EPSI I4 Web Services / **David Gayerie** - 2013-2014

WEB SERVICES RESTFUL

REST

REST est un style d'architecture pour les systèmes hypermédia distribués.

Une implémentation qui se conforme à REST est dite *RESTful*.

En 2000, Roy Fielding a formalisé l'architecture REST dans le chapitre 5 de sa **thèse (http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm)** .

En 2007, Leonard Richardson et Sam Ruby présentent ROA (Resource Oriented Architecture) dans leur ouvrage *RESTful Web Services*. ROA est devenu le modèle d'application de REST pour les Web services.

Depuis quelques années, REST et RESTful sont devenus des buzzwords :(

Idée fausse : Je développe des Web Services RESTful car j'utilise un framework REST (comme JAX-RS).

La première chose à retenir :
REST est le modèle d'architecture du Web !

Un Web service RESTful suit les mêmes principes qu'un site Web pour les humains.

La différence principale tient au fait que la sémantique des données échangées *via* le Web service est compréhensible par un programme (Web programmable).

Un Web service RESTful devrait être conçu comme un site Web (et réciproquement).

Domaines principaux d'application des Web services RESTful :

- Applications Web riches (Ajax)
- Applications pour Mobile
- Le Web des objets
- L'open data

LA RESSOURCE

REST est un style d'architecture **orienté ressource**.

Qu'est-ce qu'une ressource ?

> *“Any information that can be named can be a resource”*
>
> *Roy Fielding*

Dans le domaine de Web :

```
Information nommée = Information adressable
```

```
Adressable = URI (Uniform Resource Identifier)
```

Une ressource Web est une donnée adressable par une ou plusieurs URI.

Exemple : ce cours est adressable à partir de l'URI **http://spoonless.github.io/epsi-i4-web-services**

Une ressource de Web service peut être accessible à partir de plusieurs URI. Une bonne pratique consiste à considérer qu'une des URI est l'URI "canonique".

Exemple d'un service pour connaître la liste des étudiants :

```
http://epsi.fr/bordeaux/informatique/i4/etudiants/2014 (URI canonique)
```

```
http://epsi.fr/bordeaux/informatique/i4/etudiants/annee-courante
```

Ces deux URI désignent la même ressource pendant une période.

Le serveur peut rediriger le client vers l'URI "canonique".

Canonicalisation par redirection

```
GET /bordeaux/informatique/i4/etudiants/annee-courante HTTP/1.1
Host: epsi.fr
```

```
HTTP/1.1 303 See Other
Location: http://epsi.fr/bordeaux/informatique/i4/etudiants/2014
```

Le serveur peut simplement informer le client qu'il existe une URI "canonique" grâce à l'en-tête *Content-location* dans la réponse.

Canonicalisation par lien

```
GET /bordeaux/informatique/i4/etudiants/annee-courante HTTP/1.1
Host: epsi.fr
```

```
HTTP/1.1 200 OK
Content-location: http://epsi.fr/bordeaux/informatique/i4/etudiants/2014
Content-type: application/xml; charset=utf-8
Content-length: 436

<?xml version="1.0" encoding="utf-8"?>
<etudiants>
  ...
</etudiants>
```

Une bonne pratique est de définir des URI descriptives et hiérarchiques

```
http://epsi.fr/bordeaux/informatique/i4/etudiants/2014
```

Cette URI suit un pattern facilement compréhensible :

```
http://epsi.fr/{ville}/{spécialité}/{code année}/etudiants/{promotion}
```

Des URIs bien conçues facilitent l'utilisation des services.

REST ne doit pas être confondu avec une autre approche de système distribué : le RPC (Remote Procedure Call)

Les services RPC exposent des procédures et sont **orientés traitement**. Le client envoie des paramètres et demande au serveur de lui retourner un résultat.

Les URI des Web services RPC se terminent souvent par des verbes.

Idée fausse : Je développe des Web Services RESTful car mes services sont chacuns accessibles *via* une URI.

Exemple d'URI non RESTful:

```
http://mon.site.com/article147/delete
```

```
http://mon.site.fr/data?ressource=ma+ressource
```

LA REPRÉSENTATION

Dans une architecture REST, un client et un serveur n'échangent pas une ressource mais une représentation, c'est-à-dire *un ensemble de données relatives à l'état courant d'une ressource*.

Une même ressource Web peut avoir plusieurs représentations : une image JPEG, une page HTML, un document XML, un document JSON...

Un serveur peut choisir la représentation de la ressource pour correspondre au mieux aux capacités du client : on parle de négociation de contenu.

