Web des services : REST

author: Pierre-Antoine Champin

Date: 2011-2012

Sommaire

1 Introduction

REST : une alternative à SOAP

SOAP : *Simple Object Access Protocol*

- Une recommandation du W3C (2003) pour l'échange de données entre services Web (cf. séance précédente).
- Poussée par des membres influents du consortium (Microsoft, Sun, IBM, Canon, Oracle).

Pourtant, SOAP recontre une certaine résistance

- REST (*Representational State Transfer*), notion proposée en 2000 par Roy Fielding, est proposé comme une alternative.
- En 2006, Google abandonne son API SOAP au profit d'une API simplifiée REST-like.

SOAP: Pas si « simple »...

Une spécification volumineuse

- Son ampleur décourage certains implémenteurs : pas d'implémentation complète dans certains langages (PHP, Python...).
- Ce phénomène est amplifié par une profusion d'*extensions* (WS-Policy, WS-Security, WS-Trust, etc.)...
- ... et le fait que leurs implémentations ne sont pas toujours homogènes.

SOAP est parfois perçu comme *limitant* paradoxalement l'interopérabilité des services Web.

SOAP: Pas si « Web »...

Le succès du Web doit beaucoup à sa *simplicité* de mise en œuvre, simplicité qui fait défaut à SOAP.

- cf. transparent précédent

Avec SOAP, le prototole HTTP est utilisé comme un simple protocole de transport.

- C'est un choix délibéré (indépendance des couches, modèle OSI).
- Pourtant, HTTP a été conçu comme un protocole *applicatif*.

→ SOAP se prive d'une grande partie des mécanismes du Web

2 REST : le style architectural du Web

- **Style architectural**: notion proposée par Roy Fielding dans sa thèse (2000) : *Architectural Styles and the Design of Network-based Software Architectures*.
- Défini comme un ensemble de contraintes imposées sur la conception d'un système pour garantir un certain nombre de propriétés.
- La question sous-jacente était d'identifier les contraintes à respecter dans le développement du Web :
  - pour conserver les propriétés ayant fait son succès,
  - pour corriger les problèmes constatés,
  - pour évaluer les évolutions futures des technologies.

Objectifs

« REST emphasizes

- scalability of component interactions,
- generality of interfaces,
- independent deployment of components, and
- intermediary components to
  - reduce interaction latency,
  - enforce security, and
  - encapsulate legacy systems. »

(Roy Fielding)

Passage à l'échelle

Problème de performance

La dimension et l'expansion constante du Web obligent à considérer dès le départ les problèmes d'échelle.

Problème de robustesse

le système ne peut pas être 100% opérationnel 100% du temps.

Généralité des interfaces

Facilite le développement des composants

Réutilisation de bibliothèques standard, interopérabilité.

Évolutivité

Web de documents, Web de services, Web 2.0, Web des données, Web sémantique...

Principle of partial understanding (Michael Hausenblas)

Déploiement indépendant des composants

Contrainte d'échelle

Pas de centralisation possible sur le Web.

Adoption et évolution rapides

Grassroot movement (poussé par le bas), « sélection naturelle », modèle du Bazar (Eric Raymond)

e.g. : Mosaic → Netscape, CERN httpd → Apache

Composants intermédiaires

Réduire la latence

- en conservant en cache le résultat de certaines requêtes (*proxy*),
- en répartissant la charge entre plusieurs serveurs redondants (*reverse proxy*).

Assurer la sécurité

- en masquant l'accès à certains serveurs (*proxy* filtrant, *reverse proxy*, n-tiers).

Encapsulant des services

- en adaptant les requêtes (*gateway*) ← généralité des interfaces

Contraintes

- client-serveur
- connexion sans état (*stateless*)
- support des caches
- interface uniforme
- système en couches
- code à la demande (optionel)

Principes

Resource

Une ressource est une entité *abstraite*.

→ Ce n'est *pas* un fichier.

Une ressource est identifiée par un URI (*Uniform Resource Identifier*).

→ Un URI/URL n'identifie *pas* un fichier !

Représentation

Une ressource a une ou plusieurs représentation(s).

