Technologies pour Web Services faciles : REST, JSON

Pierre Gambarotto
INPT DSI, ENSEEIHT Département Informatique
ENSEEIHT, 2 rue Charles Camichel, B.P. 7122 31071 Toulouse CEDEX7 FRANCE

Résumé

Cet article peut se voir comme une suite du tutoriel sur les Web Services des JRES 2003, qui présentait les Web Services et les technologies XML-RPC et SOAP. Stéphane Bortzmeyer signalait à l'époque la complexité de déploiement et d'utilisation de ces technologies. L'histoire lui a donné raison si l'on considère par exemple que Google et Yahoo!, acteurs majeurs du domaine, ont tous deux abandonné leur Web Service SOAP au profit d'une technologie plus simple à mettre en œuvre par leur client.

REST est un style d'architecture réseau pour Web Services qui met l'accent sur la définition de ressources identifiées par des URI, et utilise les messages du protocole HTTP pour définir la sémantique de la communication client/serveur: GET pour le rapatriement d'une ressource, POST pour une création, PUT pour une modification/création, DELETE pour un effacement.

La représentation des ressources est libre, utilisant différents formats de représentation qui sont listés ici. L'article détaille plus amplement JSON (JavaScript Object Notation), format particulièrement adapté pour des applications web utilisant Ajax pour l'interrogation d'un Web Service.

L'article évoque également les technologies et techniques de développement connexes à l'utilisation de Web Services REST, telles que les librairies objets d'abstraction à des bases de données et l'utilisation d'AJAX dans les interfaces d'applications web.

Mots clefs

Web Services, REST, JSON, AJAX, ORM

1 Web Services

Un Web Service est un programme informatique reposant sur une architecture réseau client serveur. La spécificité des Web Services est l'utilisation de HTTP comme support des messages entre clients et serveur. Un Web Service permet donc d'ouvrir sur le réseau une application, la modalité d'accès reposant in fine sur une protocole dont les messages seront transportés par HTTP.

Présenté comme cela, les différences entre un Web *S*ervice et un site web ne semble pas évidentes...

L'idée sous-jacente est qu'un Web Service va pouvoir être facilement manipulé par une interface de programmation (API), alors qu'un simple site web par opposition est fait pour être utilisé par un être humain.

Les avantages architecturaux sont identiques à des architectures logicielles telles que Corba, RMI (Java) ou DCOM (Microsoft), et permettent à des composants logiciels écrits dans divers langages et tournant sur des machines différentes de communiquer.

Les autres avantages sont liés à l'utilisation d'HTTP pour le transport:

- port 80 généralement ouvert, facile à intégrer dans un réseau que ce soit en interne ou sur internet.
- HTTP : protocole bien connu et simple à aborder (protocole texte)

L'historique des Web Services est assez simple à retracer. Cela a commencé avec des applications web et des analyseurs syntaxiques de code HTML, puis un encodage spécifique pour des appels de procédures distantes (RPC), XML-RPC et son évolution SOAP (1998).

L'écosystème autour de SOAP est rapidement devenu touffu, le nombre et la complexité des spécifications WS-* étant là pour en attester : WSDL pour la description des Web Services, SOAP et ses nombreuses extensions pour la définition des messages, WS-Security pour l'aspect sécurité, UDDI pour le référencement de Web Services pour n'en citer qu'une infime partie. L'objectif ambitieux est de définir des architectures orientées autour de services logiciels interconnectés (SOA pour Service-Oriented Architecture). Une SOA compte typiquement un annuaire de services, chaque service y est décrit en utilisant WSDL,

et l'utilisation de chaque service consiste à coder les messages du service en SOAP, lui même encapsulé dans HTTP pour le transport. Une bonne description de ces technologies se trouve dans [1].

Inconvénients de ces premiers modèles de Web Services :

- HTTP est utilisé uniquement comme moyen de transport. Les seuls messages utilisés de HTTP sont GET et POST. Chaque Web Service dispose d'une interface spécifique, encapsulé directement dans HTTP (pour le cas simple des XML-RPC) ou dans SOAP, et dans ce dernier cas décrite en XML par le langage WSDL.

