Web Services :

REpresentational State Transfer

Table des matières

1. Introduction

a) Généralités

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.

b) Description

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 un protocole dont les messages seront transportés par HTTP.

Présenté comme cela, les différences entre un Web Service et un site web ne semblent pas évidentes...

Utilisation

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 ;
- protocole HTTP : bien connu et simple à aborder (protocole texte).

Historique

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 (**S**imple **O**bject **A**ccess **P**rotocol*)* en 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
- …

Objectifs de SOAP

L'objectif ambitieux est de définir des architectures orientées autour de services logiciels interconnectés (**S**ervice **O**riented **A**rchitecture). 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.

Limitations

Les 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 XML-RPC) ou dans SOAP ;
- Tout est décrit en XML par le langage WSDL.

Les inconvénients de l'architecture SOA sont :

- la mise en œuvre réelle est complexe ;
- les normes sont 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).

Tout cela est bien lourd à mettre en place et surtout à utiliser !

Cela explique pourquoi Google et Yahoo! ont effectué la transition de service SOAP vers des services REST.

2. REST

a) Principe

Le principe de REST est d'utiliser HTTP pour l'implémentation d'un Web Service, non plus comme simple protocole de transport, mais également pour définir l'API (**A**pplication **P**rogramming **I**nterface) 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).

b) Avantages de REST

Les avantages sont les suivants :

- Utilisation d'HTTP comme 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 serveurs caches entre le serveur et les clients de l'application ;
- interface uniforme → chaque Web Service REST a une interface orientée autour des messages de HTTP.

c) *Exemple d'utilisation basique*

Le Web Service Google Search, interrogé par ***RESTClient***, renvoie le contenu suivant :