→ négociation de contenu

Ces représentations peuvent varier dans le temps.

Les ressources sont toujours manipulées *via* leurs représentations.

Hypertexte généralisé

- La représentation d'une ressource comporte les *liens* vers d'autres ressources (identifiées par leurs URIs).
- La sémantique du lien dépend du *format* de la représentation.
- L'état d'un client change en *suivant* les liens découvert dans les représentations.
- NB : Une alternative consiste à *construire* un URI en fonction des informations fournies par une représentation. C'est le cas des formulaires HTML utilisant la méthode GET, e.g.:

```
http://www.google.fr/search?q=hypertexte
```

3 REST par l'exemple

HTTP

L'« implémentation de référence » de REST.

Avertissement

REST → HTTP mais

- HTTP est largement implémenté.
- Tout protocole RESTful serait amené à réinventer une grande partie de HTTP.

  > any sufficiently sophisticated program implements a buggy subset of half a Common Lisp interpreter

HTTP → REST

- HTTP 1.1 comporte des compromis (compatibilité avec l'existant)
- Il existe plusieurs manières d'utiliser HTTP, et toutes ne sont pas RESTful (e.g. SOAP).
- Certaines extensions de HTTP sont « RESTless » (e.g. cookies).

Exemple de requête HTTP

Requête à http://www.w3.org/

```
GET / HTTP/1.1
Host: www.w3.org
User-Agent: Mozilla/5.0 (X11; U; Linux i686;
       fr; rv:1.9.1) Gecko/20090624 Firefox/3.5
Accept: text/html,application/xhtml+xml,
       application/xml;q=0.9,*/*;q=0.8
Accept-Language: fr,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: UTF-8,*
Connection: keep-alive
Keep-Alive: 300
```

Exemple de réponse HTTP

Réponse de http://www.w3.org/

```
HTTP/1.x 200 OK
Date: Mon, 02 Nov 2009 22:46:26 GMT
Server: Apache/2
Accept-Ranges: bytes
Content-Type: text/html; charset=utf-8
Content-Length: 29794
Etag: "7462-477341dcfb940;89-3f26bd17a2f00"
Last-Modified: Sat, 31 Oct 2009 05:07:09 GMT
Content-Location: Home.html
Vary: negotiate,accept
Cache-Control: max-age=600
Expires: Mon, 02 Nov 2009 22:56:26 GMT
Connection: close

(data)                                                              ...
```

Sémantique de la requête et de la réponse

La requête consiste à appliquer un *verbe* à la ressource.

- GET signifie « obtenir une représentation de la ressource ».
- Le champs `Host` est obligatoire depuis HTTP/1.1 (car un même serveur peut servir plusieurs noms de domaine).

```
GET / HTTP/1.1
Host: www.w3.org
```

La réponse commence par un *code de statut* numérique indiquant le résultat .

- Le code est suivi d'un libellé textuel pour améliorer la lisibilité.

```
HTTP/1.x 200 OK
```

Verbes

HTTP définit quatre (principaux) verbes pour la manipulation des ressources :

- GET : pour obtenir une représentation de la ressource
- POST : pour créer une ressource subordonnée à la ressources cible
  (les formulaires HTML appliquent très rarement ce principe)
- PUT : pour modifier une ressource en fournissant sa nouvelle représentation
- DELETE : pour supprimer une ressource

Les requêtes POST et PUT supposent que l'envoi de données (*payload*) au serveur. Ces données peuvent être de n'importe quel type, qui sera spécifié dans la requête (`Content-Type`).

Codes de statut

HTTP définit 40 codes de statut, répartis en cinq catérogires :

Catégories	Exemples
1xx : Information	100 Continue
2xx : Succès	200 OK
3xx : Redirection	301 Moved Permanently
4xx : Erreur client	404 Not Found, 401 Unauthorized
5xx : Erreur serveur	500 Internal Server Error

Identification

Le client et le serveur donnent des indications sur leur identité et leur contexte.

