Les Web-Services

Jonathan Lejeune

Sorbonne Université/LIP6-INRIA

SRCS – Master 1 SAR 2019/2020

sources :

- Cours de Lionel Médini (Université Lyon 1)
- Cours de Pierre-Antoine Champin (Université Lyon 1)
- RESTFul Web Services – L. Richardson, S. Ruby
- RESTful Service Best Practice, Todd Fredrich

Les APIs de communication répartie vues précédemment

Récapitulatif

- Sockets
  - ✓ API bas niveau permettant des mécanismes fins et performants
  - ✗ Nécessité de redéfinir tout un protocole propre à une seule application
- Les appels distants
  - ✓ Intégration naturelle dans un langage de programmation
  - ✓ Développement simplifié, moins de dépendances aux API réseau
  - ✗ Sérialisation/encodage souvent propre à l'intergiciel
  - ✗ Identification des services non standardisé, ports d'écoute non standard
- Les composants (ex : EJB)
  - ✓ Abstraction complète du réseau et du déploiement
  - ✗ Communication inter-composants restreinte à l'intranet de l'entreprise
  - ✗ EJB = application et sérialisation Java obligatoire

Inconvénient commun

L'interopérabilité n'est pas universelle.

Vers une interopérabilité des applications

Objectifs

Faire un sorte que deux processus puisse communiquer fiablement :

- quel que soit leur langage de programmation source
  - ⇒ **Besoin d'un standard de communication**
- quel que soit l'OS et l'architecture de leur hôte
  - ⇒ **Besoin d'un standard de représentation des données**
- quel que soit leur réseau physique et leur domaine
  - ⇒ **Besoin d'un protocole distribué reconnu, standardisé et ouvert**
- quel que soit les services qu'ils offrent
  - ⇒ **Besoin d'une interface qui universalise :**
    - ⇒ **l'identification du service**
    - ⇒ **l'invocation du service**
    - ⇒ **la paramétrisation du service**
    - ⇒ **la sémantique de la réponse du service**

Vers une interopérabilité des applications

Le Web semble être une très bonne piste

- Il fonctionne sur TCP
  - ⇒ Canaux fiables et FIFO
- Il est défini par une RFC
  - ⇒ Protocole de communication standardisé
- Il offre des mécanismes de routage applicatif
  - ⇒ Mécanisme de répartition de charge et de firewall
- Ses enveloppes de message sont textuelles
  - ⇒ Sémantique des messages standardisée
- Ses messages transportent des données formatées
  - ⇒ La représentation des données est standardisée
- Il est utilisé par tout le monde (au sens propre)
  - ⇒ Protocole reconnu et ouvert
- Il est sans état
  - ⇒ Pas de session à gérer

Vers une interopérabilité des applications

Le Web semble être une très bonne piste

- Il peut fonctionner sur un protocole de sécurité comme TLS (HTTPs)
  $\Rightarrow$ Communication sécurisée
- Il utilise des URI
  $\Rightarrow$ identification universelle
- Il a un nombre prédéfini d'opérations élémentaires (`GET`, `POST` ...)
  $\Rightarrow$ invocation d'opération universelle
- Il définit des mécanismes de négociation de contenu
  $\Rightarrow$ paramétrisation adaptable et flexible
- Il définit des codes de retour
  $\Rightarrow$ sémantique de réponse standard

Attention

Cette interopérabilité se fait au détriment des performances
$\Rightarrow$ messages plus volumineux à cause de l'aspect textuel

L'utilisation du Web

L'utilisation classique de M. ToutLeMonde

- Un navigateur qui :
  - envoie des requêtes HTTP à un serveur paramétré par l'utilisateur
  - reçoit les réponses sous format XML (ex : HTML)
  - Afficher de manière lisible pour un humain la réponse
- L'utilisateur final interprète la signification du message à travers le navigateur

Problématique

Comment automatiser ce schéma pour les applications distribuées ?

La notion de Web-Service répond à cette problématique

Web service

Définition

Interface d'une fonctionnalité applicative accessible par le réseau en utilisant les standards de l'Internet et via la combinaison de ses protocoles (HTTP, XMl, JSON, . . .).

Deux familles de Web-Service

- **SOAP** : **S**imple **O**bject **A**ccess **P**rotocol
  $\Rightarrow$ technique historique et obsolète
- **REST** **Re**presentational **S**tate **T**ransfer

Un mot sur SOAP

Qu'est ce que c'est ?