Entête de la requête

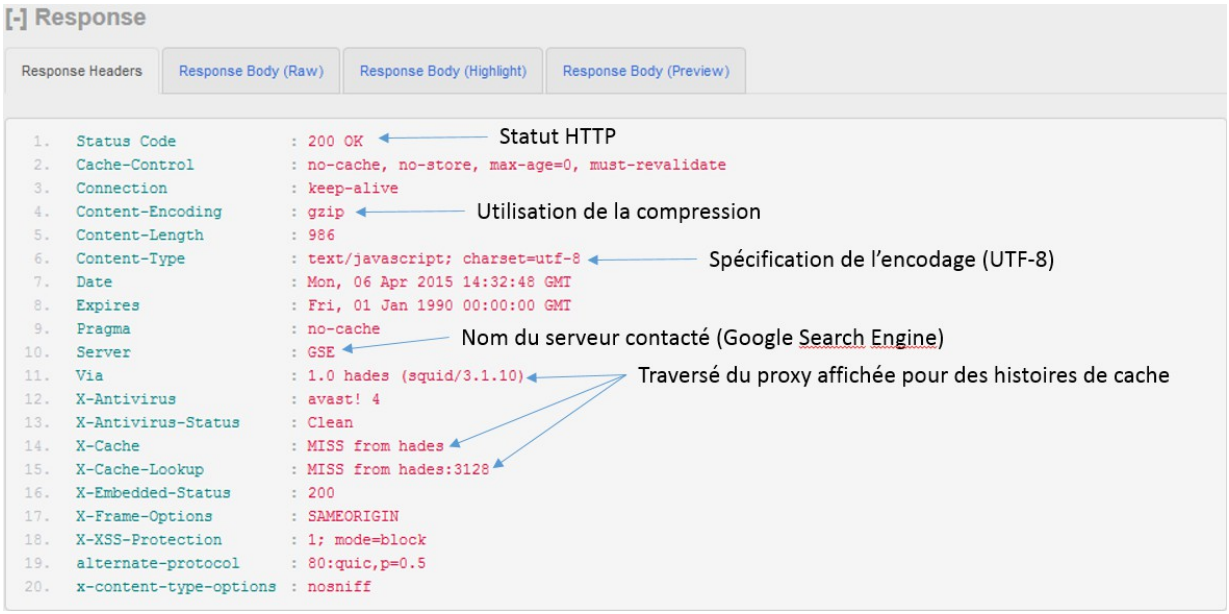

Illustration 1: Entête RESTful

Corps de la requête


```
{"responseData": {
    "results":[
        {
            "GsearchResultClass":"GwebSearch",
            "unescapedUrl":"http://en.wikipedia.org/wiki/Representational_state_transfer",
            "url":"http://en.wikipedia.org/wiki/Representational_state_transfer",
            "visibleUrl":"en.wikipedia.org",
            "cacheUrl":"http://www.google.com/search?q\u003dcache:CKDDr3lFulIJ:en.wikipedia.org",
            "title":"Representational state transfer - Wikipedia, the free encyclopedia",
            "titleNoFormatting":"Representational state transfer - Wikipedia, the free encyclopedia",
            "content":"Representational State Transfer (\u003cb\u003eREST\u003c/b\u003e) is a software architecture style consisting \no
        },
        {
            "GsearchResultClass":"GwebSearch",
            "unescapedUrl":"http://www.restapitutorial.com/lessons/whatisrest.html",
            "url":"http://www.restapitutorial.com/lessons/whatisrest.html",
            "visibleUrl":"www.restapitutorial.com",
            "cacheUrl":"http://www.google.com/search?q\u003dcache:gC0kBlToWzQJ:www.restapitutorial.com",
            "title":"What is \u003cb\u003eREST\u003c/b\u003e? - \u003cb\u003eREST\u003c/b\u003e API Tutorial",
            "titleNoFormatting":"What is REST? - REST API Tutorial",
            "content":"A tutorial on the six primary \u003cb\u003eREST\u003c/b\u003e constraints as explained by Roy Fielding."
        },
        {
            "GsearchResultClass":"GwebSearch",
            "unescapedUrl":"http://rest.elkstein.org/",
            "url":"http://rest.elkstein.org/",
            "visibleUrl":"rest.elkstein.org",
            "cacheUrl":"http://www.google.com/search?q\u003dcache:wjxxN6HbYBcJ:rest.elkstein.org",
            "title":"Learn \u003cb\u003eREST\u003c/b\u003e: A Tutorial",
            "titleNoFormatting":"Learn REST: A Tutorial",
            "content":"\u003cb\u003eREST\u003c/b\u003e stands for Representational State Transfer. (It is sometimes spelled \u0026quot;\
        },
        {
            ...
        }],
    "estimatedResultCount":"108000000",
    "currentPageIndex":0,
    "moreResultsUrl":"http://www.",
    "searchResultTime":"0.52"
    },
"responseDetails": null,
"responseStatus": 200}
```


Illustration 2: Corps RESTful

d) Genèse de REST

REST décrit un style d'architecture logicielle permettant de construire une application devant fonctionner sur des systèmes distribués (typiquement internet).

Si cela vous évoque le World Wide Web, rien d'étonnant vu que l'auteur de cette description est ***Roy Fielding***, l'un des principaux rédacteurs de la norme HTTP 1.1. (cf RFC 2616).

En résumé, REST est donc le style d'architecture soutenant le Web.

e) Principes directeurs

La construction d'un Web Service RESTful implique l'utilisation des éléments suivants :

- architecture client serveur ;
- des requêtes sans état (2 requêtes d'un client sont indépendantes) ;
- l'utilisation de mécanismes de cache possible ;
- une interface uniforme.

Architecture client-serveur

L'environnement **client-serveur** désigne un mode de communication entre plusieurs programmes :

- un client envoie des requêtes ;
- un serveur attend les requêtes des clients et y répond

Par extension, le client désigne également l'ordinateur sur lequel est exécuté le logiciel client, et le serveur, l'ordinateur sur lequel est exécuté le logiciel serveur.

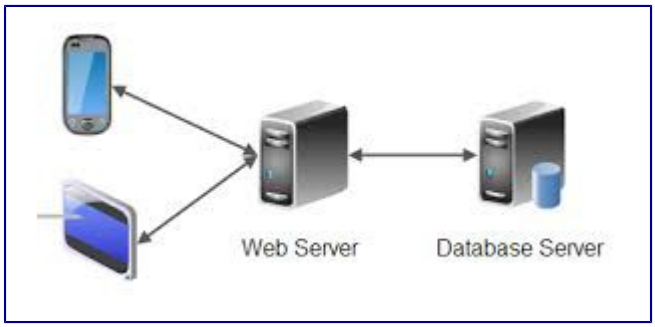

Illustration 3: Architecture 2 tiers

Requête stateless (sans état)

Cela 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.