- Utile notamment pour les terminaux mobiles.

```
User-Agent: Mozilla/5.0 (X11; U; Linux i686;
       fr; rv:1.9.1) Gecko/20090624 Firefox/3.5
```

```
Server: Apache/2
Date: Mon, 02 Nov 2009 22:46:26 GMT
```

Négociation de contenu (1)

Le client annonce les types de contenus qu'il est capable d'accepter.

```
Accept: text/html,application/xhtml+xml,
        application/xml;q=0.9,*/*;q=0.8
Accept-Language: fr,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: UTF-8,*
```

```
Content-Type: text/html; charset=utf-8
Content-Location: Home.html
Vary: negotiate,accept
```

Négociation de contenu (2)

- Si aucune représentation disponible ne peut satisfaire le client, le serveur peut retourner une code 406 *Not Acceptable*.
- Il doit normalement fournir dans la réponse une liste des représensentations disponibles.
- L'agent (logiciel client) est autorisé à sélectionner automatiquement l'une de ces représentations.
- Cette information peut également être fournie dans l'entête de la réponse (non standard).

En-tête Alternates de http://www.w3.org/

```
Alternates: {"Home.html" 1 {type text/html} {charset utf-8} {length 29813}},
{"Home.xhtml" 0.99 {type application/xhtml+xml} {charset utf-8} {length 29813}}
```

Méta-données de cache (1)

Le serveur fournit des méta-données relatives à la représentation.

```
Etag: "7462-477341dcfb940;89-3f26bd17a2f00"
Last-Modified: Sat, 31 Oct 2009 05:07:09 GMT
```

Ces méta-données font partie de l'état du client, et sont donc censées être fournie par lui lors de requêtes ultérieures (*statelessness*).

```
If-None-Match: "7462-477341dcfb940;89-3f26bd17a2f00"
If-Modified-Since: Sat, 31 Oct 2009 05:07:09 GMT
```

Si la ressource n'a pas été modifiée, le serveur répondra par un statut 304 *Not Modified* et une réponse vide → économie.

Méta-données de cache (2)

D'autres méta-données concernent la « cachabilité » de la représentation.

```
Cache-Control: max-age=600
Expires: Mon, 02 Nov 2009 22:56:26 GMT
```

`Cache-Control` (introduit dans HTTP 1.1) offre plus d'expressivité que `Expires` (`private`, `no-transform`...).

`Cache-Control` peut également être utilisé dans une *requête*, par exemple :

- pour spécifier sa tolérance à la « vieillesse » de la réponse,
- pour forcer un rechargement:

```
Cache-Control: no-cache
```

Méta-données de connexion

HTTP étant un protocole sans état, chaque requête peut être envoyée sur une nouvelle connexion TCP (ce qui était imposé par HTTP 1.0).

Pour des raisons d'optimisation, le client et le serveur peuvent spécifier qu'ils souhaitent / acceptent de garder la connexion ouverte pour des requêtes ultérieures. Ceci est fait explicitement pour conserver l'auto-suffisance des requêtes.

```
Connection: keep-alive
Keep-Alive: 300
```

```
Connection: close
```

Atom Publishing Protocol

Rappel: Atom Syndication Format (RFC 4287) définit un format de syndication par flux (similaire à RSS, aux Podcasts...) basé sur XML.

La RFC 5023 étend Atom en le munissant d'un protocole permettant la *mise à jour* des flux. Ce protocole constitue l'exemple type de protocole RESTful.

Un flux (`feed`) est maintenant considéré comme une *collection*; les membres de cette collections sont représentés par des entrées (`entry`).

Exemple d'Atom

```
<feed xmlns="http://www.w3.org/2005/Atom">
 <title type="text">PA Champin's blog</title>
 <updated>2009-11-10T20:03:29Z</updated>
 ...
 <entry>
  <id>tag:champin.net,2009:3.1415</id>
  <title>An atom example</title>
  <author><name>Pierre-Antoine Champin</name>...</author>
  <updated>2009-11-09T11:23:58Z</updated>
  <link href="http://champin.net/2009/11/09/an_atom_example"/>
  <content>This is an atom entry...</content>
  ...
 </entry>
 <entry>...</entry>
 ...
</feed>
```

Interface d'une collection

Une collection répond aux verbes GET et POST.