- architecture SOA : la mise en œuvre réelle est complexe, les normes volumineuses et difficiles à véritablement maîtriser. L'investissement du développeur est assez important. Les outils masquent la complexité d'utilisation mais limitent l'utilisation à un même cercle d'utilisateurs - typiquement le monde entreprise (Java, .Net). De plus, le fait de pouvoir par un simple clic exposer une fonctionnalité interne à travers un Web Service tend à imposer le style RPC même pour les services utilisant les normes WS-* censés apporter une architecture orientée service (SOA).

Cela explique que des têtes de ponts telles que Google et Yahoo! ont effectué la transition de service SOAP vers des services REST.

Le principe de REST est d'utiliser HTTP pour l'implémentation d'un Web Service, non plus seulement comme simple protocole de transport, mais également pour définir l'API de chaque service, c'est à dire la définition même des messages entre clients et serveur. Paradoxalement, c'est donc un retour aux sources vu que la spécification de HTTP 1.1, la dernière version en date, est légèrement antérieure (1997) aux premiers Web Services basés sur les RPC (1998).

Avantages de REST:

- HTTP protocole applicatif et non protocole de transport
- cache réseau : en respectant les entêtes et les requêtes préconisés dans la norme HTTP, on permet ainsi l'utilisation efficace de serveur cache entre le serveur et les clients de l'application.
- interface uniforme: chaque Web Service REST a une interface orientée autour des messages de HTTP.

Exemple d'utilisation basique : le Web Service Google Search, utilisée par le client curl en ligne de commande, renvoie un document JSON listant les résultats de recherche :

```
curl -e http://karma.enseeiht.fr 'http://ajax.googleapis.com/ajax/services/search/web?v=1.0&q=REST
+jres2009+gambarotto'
{"responseData": {
 "results": [
  {
   "GsearchResultClass":"GwebSearch",
   "unescapedUrl":"https://2009.jres.org/planning",
   "url":"https://2009.jres.org/planning",
   "visibleUrl":"2009.jres.org",
   "cacheUrl":"http://www.google.com/search?q\u003dcache:QeeMRpNPfOkJ:2009.jres.org",
   "title":"Retour - planning [Les Journées Réseaux 2009]",
   "titleNoFormatting":"Retour - planning [Les Journées Réseaux 2009]",
   "content":"18 juin 2009 \u003cb\u003e...\u003c/b\u003e Programme provisoire des..."
  },
  {
   "GsearchResultClass":"GwebSearch",
   ...
  ],
 "cursor":{
  "pages":[{"start":"0","label":1}],
  "estimatedResultCount":"2",
  "currentPageIndex":0},
```

```
"responseDetails": null, "responseStatus": 200}
```

2 REST

REST est l'acronyme de Representational State Transfer. REST décrit un style d'architecture logicielle permettant de construire une application devant fonctionner sur des systèmes distribués, typiquement internet. Si cela vous évoque un tant soit peu le Web (World Wide Web), rien d'étonnant vu que l'auteur de cette description n'est autre que Roy Fielding dans sa thèse [2], un des principaux rédacteurs de la norme HTTP 1.1. (voir RFC 2616 [3]). En résumé, REST est donc le style d'architecture soutenant le Web.

2.1 Principes directeurs

- architecture client serveur
- sans état : 2 requêtes d'un client sont indépendantes, ce qui veut dire qu'au niveau serveur on ne traite pas une requête en référençant des éléments d'une requête précédente. Au niveau client, tout ce qui est nécessaire au traitement de la requête doit être inclus dans celle-ci. Au niveau HTTP, cela veut dire que l'on ne crée pas de session utilisateur dans laquelle on stocke des informations.
- utilisation de mécanismes de cache possible, système en couche : l'idée est de pouvoir bénéficier du système distribuée sous-jacent en permettant la mise en place de cache à chaque étape entre le client et le serveur. Pour HTTP, cela consiste essentiellement en l'utilisation de serveur proxy.
- interface uniforme : c'est le point principal de différence par rapport aux Web Services précédents: tout élément offert à la manipulation par l'application est nommé ressource et est identifié de manière unique. HTTP définit les Identifiants de Ressource Uniforme (URI ci-après) suivant le schéma :

```
http_URL = "http:" "//" host [ ":" port ] [ abs_path [ "?" query ]]
```

Les différentes actions possibles sur ces ressources sont données par les différents types de requêtes HTTP, principalement GET, POST, PUT et DELETE.