- Protocole d'échange d'information structurée pour invoquer des services
- Les messages SOAP sont encapsulés dans une requête HTTP
- Un message SOAP est composé de headers et d'un corps
- Définitions d'interfaces grâce à un langage pseudo-XML : le WDSL

Le WebService Description Language

Permet de décrire un web-service :

- le protocole de communication
- le format de messages requis pour communiquer avec ce service
- les méthodes que le client peut invoquer
- la localisation du service.

Les raisons de l'obsolescence de SOAP

Inconvénients de SOAP

- HTTP n'est réduit qu' à son aspect de protocole de transport
  ⇒ **Inexploitation des propriétés d'interopérabilité applicative**
- Redéfinition d'une couche protocolaire au-dessus de HTTP
  ⇒ **Aggravation des performances à cause de messages lourds**
- Il reste relativement complexe à prendre en main
  ⇒ **Contraire à la philosophie du web**
- Il a une spécification volumineuse amplifiée par une profusion d'extensions (WS-Policy, WS-Security, WS-Trust . . .)
  ⇒ **implantations incomplètes dans certains langages (PHP, Python)**
- La définition d'interfaces = couplage fort entre serveur et client
  ⇒ **Perte de standardisation et de réutilisabilité**

Au final, on a paradoxalement limité l'interopérabilité qui était l'objectif initial

REST

Qu'est-ce que c'est ?

- Un style architectural pour construire des applications réparties basées sur les technologies Web (thèse de Roy Fielding, 2000)
- Ensemble de conventions et de bonnes pratiques à respecter
  $\Rightarrow$ **Définition de 6 contraintes**
- En pratique, REST est une exploitation pleine de HTTP

Qu'est-ce que ça n'est pas ?

Une technologie à part entière
$\Rightarrow$ On ne réinvente rien, on utilise les spécifications originelles de HTTP

Jargon du monde REST

- **RESTful** : API respectant pleinement les principes REST
- **RESTlike** : API respectant partiellement les principes REST
- **RESTafarian** : programmeur utilisant REST

Contrainte no. 1 : Être client-serveur

Objectif (rappel)

Séparation des responsabilités :

- Le serveur fournit un service spécifique à travers une interface
- Le client requête le service et traite sa réponse

✓ Séparation spécification service et de son implémentation

✓ Client et serveur peuvent évoluer indépendamment

En pratique

HTTP est par définition un protocole client/serveur permettant :

- Envoi de requête par le client
- Envoie d'une réponse par le serveur

Contrainte no. 2 : Connexion sans état (stateless)

Objectif (rappel)

Chaque requête contient toute information nécessaire pour la comprendre

- Le serveur ne gère pas les états de l'interaction avec le client
- Les requêtes sont indépendantes l'une des autres
- ✓ Moins de ressources consommées par le serveur
- ✓ Maintenance et évolution facilitées
- ✓ Meilleure tolérance aux pannes

En pratique

- L'aspect sans état est une propriété fondamentale de HTTP
- Pour passer un contexte dans une requête HTTP, on peut utiliser :
  - la partie query de l'URI : `https://www.foo.com/?q=test&t=web`
  - le corps de la requête : `{"query":"test","type":"web"}`
  - les headers de la requête : `authorization`, `cookie` (stateless, mais pas restful) . . .

Contrainte no. 3 : Support des caches

Objectif

Mise en cache de réponse sur un serveur, un intermédiaire ou sur le client :

- Nécessité de taguer les réponses comme cacheable ou non pour empêcher la lecture de données obsolètes ou inappropriées

✓ cache ⇒ évite des interactions client-serveur ⇒ gain en performance

En pratique

Le cache est une fonctionnalité optionnelle de HTTP (RFC 7234)

- Notions de Freshness, d'âge, de Stale response (périmée)
- Headers : `Age`, `Cache-Control`, `Expires`, `Warning`
- Fonctionnement grâce aux requêtes conditionnelles :
  - avec `Last-Modified` → `If-Modified-Since`, `If-Unmodified-Since`, `If-Range`
  - avec `Etag` (entity tag) → `If-Match`, `If-None-Match`
- Codes de réponse : 304 Not Modified, 412 Precondition Failed

Contrainte no. 4 : système en couches

Objectif

Déploiement indépendant des composants (client, serveur, intermédiaire) :