- GET retourne une description de la collection (Atom classique).
- POST sert à créer une nouvelle entrée ; les données envoyées sont une description de cette entrée (fichier XML dont la racine est `entry`).
  - En cas de succès, le statut de la réponse est `201 Created`, et l'en-tête contient un champs `Location` contenant l'URI de la ressource créée (pour permettre sa modification ultérieure).
  - Les données de la réponse sont une description en XML de l'entrée crée. Cette description être différente de celle soumise, car le serveur peut ajouter des méta-données (notament les balise `link`, cf. ci-après), voire même modifier celles fournies dans la requête (e.g. `id`).

Identification des ressources

Il est important de distinguer la sémantique de différentes URIs présents dans une entrée :

`<link href="URI"/>`

L'URI identifie une *description alternative* de la ressource membre (comme si l'attribut `rel="alternate"` était présent).

`<content type="text/html" src="URI">`

Ici, l'URI identifie le *contenu* externalisé de la ressouce membre.

`Location:` (http)

`<link rel="edit" href="URI"/>` (XML)

Cet URI (retourné par le POST ou rapellé dans le XML) est celui de la ressource elle-même (au sens de REST).

Interface d'une ressource membre

Une ressource membre répond aux verbes GET, PUT et DELETE.

- GET retourne une description de la ressource sous forme d'une `entry`.
- PUT sert à modifier la ressource en soumettant une nouvelle description.
- DELETE sert à supprimer cette ressource.

Gestion des contenus multimédia (1)

Atom autorise un flux à contenir des ressources non textuelles, en utilisant la balise `content` sous la forme suivante :

Media Resource

```
<entry>
  ...
  <content type="image/png"
   src="http://champin.net/img/photo123.png"/>
</entry>
```

On pourrait gérer ce type d'entrées comme les autres, mais il faudrait alors gérer séparément la ressource contenu. Le protocole Atom offre des mécanismes pour gérer de manière cohérente un contenu multimédia et l'entrée qui le décrit.

Gestion des contenus multimédia (2)

Les données d'une requête POST peuvent en fait être d'un type arbitraire, auquel cas :

- le serveur crée *deux* ressources membres :
  - une ressource multimédia contenant les données postées,
  - une ressource *media link entry*, décrite en XML, contenant les méta-données (extraites ou inférées) de la ressource précédente et dont le contenu (`content`) est cette ressource ;

Gestion des contenus multimédia (3)

- la réponse à la requête POST contient des informations (champs `Location`, données) portant sur la ressource *media link entry* ;
- la ressource *media link entry* peut contenir un lien permettant de modifier (PUT) le contenu multimedia, sous la forme :

```
<link rel="edit-media" src="URI"/>
```

- la suppression d'une des deux ressources entraînera automatiquement la suppression de l'autre.

Autres fonctionalités du protocole Atom

Le protocole Atom étend également la syntaxe XML d'Atom :

- pour décrire un *service*, vu comme un ensemble de *workspaces*, chacun contenant à son tour un ensemble de collections ;
- pour déclarer le type de contenus multimédia acceptés par chaque collection ;
- pour déclarer des thésaurus de *catégories* auxquelles les entrées d'une collection peuvent être rattachées.

Retour sur les contraintes REST

À la lumière des deux exemples précédents, on peut voir la mise en application des *contraintes* du style architectural REST.

Client-serveur

Separation of concern (Edsger W. Dikstra)

- Le serveur est responsable du stockage des données (état des *ressources*).
- Le client est responsable :
  - de la présentation/du traitement des données,
  - de maintenir le contexte/état de l'*interaction* (cf. ci après).

NB : certains composants sont à la fois client et serveur (e.g. proxy). Ils respectent cependant la séparation des préoccupations *vis-à-vis* des composants avec lesquels ils communiquent.

Connexion sans état

Le serveur ne doit pas s'encombrer du *contexte* de l'interaction. Ce contexte est rappelé à chaque requête par le client, et les modifications de contexte sont indiquées dans la réponse

→ auto-suffisance des messages.