Mécanisme de cache

L'idée est de pouvoir bénéficier du système distribué sous-jacent en permettant la mise en place de caches à chaque étape entre le client et le serveur.

Pour HTTP, cela consiste essentiellement en l'utilisation de serveurs ***proxy***.

Le plus connu est Squid :

Illustration 4: Logo du proxy Squid sous Linux

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 :

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

Exemple 1: Syntaxe d'une requête HTTP

f) Actions sur les ressources

Les différentes actions possibles sur ces ressources sont données par les différents types de requêtes HTTP, principalement :

- GET ;
- POST ;
- PUT ;
- 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 référencée par un hyperlien.

REST n'est pas un protocole, il n'existe donc pas de norme 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.

WSDL (**W**eb **S**ervice **D**escription **L**anguage) dans sa norme 2.0 permet :

- d'utiliser tous les messages de HTTP et donc de programmer des Web Services RESTful ;
- de garder la description des messages du protocole en XML, pour ceux qui veulent rester compatibles avec les normes du W3C.

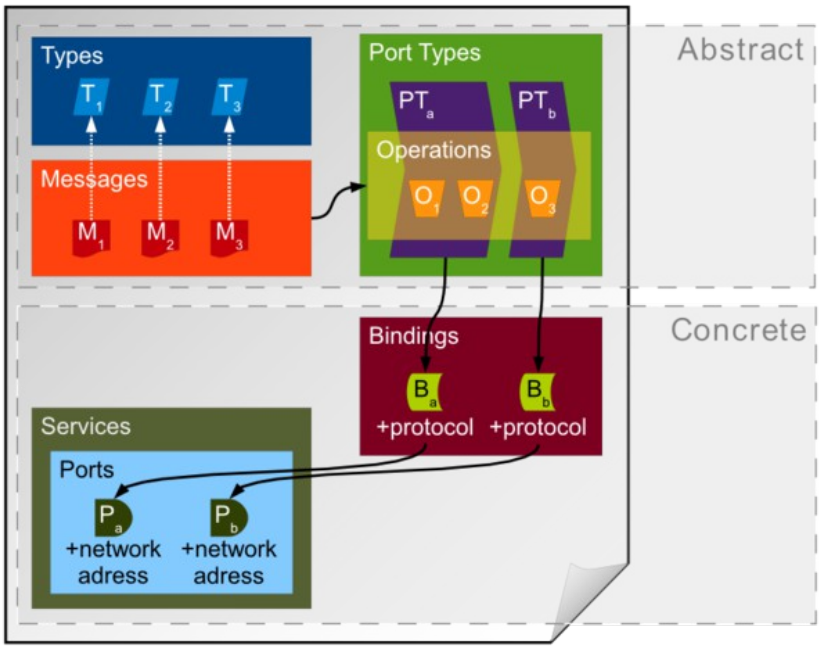

Illustration 5: Concepts définis par un document WSDL 1.1

3. Principes d'implémentation

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

- définition des ressources manipulées, collection de ressources, 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

a) Sémantique

Les différentes actions possibles sont définies par les verbes HTTP :

- GET : récupération d'une ressource ou d'une liste de ressources ;
- 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'informations 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.

b) 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, …)
- 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 ;

Traitement côté serveur

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
```

Exemple 2: Exemple de requête RESTful

Explication :

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

Traitement côté client

Pour le client, décodage d'une réponse HTTP (message en provenance du serveur) :

- décodage du code résultat ;
- en fonction du résultat (erreur ou non), 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.

4. Représentation des ressources

a) Objectifs

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 modèle).

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.

Cette opération s'appelle la sérialisation.

b) Sérialisation / Désérialisation

Sérialisation

C'est le processus visant à coder l'état d'une information qui est en mémoire sous la forme d'une suite d'informations plus petites (atomiques), le plus souvent des octets voire des bits.

Cette suite pourra être utilisée pour la sauvegarde (persistance) ou le transport sur le réseau (proxy, RPC…).

Désérialisation

L'activité symétrique, visant à décoder cette suite pour créer une copie conforme de l'information d'origine, s'appelle la désérialisation (ou ***unmarshalling***).

Cas d'utilisation

Les termes *marshalling* et *unmarshalling* s'emploient le plus souvent dans le contexte d'échanges entre programmes informatiques, alors que les termes *sérialisation* et *désérialisation* sont plus généraux.

Complexité

D'apparence simple, ces opérations posent en réalité un certain nombre de problèmes, comme la gestion des références entre objets ou la portabilité des encodages.

Les choix entre les diverses techniques de sérialisation ont une influence sur les critères de performances comme la taille des suites d'octets sérialisées ou la vitesse de leur traitement.

c) Format

Comme spécifié plus haut, les messages d'une application RESTful sont indépendants les uns des autres ce qui implique 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
```

