Programmation réseau avec REST

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 13 mars 2006. Dernière mise à jour le 8 novembre 2006

https://www.bortzmeyer.org/programmation-rest.html

Depuis longtemps, plusieurs techniques ont été développées pour faciliter la tâche des programmeurs qui développent des applications fonctionnant en réseau. La mode actuelle est REST et cet article est l'occasion de faire un peu de REST.

Dans le monde de l'informatique, où l'ignorance est souvent bien partagée et où "nouveau" veut dire "à la mode depuis une semaine", la programmation en réseau a déjà vu passer plusieurs modes fortement poussées par le marketing. Par exemple, Corba, dont l'argument favori des promoteurs était que Corba permettait à des programmes écrits dans des langages de programmation différents de communiquer, comme si cela ne s'était pas fait depuis les débuts de l'informatique.

Aujourd'hui, les techniques les plus à la mode sont souvent présentées sous le parapluie de "*Web Services*" même si personne ne sait trop ce que cela veut dire. Sous ce parapluie, on trouve, XML-RPC, SOAP et REST.

XML-RPC est particulièrement simple et permet au débutant d'écrire des programmes en quelques minutes après avoir commencé le tutoriel. Pour moi, ses principales limites sont le fait que la norme n'est pas contrôlée par un groupe mais par une entreprise privée qui ne la fait pas évoluer.

SOAP est le seul protocole connu par les lecteurs de 01 (ou autres magazines pour décideurs). Il bénéficie du meilleur marketing mais est très complexe et l'interopérabilité est faible, comme l'a récemment rappelé un séminaire du W3C <http://xmlfr.org/actualites/tech/060306-0001>.

REST ("*Representational State Transfer*"), le sujet de cet article, n'est pas un protocole ou un format, c'est une architecture, c'est même l'architecture originale du Web, bâtie sur quelques principes simples :

— L'URI est important : connaitre l'URI doit suffire pour accéder à la ressource (pour pas mal de sites Web, notamment commerciaux, l'URI est inutilisable, parce qu'ils utilisent des "*frames*" ou bien parce qu'il y a un état caché ; on ne peut donc pas marquer une page donnée, envoyer son URI à d'autres, etc).

— HTTP fournit toutes les opérations nécessaires (GET, PUT et DELETE, essentiellement).
— Chaque opération est auto-suffisante : il n'y a pas d'état.
— En outre, et c'est un point où le sens moderne de REST dévie de l'architecture originale du Web, les requêtes et les réponses sont aujourd'hui typiquement encodées en XML. (Notons que le support du XML par les navigateurs Web est très variable mais le client REST n'est en général pas un navigateur.)

Plusieurs gros services sont aujourd'hui accessibles en REST comme Amazon, Yahoo ou del.icio.us. Un excellent exposé à Xtech donnait bien d'autres exemples, en comparant leurs API et bien d'autres aspects ("*Connecting Social Content Services using FOAF, RDF and REST*" <http://www.idealliance.org/proceedings/xtech05/papers/02-07-04/>).

Pour illustrer une application REST typique, prenons l'exemple d'un registre de noms de domaine. Il doit permettre de :

— Obtenir de l'information sur un domaine (cela se fait historiquement par le protocole whois, spécifié dans le RFC 3912 [1] mais, aujourd'hui, cela se fait plus logiquement en REST).
— Ajouter un nouveau nom de domaine.
— Supprimer un domaine.

Ces services se mettent facilement en correspondance avec les opérations de HTTP, décrites dans le RFC 2616 :

— On va obtenir de l'information par une opération GET (l'opération par défaut pour un navigateur).
— On va créer un domaine avec PUT.
— On va le détruire avec DELETE.

L'URL inclut logiquement le nom de domaine (ci-dessous, le `foobar.fr`).

On choisit de fournir les paramètres, s'il y en a, et d'envoyer les réponses en XML. Les requêtes et les réponses sont vérifiées par rapport à un schéma écrit en RelaxNG.

Le client est écrit en Python, comme le serveur mais on peut aussi, et c'est la beauté de REST, utiliser comme client un client HTTP ordinaire, par exemple curl :

```
% curl --include -X PUT --data '<domain><holder>AR41-FXNIC</holder></domain>' http://localhost:8080/foobar.f
HTTP/1.0 201 Created
Server: BaseHTTP/0.3 Python/2.3.5
Date: Mon, 13 Mar 2006 13:14:07 GMT

                    <domain>
                         <holder>AR41-FXNIC</holder>
                         <created>2006-03-13T13:14:07</created>
                         <creator>127.0.0.1</creator>
                    </domain>
```

Voici les fichiers qui composent l'application :

— Le code du serveur (en ligne sur `https://www.bortzmeyer.org/files/REST-server-xml.py`). Basé sur la bibliothèque BaseHTTPServer <http://docs.python.org/lib/module-BaseHTTPServ html>. Il utilise ElementTree <http://effbot.org/zone/element-index.htm>, pour analyser le XML. ElementTree fournit une interface plus simple et plus Pythonesque que DOM. La liste des noms de domaine est simplement gardée en mémoire dans un dictionnaire.