Inconvénients

- surcoût en bande passante

Avantages

- performances (un serveur / beaucoup de clients)
- robustesse (redémarrage ou changement de serveur)
- facilite le travail des intermédiaires (cache, proxy)

Connexion sans état (remarques)

- On parle bien ici de l'état du *client*. Le serveur gère évidemment l'état des *ressources* dont il a la charge.
- Cette contrainte est souvent violée par les sites web:
  - identifiants de session
  - cookies
  - etc...

Support des caches

Les verbes et les statuts pré-définis par HTTP ont une sémantique suffisament précise pour informer les caches de la possibilité de « cacher » ou non un résultat.

- L'utilisation systématique de POST, par les premières versions de SOAP, rendait les caches totalement inopérants.

HTTP permet de contrôler plus finement le cache avec un certain nombres de champs d'en-tête (`Cache-Control`, `Expires`).

Interface uniforme

Les quatres principaux verbes GET, POST, PUT, DELETE constituent l'*interface uniforme* des ressources.

En principe, HTTP peut être étendu par de nouveaux verbes (e.g. WebDAV). Cependant, c'est une solution de dernier recours, car :

- cela limite l'interopérabilité, avec certains clients/serveurs, mais également avec les éventuels intermédiaires (proxy) ;
- cela limite l'adoption de l'application en donnant plus de travail (de compréhension et de codage) aux développeurs ;
- en pratique, rares sont les applications qui ne peuvent être décrites avec les quatres verbes de base.

Système en couches

- Cette contrainte spécifie qu'un composant ne doit se soucier que des composants avec qui il est en communication directe.
- Le protocole HTTP assure lui même la cohérence des messages le long de la chaîne d'intérmédiaires en spécificiant, en fonction de leur sémantique, quelles parties d'une requête ou d'une réponse sont *Hop-by-hop* ou *End-by-end*.

Code à la demande (optionel)

- Cette contrainte consiste à permettre au serveur d'envoyer au client non seulement la description d'un état, mais également une logique de traitement (programme).
- On peut voir AJAX (*Asynchronous Javascript and XML*) et ses dérivés (JSON) comme une généralisation de ce principe (même si AJAX n'utilise pas forcément des échanges RESTful).

4 Discussions

Représentation / Opérations

Sémantique déclarative plutôt qu'opérationnelle

- Alors que SOAP met l'accent sur les *opérations*, REST fournit un ensemble pré-défini d'opétations (interface unifiée) et met l'accent sur les *représentations*.
- La sémantique propre à l'application est donc principalement portée par le *format* des données échangées.

Analogie : les fichiers sous UNIX

Une des évolutions majeures apportée par UNIX au domaine des systèmes d'exploitations a été l'unification de la notion de fichier :

- un espace de nommage unique (le système de fichier)
- une interface uniforme (`open`, `read`, `write`...)

pour manipuler des ressources aussi variées que des fichiers de données, des périphériques, des *sockets*...

→ le succès de cette unification donne un certain crédit à l'unification proposée par REST.

Cookies

- Un cookie est une chaîne de caractères associée au site, envoyée par le serveur, et que le client doit joindre à toute future requête sur ce serveur.
- Le plus souvent, cette chaîne est un jeton opaque interprétable uniquement par le serveur (analogie aux *fortune cookies*).

Avantages et inconvénients

- Les cookies ne violent pas totalement les principes REST dans la mesure ou ils sont *explicitement* présents dans chaque requête.
    - Et contrairement aux identifiants de session passés dans l'URI, ils ne « polluent » pas les URIs des ressources.
- Cependant, puisqu'ils sont opaques, ils forcent le serveur à conserver une information qui concerne principalement l'état de l'interaction. On est donc entre deux.
    - En contre-partie, on gagne également de la bande passante.
    - ⚠ La sécurité par l'obscurité a ses limites.
- Autre inconvénient : les cookies changent *a posteriori* la sémantique des états précédemment mémorisés rencontrés par le client. Ils « cassent » le bouton *back* et certains mécanismes de cache.

Description des interfaces

Le chaînon manquant

Contrairement à SOAP, REST ne dispose pas d'un langage standard pour la description formelle des interfaces des services.

Un tel langage offrirait pourtant des fonctionalités intéressantes :

- vérification automatique de la conformité,
- génération automatique de code (squelettes, *stubs*),
- configuration automatique.