- La connexion à un serveur intermédiaire ou final doit être transparente pour le client
- ✓ répartition de charge possible
- ✓ cache partagé possible
- ✓ politiques de sécurité renforcées

En pratique

La notion de composant intermédiaire est intégrée à HTTP

- Messages auto descriptifs donc interprétables par les intermédiaires
- Notion de proxy : à la fois des clients et des serveurs
- Codes de retour (305 Use Proxy, 502 Bad Gateway)
- Directives de cache (`no-store`)

Contrainte no. 5 (optionnelle) : code à la demande

Objectif

Apporter au client la possibilité d'étendre la fonctionnalité du service en téléchargeant et en exécutant du code

- Format applet ou script
- ✓ Simplifie le travail du client
- ✓ Permet de faire évoluer l'interface
- Facultatif car réduit la visibilité de l'organisation des ressources

En pratique

- Javascript + (XML ou JSON)
- Applet Java

Contrainte no. 6 : interface uniforme

Objectif

Permettre de découpler clients et serveurs uniformément

- Analogie à l'interface des systèmes de fichiers UNIX (`open`, `read` ...)
- REST définit 4 principes

Principe 1 : Orienté ressource

Une ressource symbolise un objet du domaine

- peut être une entité réelle (fichier, tuple de BD, ...) ou virtuelle
- doit être identifiée par un nom
- est conceptuellement distincte de sa représentation retournée au client

Principe 1 : Orienté ressource en pratique

- L'identification est assurée par l'URI et particulièrement son path
- La ressource peut être représentée en HTML, XML ou JSON

Contrainte no. 6 : interface uniforme

Principe 2 : Manipulation des ressources par des représentations

La manipulation d'une ressource se résume par 4 actions (**CRUD**) :

- **C**reate : créer une nouvelle ressource
- **R**ead : obtenir une représentation d'une ressource
- **U**pdate : mettre à jour l'état d'une ressource
- **D**elete : supprimer une ressource

Principe 2 en pratique

HTTP définit des méthodes (verbes) correspondant aux actions possibles :

- **C**reate → `POST` (ou `PUT`)
- **R**ead → `GET` : action safe, idempotente et cacheable
- **U**pdate → `PUT` : action idempotente
- **D**elete → `DELETE` : action idempotente

Contrainte no. 6 : interface uniforme

Principe 3 : Messages auto-descriptif

Chaque message contient assez d'information pour savoir comment l'interpréter.

- les requêtes peuvent préciser le type de ressources à renvoyer
- les réponses indiquent explicitement leur sémantique

Principe 3 en pratique

- HTTP offre un mécanisme de négociation de contenu dans la requête
  - header : `Accept`, `Accept-Language`, `Accept-Encoding`
  - extension dans l'URI : `http://truc.fr/users/toto.html` ou `http://truc.fr/users/toto.json` ou `http://truc.fr/users/toto.xml`
- HTTP définit des codes de retour pour la réponse :
  - `GET` : `200 OK` ou `206 Partial Content`
  - `POST` : `201 Created` + lien vers la ressource dans le header `Location`
  - `PUT` : `200 OK` ou `204 No Content`
  - `DELETE` : `204 No Content`

Contrainte no. 6 : interface uniforme

Principe 3 : Messages auto-descriptif

Chaque message contient assez d'information pour savoir comment l'interpréter.

- les requêtes peuvent préciser le type de ressources à renvoyer
- les réponses indiquent explicitement leur sémantique

Principe 3 en pratique

- HTTP offre un mécanisme de négociation de contenu dans la requête
  - header : `Accept`, `Accept-Language`, `Accept-Encoding`
  - extension dans l'URI : `http://truc.fr/users/toto.html` ou `http://truc.fr/users/toto.json` ou `http://truc.fr/users/toto.xml`
- HTTP définit des codes de retour pour la réponse :
  - `GET` : `200 OK` ou `206 Partial Content`
  - `POST` : `201 Created` + lien vers la ressource dans le header `Location`
  - `PUT` : `200 OK` ou `204 No Content`
  - `DELETE` : `204 No Content`

Contrainte no. 6 : interface uniforme

Principe 4 : HATEOAS

Hypermedia **A**s **T**he **E**ngine **O**f **A**pplication **S**tate

- Être en mesure d'utiliser les hypermédias fournis dans la réponse
- Le client choisit librement quels liens suivre pour exécuter l'application
- Notion d'**affordance** : Le client doit pouvoir comprendre la sémantique des liens (destination, mode d'utilisation, effet attendu)