On manipule des représentations des ressources, par les ressources directement. Les ressources sont donc encodées selon un format spécifique dans les messages HTTP.

Au lieu d'être incluse dans un message, une ressource peut être référencé par un hyperlien.

REST n'est pas un protocole, il n'existe donc pas de normes en tant que telle, mais plutôt des conventions de codage respectant les principes cités ci-dessus. Pour un protocole applicatif respectant ces principes on parlera d'implémentation RESTful, et comme exemple de réalisation, un Web Service utilisant HTTP sur ces principes sera également qualifié de RESTful.

Exemple : une illustration complète est constituée par le protocole de publication Atom (APP, voir [5]), qui permet l'édition de ressources web,typiquement utilisé dans la syndication de sites web.

WDSL dans sa norme 2.0 (le langage qui sert à décrire une API SOAP) permet d'utiliser tous les messages de HTTP et donc de programmer des Web Services RESTful, tout en gardant la description des messages du protocole en XML, pour ceux qui veulent rester compatible avec les normes du W3C.

2.2 Principes d'implémentation

Pour définir une API REST, les étapes suivantes doivent être suivies :

- définition des ressources manipulées, collection de ressources (liste) ou ressource unique.
- codage de la représentation des ressources : quels sont les attributs d'une ressource, quel format va-t-on utiliser ?
- sémantique des messages : les actions possibles sur les ressources sont indiquées par les messages du protocole de transport, ce qui donne pour HTTP :
    - GET : récupération de la représentation d'une ressource ou d'une liste de ressource.
    - PUT : mise à jour d'une ressource existante, création d'une ressource en spécifiant l'URI de la ressource.
    - POST : création d'une sous ressource (le serveur décide de l'URI), ajout d'information à une ressource existante.
    - DELETE : effacement.
    - HEAD : informations sur une ressource.

  Une ressource donnée ne sera pas obligatoirement manipulable par tous les messages. Par exemple, une ressource accessible en lecture seulement peut n'être accessible que par les messages de type GET.

2.3 Génération d'un message

Pour générer un message, il faut suivre les étapes suivantes :

– définir la ou les ressources visées (URI collection, URI member)
– requête : définir l'action demandée, et donc le type de message : GET, PUT, POST, DELETE
– réponse: décider du code d'erreur
– code de la représentation de la ressource : ce codage se prête bien à une programmation orientée objet et à l'utilisation de bibliothèques de sérialisation/désérialisation

En suivant ces principes cela donne :

– Pour le serveur, décodage d'une requête (message issu d'un client) :
  - identifier la ressource visée
  - identifier l'action demandée par le type de requête HTTP
  - en fonction de la requête, procéder à l'analyse de la représentation des ressources fournies (PUT et POST) dans la requête
  - en fonction des 2 étapes précédente, procéder à la résolution de l'action demandée
  - générer la réponse : représentation de la ressource incluse dans la réponse, génération du code résultat

Exemple : réception de la requête HTTP:

```
GET /users/toto HTTP/1.1

Accept: text/xml
```

La ressource visée est `/users/toto`. L'action demandée est GET. L'encodage demandé est XML.

– Pour le client, décodage d'une réponse HTTP (message en provenance du serveur) :
  - décodage du code résultat
  - en fonction du résultat, analyse de la représentation de la ressource incluse dans la réponse

Les principes de programmation d'une application RESTful sont donc très proches d'une manipulation basique de HTTP ce qui rend relativement aisée une réalisation que ce soit dans le cadre d'une application existante ou dans le cadre d'un nouveau développement.

3 Représentation des ressources : JSON

Que ce soit au niveau du serveur ou au niveau des clients, une API RESTful manipule des ressources par une représentation de celles-ci.

Le principe en pratique est de passer de la représentation utilisée en interne, que ce soit sur le client ou le serveur, à la représentation utilisée dans le message HTTP, requête ou réponse.