Certains efforts dans ce sens existent.

WSDL 2.0

- Évolution de *Web Service Description Language* recommandé en 2007 par le W3C.
- WSDL 2.0 permet de spécifier un *binding* HTTP au lieu de SOAP, utilisant n'importe quel verbe HTTP.
  - Permet donc en théorie de décrire des interfaces REST-like et RESTful...
  - ... quoi que le modèle conceptuel sous-jacent reste centré sur la notion d'*opération*.
- Des paramètres (partie de l'URI après le ?) peuvent être spécifiés à l'aide d'un schéma XML.
  - Cela permet notamment de leur associer un type de données.

hRESTS et SA-REST

- Propositions académiques (2007-2008) pour décrire les services REST-like.
- Elles utilisent l'*annotation sémantique* de la description textuelle des services, par un micro-format (hRESTS) ou RDFa (SA-REST).
- Propositions très influencées par SOAP :
  - SA-REST découle explicitement de SA-WSDL,
  - le modèle conceptuel de hREST est très proche de celui des services SOAP, centré sur la notion d'*opération*.

WADL

- *Web Application Description Language* : proposition de Sun en 2005-2006, publiée en 2009 comme *Member submission* du W3C.
- Plus simple que WSDL (pas de séparation entre interface et *binding*).
- La notion de *ressource* apparaît explicitement dans le modèle conceptuel.
    - Cependant la notion de *paramètre* reste hybride, pouvant être vue comme une partie de l'identifiant de la ressource *ou* comme un message envoyé à cette ressource.

Opacité des URIs (1)

Tim Berners-Lee défend le principe d'opacité des URIs, dont la violation conduit à des comportements *unRESTful*, comme :

- l'utilisation de l'extention (e.g. `.html`) pour inférer le type de représentation,
- le refus de stocker dans un cache les représentations dont l'URI comporte un ?,
- l'utilisation de l'heuristique ci-dessus pour empêcher la mise en cache d'une représentation (au lieu d'utiliser `Cache-Control`)...

Opacité des URIs (2)

Mais ce n'est qu'un principe *a priori* :

- les paramètres situés après le ? sont le plus souvent *construits* par le client à partir d'une spécification transmise par le serveur (e.g. formulaire HTML) ;
- un client peut utiliser une régularité *explicite* des URIs pour découvrir de nouvelles ressources, c'est notamment ce que propose WADL ;
- Roy Fielding a d'ailleurs préconisé ce mode de construction.

→ débat ouvert...

5 Références

- *Architectural Styles and the Design of Network-based Software Architectures*, Roy Fielding, Thèse de doctorat, 2000
- *Hypertext Transfer Protocol — HTTP/1.1*, Roy Fielding *et al.*, 1999, RFC 2616, IETF.
- *RESTful Web Services*, John Cowan, 2005 http://www.ccil.org/~cowan
- *Atom Publishing Protocol*, J. Gregorio & B. de Hora, 2007, RFC 5023, IETF.
- *Getting to know the Atom Publishing Protocol*, James Snell, 2006 http://www.ibm.com/developerworks/library/x-atompp1/
- *Describe REST Web services with WSDL 2.0*, Lawrence Mendel, 2008 http://www.ibm.com/developerworks/webservices/library/ws-restwsdl/

- *WSDL 2.0*, Roberto Chinnici *et al.*, 2007, W3C Recommendation, http://www.w3.org/TR/wsdl20, http://www.w3.org/TR/wsdl20-adjuncts
- *Web Application Description Language*, Marc Hadley, 2009, W3C Member Submission, http://www.w3.org/Submission/wadl/
- *hRESTS: an HTML Microformat for Describing RESTful Web Services*, Jacek Kopecký *et al.*, 2008, Web Intelligence
- *SA-REST: Semantically Interoperable and Easier-to-Use Services and Mashups*, Amit P.Sheth *et al.*, 2007, IEEE Internet Computing

- *Atom Publishing Protocol*, J. Gregorio *et al.*, 2007, Internet Draft, http://bitworking.org/projects/URI-Templates/draft-gregorio-uritemplate-00.html