Principe 4 en pratique

- Le serveur répond aux requêtes en renvoyant des contrôles hypermédias en utilisant header et corps de la réponse

```
Location: https://foo.com/users/toto
Link: <https://foo.com/users/titi>; rel="previous"; title="previous user"

{"userId":"toto", "login":"toto",
   "links": [{"href": "https://foo.com/users/toto/messages", "rel": "messages", "type" : "GET"}]
}
```

- Le client parcourt les liens grâce aux contrôles hypermédias soit manuellement (utilisateur humain) soit automatiquement (programme)

Un mot sur les cookies

Rappels

Un cookie est une chaîne de caractères url-encodée stockée sur le disque dur du client permettant de gérer des sessions.

Cas des cookies de session

Contiennent uniquement un identifiant ; les données de session sont conservées côté serveur

$\Rightarrow$ propriété Stateless violée

Cas des cookies contenant explicitement les "données de session"

- Stateless, car les données sont bien envoyées dans la requête
- Pas RESTful car le client n'est pas libre de choisir l'état de l'interaction

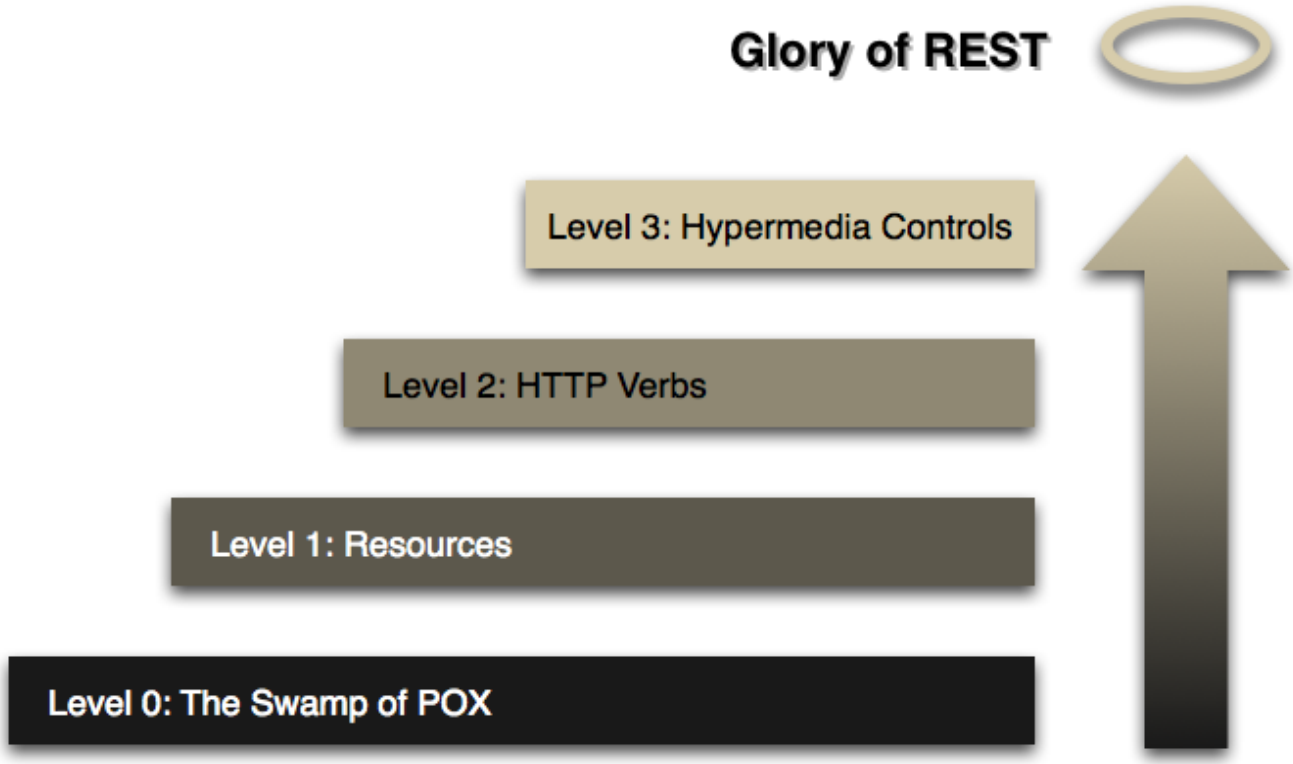
  $\Rightarrow$ c'est le serveur qui impose la valeur des cookies

Les Cookies sont à proscrire en REST

Définir une API REST

Le modèle de maturité de Richardson

- Modèle décrivant une méthode pour noter une API en fonction des contraintes REST
- Une note de 0 à 3 : plus la note est haute, plus l'API est RESTful.