Concernant la spécification du format de représentation utilisé dans le message : comme spécifié plus haut, les messages d'une application RESTful sont indépendants les uns des autres ce qui implique en particulier que le codage de la représentation des ressources peut être différent entre deux messages et donc qu'il faut spécifier dans le message le codage utilisé.

Pour un Web Service RESTful on utilise typiquement un entête HTTP :

```
Content-type: text/xml
```

Le client doit demander un format d'encodage spécifique en utilisant l'entête Accept :

```
Accept: text/xml
```

Pour les Web Services RESTful, les encodages les plus utilisés sont XML et JSON.

XML est un standard incontesté mais souffre de quelques inconvénients:

– verbeux
– difficilement lisible par un humain

– dualité entre les attributs et les éléments.

JSON est l'acronyme de JavaScript Object Notation. C'est un format texte qui permet de représenter des données et de les échanger facilement à l'instar d'XML.

JSON est un sous ensemble d'ECMAScript (JavaScript) et est décrit dans le RFC 4627 [4].

Ce sous ensemble de JavaScript permet de décrire le modèle objet de JavaScript. Deux types de structures sont disponibles :

– Objet : une collection de paire nom/valeur, i.e un tableau associatif.

– Tableau : une liste ordonnée de valeurs.

Les valeurs peuvent être des types suivants : booléen, chaîne de caractères, nombre, ou valeur nulle. (boolean, string, number, null), ou une des structures ci-dessus.

La syntaxe est celle de JavaScript, simpliste. Voici par exemple la description d'un utilisateur avec une adresse et des numéros de téléphone :

```
{
 "nom":"Bob",
 "age":34,
 "addresse": { "rue":"avenue Grande", "ville":"Rio", "code":86945},
  "telephone":[  {"type":"maison",  "numero"  :  123456},  {"type":"portable",
"numero": 654321} ]
}
```

L'équivalent en XML serait :

```
<utilisateur nom="Bob" age="34">
  <adresse>
    <rue>avenue Grande</rue>
    <ville>Rio</ville>
    <code>86945</code>
  </adresse>
  <telephones>
    <telephone type="maison">123456</telephone>
    <telephone type="portable">654321</telephone>
  </telephones>
</utilisateur>
```

En comparaison, le format JSON ne fait pas de différence entre attribut et élément comme en XML. Il est donc moins sujet à variation, et est globalement plus concis et plus lisible. Les valeurs sont typées alors qu'en XML on ne dispose que de chaînes de caractères.

Au delà de la lisibilité, l'avantage majeur est de pouvoir facilement intégrer des objets JSON dans une application JavaScript. Par exemple si `user` désigne l'utilisateur représenté ci-dessus :

```
var bob = eval("(" + user + ")");
```

Les butineurs web récents comme Firefox 3.5 disposent d'un analyseur syntaxique JSON intégré qui permettent moins de dérive que l'évaluation brute par eval:

```
var bob = JSON.parse(user);
```

L'utilisation d'une bibliothèque JavaScript permettant d'enrober les différences de gestion entres les navigateurs est recommandée pour éviter des problèmes de portabilité.

4 API REST: exemple détaillé

Pour illustrer les principes de création d'une API s'appuyant sur les principes REST, je vais décrire une gestion d'utilisateurs très générale, telle qu'on la retrouve souvent dès qu'une application a besoin de gérer une authentification de ses utilisateurs.

Les utilisateurs sont identifiés en tant que ressources par un schéma d'URI du type : `http://server/user/id`, où `id` est une clef permettant d'identifier de manière unique la ressource, par exemple le login pour un utilisateur. L'ensemble des utilisateurs est référencé par le schéma d'URI `http://server/users` (notez bien le pluriel). Les différentes actions possibles sur ces 2 types de ressource sont :

URI	Sémantique	Code réponse
GET http://server/users	Récupère la liste des utilisateurs	200 OK
POST http://server/users	Création d'un nouvel utilisateur	201 Created
GET http://server/user/id_user	Récupère la représentation de l'utilisateur identifié par id_user	200 Ok, 404 resource not found
PUT http://server/user/id_user	Modifie un utilisateur	200 Ok, 404 ressource not found
DELETE http://server/user/id_user	Efface un utilisateur	200 Ok, 404 ressource not found

Les deux premiers schémas de requêtes adressent la collection entière des utilisateurs, tandis que les suivants visent un membre particulier de la collection. Ce sont deux types de ressources différents, c'est donc normal que le schéma d'URI soit spécifique à chacun.