Exemple 3: Spécification du type côté serveur

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

```
Accept: text/xml
```

Exemple 4: Spécification du type côté client

d) JavaScript Object Notation : JSON

Raison du succès

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 un format texte qui permet de représenter des données et de les échanger facilement à l'instar d'XML.

Description du langage

JSON permet de décrire le modèle objet de JavaScript grâce à deux types de structures disponibles :

- Objet : une collection de paires nom/valeur (un tableau associatif) ;
- Tableau : une liste ordonnée de valeurs.

Les valeurs peuvent être des types suivants : booléen, chaîne de caractères, nombre, ou valeur nulle ou une des structures ci-dessus.

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

```
{
"nom":"JC",
"age":29,
"adresse": { "rue":"Ras el mâa", "ville":"Casablanca", "code":20200, "pays":"Maroc"},
"telephone":[ {"type":"maison", "numero" : 123456}, {"type":"portable", "numero": 654321} ]
}
```

Exemple 5: Marshalling JSON

L'équivalent en XML serait :

```
<utilisateur nom="JC" age="29">
	<adresse>
		<rue>Ras el mâa</rue>
		<ville>Casablanca</ville>
		<code>20200</code>
		<pays>Maroc</pays>
	</adresse>
	<telephones>
		<telephone type="maison">123456</telephone>
		<telephone type="portable">654321</telephone>
	</telephones>
</utilisateur>
```

Exemple 6: Marshalling XML

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 + ")");
```

Exemple 7: Code JavaScript permettant de désérialiser un objet

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

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

Exemple 8: Utilisation des librairies JavaScript pour la désérialisation

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

5. API REST

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.

Définition des ressources

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.

Définition de l'API

Les différentes actions possibles sur ces 2 types de ressources sont :

URI	Sémantique	Code réponse
GET http://server/users	Récupère la liste des utilisateurs	200 OK
POST http://server/user	Crée 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 resource not found
DELETE http://server/user/id_user	Efface un utilisateur	200 Ok 404 resource not found

Illustration 6: Exemple d'implémentation d'une API RESTful

Les deux premiers schémas de requêtes adressent la collection entière des utilisateurs, tandis que les suivants visent un membre particulier de la collection.

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.

Définition du traitement

C'est à la partie serveur de réaliser l'analyse des requêtes pour déterminer quelles sont les ressources visées par une requête particulière.

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

- Utilisateur :

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

Exemple 9: Représentation d'un utilisateur

- Liste :

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

Exemple 10: Représentation d'une collection d'utilisateurs

6. 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.

a) 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.

Authentification

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 ...)

Version d'un objet : timestamp, cache et Etag

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 conditionnels 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.

b) Asynchronous JavaScript And XML : 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.

Reprenons l'exemple de l'API utilisateur. Le code JavaScript suivant permet d'y accéder pour récupérer la liste des utilisateurs :

```
function getXhr() {
	var xhr;
	if (window.XMLHttpRequest) {
		// code for IE7+, Firefox, Chrome, Opera, Safari
		xhr = new XMLHttpRequest();
	} else {
		// code for IE6, IE5
		xhr = new ActiveXObject("Microsoft.XMLHTTP");
	}
	return xhr;
}
function getUserList(){
	var xhr = getXhr();
	xhr.open("GET", "http://serveur.com/users",false);
	xhr.send();
	var result = xhr.responseText;
	var users = JSON.parse(result);
	// permet d'afficher le premier utilisateur
	alert(users[0].login+" : "+users[0].password);
}
```

Exemple 11: Code AJAX permettant d'interagir avec un WS RESTFul

Annexes

1. Index des exemples

Index des exemples

2. Index des illustrations

Index des illustrations