1. Pour voir le RFC de numéro NNN, `https://www.ietf.org/rfc/rfcNNN.txt`, par exemple `https://www.ietf.org/rfc/rfc3912.txt`

— Le code du client (en ligne sur `https://www.bortzmeyer.org/files/REST-client-xml.py`) mais, on l'a vu, un utilitaire générique comme curl peut convenir pour des requêtes comme `curl -X DELETE http://localhost:8080/machin.fx`.
— Le schéma général (en ligne sur `https://www.bortzmeyer.org/files/REST-common.rnc`), sa déclinaison pour les requêtes (en ligne sur `https://www.bortzmeyer.org/files/REST-schema-request.rnc`) et pour les réponses (en ligne sur `https://www.bortzmeyer.org/files/REST-schema-reply.rnc`).
— Le module de validation RelaxNG (en ligne sur `https://www.bortzmeyer.org/files/RelaxNGCheck.py`). Il utilise rnv <http://www.davidashen.net/rnv.html> car le support RelaxNG dans la bibliothèque 4suite <http://www.4suite.org> est très limité <http://xmlfr.org/listes/xml-tech/2006/03/0075.html>.

Une autre application REST, plus complète mais uniquement pédagogique, est publiée en <https://www.bortzmeyer.org/rest-sql-unicode-exemple.html>.

Dans le service ci-dessus, les paramètres en entrée étaient passés sous forme de texte XML. Mais on peut aussi les passer en paramètres HTTP classiques, dans l'URL ou bien dans les données envoyées par la commande POST. Par exemple, le service "*Web Search*" <http://developer.yahoo.com/search/web/V1/webSearch.html> de Yahoo est accessible ainsi :

```
% curl 'http://search.yahooapis.com/WebSearchService/V1/webSearch?appid=GETYOUROWNAPPID&query=Mycobacterium'

<ResultSet xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="urn:yahoo:srch"  totalResultsAvailable="
<Result><Title>Mycobacterium -
MicrobeWiki</Title><Summary>Mycobacterium. From MicrobeWiki, the
student-edited ... Deciphering the biology of Mycobacterium
tuberculosis from the complete genome
sequence.
...</Summary><Url>http://microbewiki.kenyon.edu/index.php/Mycobacterium</Url>
```

Par défaut, Yahoo renvoie du XML mais on peut demander du JSON ou d'autres formats. Voici un petit programme Python qui interroge Yahoo avec les requêtes données sur la ligne de commande en limitant la recherche au site <http://www.afnic.fr/> :

```
# http://sourceforge.net/projects/json-py/
import json

import sys
import urllib
import urllib2

application_id = "GETYOUROWNAPPID"
site = "www.afnic.fr"
myencoding="latin-1"
number_to_retrieve = 10

url = "http://search.yahooapis.com/WebSearchService/V1/webSearch?appid=%s&site=%s&region=fr&adult_ok=1&output=js

def tr(text):
        return text.encode(myencoding, 'replace')

for query in sys.argv[1:]:
    reply = urllib2.urlopen(url % (application_id, site, number_to_retrieve,
                                   urllib.quote(query.
                                                decode(myencoding).encode("UTF-8"))))
    results = json.read(reply.read())["ResultSet"]
    print "\"%s\": %s result(s)" % (query,
                                    results["totalResultsAvailable"])
    for result in results["Result"]:
        print "\t%s: %s" % (tr(result["Title"]), result["Url"])
    print
```

Un autre exemple de client REST se trouve dans mon programme de signalement à Signal-Spam <https://www.bortzmeyer.org/signaler-a-signal-spam.html>.

Un dernier exemple est le service Vélib' où la présence de vélos dans les stations, ou bien d'emplacements libres pour y raccrocher un vélo, peuvent être déouverts via un service REST <https://www.bortzmeyer.org/velib-rest.html>.

Un bon article d'Eric van der Vlist synthétisant REST <http://dyomedea.com/papers/2004-wsc/2-rest.html>.