Dans une API REST on retrouve presque systématiquement cette dualité collection/membre. Notez que http://server/user (au singulier) aurait pu être l'URI identifiant la collection. C'est à la partie serveur de réaliser l'analyse des requêtes pour déterminer quelles sont les ressources visées par une requête particulière.

Représentations des ressources : JSON est utilisé pour représenter les deux types de ressources, utilisateur et liste d'utilisateurs.

– Utilisateur :

```
{user:{login: id_user, password: secret }}
```

– Liste :

```
[{user:{login: id1, password: secret1}}, {user:{login:id2, password:secret2}}]
```

Un utilisateur est donc ici représenté par un login et un password.

La 3ième colonne du tableau liste les codes de réponses HTTP utilisés.

5 Écosystème

Dans les deux sections précédentes, nous avons présenté les principes de base d'une application RESTful.Intéressons nous maintenant aux techniques et aux composants logiciels fréquemment utilisés dans la conception d'un Web Service RESTful.

5.1 HTTP

En premier lieu, et c'est la caractéristique principale de l'architecture REST, HTTP joue un rôle central, pas seulement dans le nommage des ressources (URI) et la sémantique des messages entre clients et serveur.

HTTP est également utilisé pour l'authentification quand elle est nécessaire, typiquement par l'utilisation des mécanismes intégrés (HTTP Basic) en plus de SSL. L'alternative constatée est d'utiliser un entête HTTP particulier et un mécanisme d'attribution de clef que l'utilisateur du Web Service doit rappeler systématiquement dans l'entête. Dans tous les cas, on utilise au maximum les codes et entêtes HTTP standards (entête Authentication, code de réponse 401 …)

Pour optimiser le trafic réseau entre les clients et le serveur, on peut également profiter des capacités de cache que permet HTTP.

Au niveau serveur, il est possible de générer l'entête Etag permettant de représenter la version d'une entité. On peut par exemple prendre le hash MD5 de la représentation de la ressource, ou un *timestamp* en provenance d'une base de données.

Lors de la première requête GET sur une ressource désignée par une URI, on récupère l'Etag positionné par le serveur.

On peut ensuite effectuer un GET conditionnel en rappelant cette valeur avec un des entêtes `If-Match`, `If-None-Match`, `If-Range`. Certains serveurs proxy tels que Squid peuvent utiliser les Etags pour constituer un cache.

L'alternative est d'utiliser la date de dernière modification, là aussi positionnée par le serveur. Le principe est identique: le serveur positionne la date de dernière modification par l'entête `Last-Modified`. Le client peut alors réutiliser cette date en effectuant des GET conditionnel avec les entêtes `If-Modified-Since` et `If-Unmodifed-Since`.

Dans les 2 cas, si le serveur reçoit un GET conditionnel et détecte que la ressource n'a pas été modifiée, il doit alors renvoyer le code réponse : 304 Not Modified.

5.2 Ajax

Ajax repose sur l'utilisation de la fonction JavaScript `XMLHTTPRequest`, qui permet à un navigateur web de générer une requête HTTP à partir d'un événement JavaScript, et ce de manière asynchrone. Cela permet de créer des interfaces web qui interrogent le côté serveur tout en continuant à répondre aux sollicitations de l'utilisateur, se rapprochant des interfaces classiques.

6 Cas concrets d'utilisation

6.1 Ouverture d'une application existante

Pour récupérer les inscriptions des étudiants dans Apogee, j'ai créé un Web Service REST très simple, ouvert uniquement en lecture, en suivant les principes suivants :

- définition des ressources par équivalences avec des requêtes SQL : étudiant en provenance de la table « individu », inscription à partir de la table « ins_adm_etp ».
- utilisation d'une bibliothèque pour le passage base de données ↔ représentation objet.
- utilisation d'une bibliothèque de sérialisation JSON pour le passage objet ↔ représentation JSON.