Le modèle de maturité de Richardson - Niveau 0

The Swamp of Plain Old XML (niveau de SOAP)

- Utiliser HTTP comme un protocole de transport
- Utiliser une seule méthode HTTP (en pratique `POST`)
- Quelques contraintes de syntaxes sur les URIs :
  - Utilisation
    - des tirets pour séparer des mots dans un nom
    - des minuscules exclusivement
  - Proscrire :
    - les underscores
    - les extensions de fichiers (utiliser le header `Content-Type`)

Le modèle de maturité de Richardson - Niveau 1

Resources

- Les URIs servent à interagir avec les différentes ressources de l'appli
- Ne pas utiliser de verbes dans le nom d'un URI
- Ajout de contraintes supplémentaire sur la syntaxe des URIs
  - L'URI ne doit pas se terminer par un slash
    2 URIs différentes $\Leftrightarrow$ 2 ressources différentes
  - **le slash indique une relation hiérarchique entre les ressources**
    $\Rightarrow$ partie path de l'URI
  - Pluriel ou singulier du dernier nœud du path ?
    - pluriel : désigne une collection d'objets
    - singulier : désigne l'identifiant d'un objet

Exemples :

```
http://example.org/foo/articles
http://example.org/foo/articles/1
http://example.org/foo/articles/2
http://example.org/bar/singleton
```

Le modèle de maturité de Richardson - Niveau 2

HTTP verbs

- Utiliser les méthodes HTTP pour agir sur la ressource **et** garder leur sémantique
- Utiliser les headers de HTTP pour les métadonnées
- Pour les ressources collections utilisation de la partie query de l'URI pour :
  - renseigner un range d'id de sous-ressources voulu
  - filtrer le résultat en fonction d'attribut
  - trier les résultats
  - chercher une sous-ressource
- Utiliser les codes de retour définis par HTTP et préserver leur sémantique

Le modèle de maturité de Richardson - Niveau 3

Hypermedia controls

- Utiliser le mécanisme de négociation de type de ressource

  `Accept`, `Content-Type`

- **H**ypermedia **A**s **T**ransfer **E**ngine **O**f **A**pplication **S**tate (HATEOAS)
- Pouvoir versionner les ressources et avoir la capacité de préciser une version voulue :
  - via Header : `Accept: application/vnd.hashmapinc.v2+json`
  - via URI : `POST /v2/user`

Web API

Caractéristiques

- Un ensemble de ressources RESTful
- Un point d'entrée unique (URI de base)
- Une **documentation** permettant
  - de comprendre comment elle s'utilise
  - de naviguer entre les ressources (HATEOAS)

Vocabulaire

- Un serveur expose une **Web API**
- Un client consomme une **Web API** et peut en agréger plusieurs (mashup)

Un site qui recense la plupart des Web-APIs existantes : `https://any-api.com/`

Programmer une application distribuée Web

Point de vue global

Suivre les recommandations de REST pour alléger les serveurs et permettre l'évolution des clients :

- Exposer des ressources sur un serveur
- **Documenter** l'API du serveur Web
- Réaliser le client séparément :
  - JavaScript pour les navigateurs
  - Application desktop ou mobile
  - Sur un autre serveur qui consomme les ressources du premier
  - Sur des composants intermédiaires qui exposent/filtrent/transforment les données

Programmer une application distribuée Web

Frameworks orientés ressources pour Java

- Java API for RESTful Web Services (JAX-RS) → intégré à JavaEE
  - Une spécification Java pour développer facilement une API REST :
    - `@Path`, `@PathParam`, `@QueryParam`
    - `@GET`, `@POST`, `@DELETE`, `@PUT`, `@HEAD`
    - `@Produces`, `@Consumes`, ...
  - Des implémentations : Apache CXF, Jersey, TomEE+ ...
- Restlet (dévelopement : Talend)
  - fonctionne en JavaSE ou en JavaEE
  - Offre une API de développement propre mais peut aussi s'étendre pour se conformer à JAX-RS

Autres outils

- Documenter / tester une API : Swagger
- Tester / scénariser : Postman, Soapui