmeyer.org/files/rest-registry.py>). Nous chargeons les module `cgi` (<http://docs.python.org/lib/module-cgi.html>) et `urllib` (<http://docs.python.org/lib/module-urllib.html>) de Python :

```
import cgi
```

et nous analysons la méthode utilisée, que le protocole CGI met dans la variable d'environnement `REQUEST_METHOD` :

```
method = os.environ['REQUEST_METHOD']
request = urllib.unquote(os.environ['REQUEST_URI'])
request = unicode(request, web_encoding)
request = request.replace(prefix, "", 1)
form = cgi.FieldStorage()
```

<http://www.bortzmeyer.org/rest-sql-unicode-exemple.html>

Une fois que ces variables sont remplies, le programme peut vraiment commencer :

```
if method == "GET":
    ...
elif method == "PUT":
    ...
else: # Unknown method
    print headers("400 unknown method")
    print response("Unknown method %s" % method)
```

Le petit nombre de méthodes possible fait qu'une succession de `if` est la technique la plus simple.

Voyons le code complet pour la méthode DELETE :

```
try:
    delete(request)
    print headers()
    print response("%s deleted" % request)
except NotFound:
    print headers("404 Not found")
    print response("%s not found" % request)
```

Ce code s'appuie sur des fonctions `headers` et `response`, qui prennent en charge les détails du protocole HTTP.

Et la fonction `delete`? Elle va devoir parler à la base de données, ce qui va être notre prochaine étape. Pour parler au SGBD PostgreSQL, nous utilisons le module `psycopg` (<http://www.initd.org/tracker/psycopg>):

```
import psycopg2
import psycopg2.extras
```

et nous ouvrons la base au lancement du programme :

```
connection = psycopg2.connect("dbname=%s user=%s" % (db, db_user))
connection.set_isolation_level(psycopg2.extensions.ISOLATION_LEVEL_READ_COMMITTED)
cursor = connection.cursor()
cursor.execute("SET client_encoding TO %s;", (web_encoding, ))
```

Notez que l'encodage que nous utilisons est explicitement spécifié, pour éviter de dépendre de l'environnement.

Nous pouvons alors envoyer des requêtes SQL, par exemple, pour détruire une entrée de la base :

```
def delete(word):
    cursor.execute("DELETE FROM Words WHERE word = %s", (word, ))
    if cursor.rowcount == 0:
        raise NotFound
    cursor.execute("COMMIT")
```

Dans une future version, cette application fera peut-être de la négociation de contenu (<http://www.bortzmeyer.org/negociation-contenu-http.html>) et pourra envoyer par exemple du CSV ou du texte brut. Mais pour l'instant, elle n'envoie que du XHTML. Comment le générer ? On pourrait le faire entièrement à la main (<http://www.bortzmeyer.org/creer-xml-par-programme.html>), par des `print` dans le script Python mais il est plus simple d'utiliser un système de gabarits (<http://www.bortzmeyer.org/generer-html-avec-tal.html>) et nous nous servirons de TAL.

On importe le module :

```
from simpletal import simpleTAL, simpleTALES, simpleTALUtils
...
context = simpleTALES.Context()
```

Puis on définit les gabarits (ici, la table de toutes les chaînes de caractères) :

```
all_words_blurb = """
<div>
<h2>All the registry</h2>
<table id="registry">
<tr><th>Word</th><th>Origin</th><th>Creation</th></tr>
<tr tal:repeat="word allwords"><td><a tal:attributes="href word/word" tal:content="word/word"/></td><td tal:cont
</table>
</div>
"""
```

Ce gabarit pourra être incarné lorsqu'on renverra l'information sur une entrée de la base. Ici, on lit la base avec `cursor.fetchall()`, on met son contenu ans la variable Python `words` et on dit à TAL que la variable TAL `allwords` vaudra cette variable `words` :

```
all_words_template = simpleTAL.compileXMLTemplate(all_words_blurb)
...
for tuple in cursor.fetchall():
    words.append({'word': unicode(tuple[0], db_encoding), 'origin': tuple[1],
                'created': tuple[2], 'updated': tuple[3], 'comments': tuple[4]})
context.addGlobal("allwords", words)
print headers()
print response(title="All the registry", content=all_words_template)
```

La plupart des utilisateurs n'étant pas enthousiastes à l'idée d'utiliser l'application depuis la ligne de commande avec `curl`, nous fournissons également une interface avec un formulaire HTML. Voici le code HTML du formulaire :

```
<form method="POST">
<p>Type a word: <input type="text" name="word" /></p>
<!-- The syntax of the <textarea> element (no content, but a start and an end tag is because of a parsing bug in
<p>Type (optional) comments:<br/> <textarea cols="40" rows="20" name="comments"></textarea></p>
<p><input type="submit" name="register" value="Register it" /></p>
</form>
```

On note que l'activation du bouton *"Register it"* envoie une requête POST. À partir de là, c'est le code ordinaire qui s'en charge. En voici une petite partie. Notez que toutes les chaînes de caractères sont traduites en Unicode tout de suite :

<http://www.bortzmeyer.org/rest-sql-unicode-exemple.html>

```
word = unicode(form.getfirst("word", ""), web_encoding)
comments = unicode(form.getfirst("comments", ""), web_encoding)
if comments.strip() == "":
    comments = None
client = os.environ["REMOTE_ADDR"]
cursor.execute("""
    INSERT INTO Words (word, origin, useragent, comments)
        VALUES (%s, %s, %s, %s)""",
    (word, client,
     os.environ["HTTP_USER_AGENT"], comments))
cursor.execute("COMMIT")
```

C'est au client HTTP d'envoyer de l'UTF-8 dans l'URL, l'en-tête HTTP `Content-type` ne couvre que le corps, pas l'URL. Dans le cas du formulaire, le client HTTP doit normalement le faire spontanément dans le même encodage que la page contenant le formulaire. Si le client ne respecte pas cette convention, l'application plante avec :

```
UnicodeDecodeError: 'utf8' codec can't decode byte 0xe9 in position 3:
unexpected end of data
```

Enfin, pour afficher les caractéristiques de la chaîne de caractères (nom, point de code et catégorie des caractères), nous utilisons le module Python `unicode data` (<http://docs.python.org/lib/module-unicodedata.html>):

```
for char in word:
    ochar = ("U+%06x" % ord(char)).upper()
    result.append("%s %s (%s)" % (ochar,
        unicodedata.name(char,
            u"Unknown character"),
        unicodedata.category(char)))
```