Au niveau algorithmique, la séquence des procédures appelées lors de la réception d'une requête GET au niveau serveur est la suivante : si la requête reçue ressemble à `GET /student/12345`

- analyse de l'URI pour déterminer la ressource visée, ici l'étudiant 12345.
- obtention de la représentation objet de l'étudiant 12345 à partir de la base de données Apogee par utilisation d'une bibliothèque ORM (*Object Relationnal Mapping*) qui gère l'état d'un objet représentant une ligne dans une table d'une base de données en générant le SQL adéquat.
- utilisation d'une bibliothèque de sérialisation en JSON ou en XML.

A partir du moment où l'on connait la structure de la base de données d'une application existante, il est alors possible avec peu de code d'offrir en lecture l'accès aux données par un Web Service REST et ce de manière assez systématique.

6.2 Utilisation d'un Web Service dans une page web par appel JavaScript

J'utilise ici la bibliothèque jQuery pour enrober l'appel à `XMLHttpRequest.` Ce fragment de page HTML comprend un formulaire décrivant un utilisateur, et la fonction JavaScript déclenchée lors de la validation du formulaire.

```
<script>
$(document).ready(function() {
  $('form.edit_user').bind('submit', function() {
    $.ajax({
      type: 'PUT',                    // type de requête HTTP
      url: $(this).attr('action'),    // this = formulaire
      dataType: 'json',               // format d'encodage des données
      data: $(this).serialize(),      // data = serialisation du formulaire
      processData : false,
```

```
      success: function(msg) {
        alert( "Data Saved: " + msg );
      },
      error: function (XMLHttpRequest, textStatus, errorThrown) {
        alert("Ooooops!, request failed with status: " +
              XMLHttpRequest.status + ' ' + XMLHttpRequest.responseText);
      }
    });
    return false;
  });
});
</script>

<form action="/users/1" class="edit_user" method="post">
 <input id="user_login" name="user[login]" size="30" type="text" value="login" />
 <input id="user_password" name="user[password]" size="30" type="text" value="secret"/>
 <input id="user_submit" name="commit" type="submit" value="Update" />
</form>
```

La validation du formulaire déclenche donc l'émission d'une requête PUT à destination de l'URI déterminée par le paramètre `action` du formulaire.

7 Conclusion

Un Web Service RESTful est très simple à utiliser (partie cliente) et relativement simple à écrire (partie serveur) à partir du moment où l'on fait appel à des bibliothèques existantes. Le seul prérequis est la connaissance de HTTP.

Ce type d'architecture connaît un grand succès en ce moment, confirmé par l'intérêt d'acteurs tels que Google, Yahoo! ou encore Amazon. Une des raisons principales tient à l'importance croissante des navigateurs en tant que plateforme d'exécution sur les postes clients fixes mais également sur les terminaux mobiles. Les applications web sont en train de passer d'un mode « document » (le serveur retourne une mise en forme du résultat) à un mode desktop (le client charge la partie interface de l'application, les requêtes au serveur ne font plus que charger les données) essentiellement grâce à l'utilisation d'Ajax. Dans ce contexte, fournir une interface RESTful à une application permet d'envisager une intégration relativement aisée. Comme le coût technique est peu élevé je pense qu'il serait donc sage, que ce dans nos développements internes ou comme fonctionnalité à inclure lors d'un appel d'offres, nous nous assurions de la présence d'une interface REST.

L'investissement pour un développeur est relativement minime, notamment s'il possède déjà une expérience en programmation d'applications web assez bas niveau, proche de HTTP. Il existe pour le moment peu de ressources livresques sur le sujet si l'on excepte [6]. Par contre, de nombreuses ressources décrivant des API existantes sont disponibles par une recherche web notamment sur les sites de Google (http://code.google.com) et Yahoo (http://developer.yahoo.com).

Bibliographie

1: Stéphane Borzmeyer, Les Web services : connecter des applications, 2003

2: Fielding, Roy Thomas, Architectural Styles and the Design of Network-based Software Architectures, 2000

3: Fielding and al, Hypertext Transfer Protocol -- HTTP/1.1, 1999

5: J. Gregorio, B. de hOra, The Atom Publishing Protocol, 2007

4: D. Crockford, The application/json Media Type for JavaScript Object Notation (JSON), 2006

6: Leonard Richardson, Sam Ruby, RESTful Web Services, 2007