La négociation de contenu peut se faire grâce à l'en-tête *Accept* dans la requête

Négociation de contenu par en-tête

```
GET /bordeaux/informatique/i4/etudiants/2014 HTTP/1.1
Host: epsi.fr
Accept: image/jpeg, application/json, */*
```

```
HTTP/1.1 200 OK
Content-type: application/json; charset=utf-8
Content-length: 350

{
  ...
}
```

Si le serveur ne peut fournir une représentation acceptable par le client, il peut répondre le code erreur 415 "Unsupported Media Type".

La négociation de contenu peut se faire avec l'extension utilisée dans le chemin de la ressource.

Négociation de contenu par extension

```
GET /bordeaux/informatique/i4/etudiants/2014.json HTTP/1.1
Host: epsi.fr
```

```
HTTP/1.1 200 OK
Content-type: application/json; charset=utf-8
Content-length: 350

{
  ...
}
```

Idée fausse : Je développe des Web Services RESTful car mon serveur retourne du JSON (ou XML (ou YAML (ou ATOM))).

Pour Java EE, avec JAX-RS, le format de la représentation est géré par les annotations

***@Consumes* (http://docs.oracle.com/javaee/6/api/javax/ws/rs/Consumes.html)**

Indique les types MIME du corps de la requête que peut accepter une méthode

***@Produces* (http://docs.oracle.com/javaee/6/api/javax/ws/rs/Produces.html)**

Indique les types MIME du corps de la réponse que peut produire une méthode

JAX-RS inclut un **binding (http://docs.oracle.com/javaee/6/tutorial/doc/gkknj.html)** entre les classes Java et les types MIME application/xml, application/json grâce aux annotations JAXB.

Utilisation des annotations @Consumes et @Produces :

```
@PUT
@Path("{id}")
@Consumes({ "application/json", "application/xml" })
public void merge(@PathParam("id") String id, MyRepresentation rep) {
  // ...
}

@GET
@Path("{id}")
@Produces({ "application/json", "application/xml" })
public MyRepresentation get(@PathParam("id") String id) {
  // ...
}
```

Utilisation du binding JAXB sur la représentation

```
@XmlRootElement
public class MyRepresentation {
  // ...
}
```

Le type MIME application/x-www-form-urlencoded (formulaire HTML) est supporté grâce à l'annotation **@FormParam (http://docs.oracle.com/javaee/6/api/javax/ws/rs/FormParam.html)** qui permet de passer une valeur d'un élément du formulaire comme paramètre à la méthode Java.

Utilisation des annotations @Consumes et @FormParam :

```
@PUT
@Path("{id}")
@Consumes("application/x-www-form-urlencoded")
public void merge(@PathParam("id") String id,
                  @FormParam("name") String name,
                  @FormParam("age") int age) {
  // ...
}
```

Pour le support des autres types MIME, JAX-RS peut être étendu en fournissant une implémentation des interfaces :

***MessageBodyReader<T>* (http://docs.oracle.com/javaee/6/api/javax/ws/rs/ext/MessageBodyReader.html)**

Une interface définissant les méthodes pour la conversion du flux d'entrée de la requête vers une classe Java.

***MessageBodyWriter<T>* (http://docs.oracle.com/javaee/6/api/javax/ws/rs/ext/MessageBodyWriter.html)**

Une interface définissant les méthodes pour la conversion d'une classe Java dans le flux de sortie de la réponse.

Une implémentation d'un MessageBodyReader<T> ou d'un MessageBodyWriter<T> est limitée au type Java T et aux types MIME spécifiés avec les annotations **@Consumes (http://docs.oracle.com/javaee/6/api/javax/ws/rs/Consumes.html)** et **@Produces (http://docs.oracle.com/javaee/6/api/javax/ws/rs/Produces.html)** .

Une implémentation d'un MessageBodyReader<T> ou d'un MessageBodyWriter<T> DOIT être annotée avec **@Provider (http://docs.oracle.com/javaee/6/api/javax/ws/rs/ext/Provider.html)** .

La structure d'une implémentation d'un MessageBodyWriter<T>

```
@Provider
@Produces("image/png")
public class ImageBodyWriter implements MessageBodyWriter<MyRepresentation>{

  public long getSize(String s, Class<?> rawType, Type genericType,
                      Annotation[] annotations, MediaType mediaType) {
    // ...
  }

  public boolean isWriteable(Class<?> rawType, Type genericType,
                             Annotation[] annotations, MediaType mediaType){
    // ...
  }

  public void writeTo(MyRepresentation s, Class<?> rawType,
                      Type genericType, Annotation[] annotations,
                      MediaType mediaType,
                      MultivaluedMap<String, Object> httpHeaders,
                      OutputStream entityStream) throws IOException {
    // ...
  }

}
```

Les classes annotées avec **@Provider (http://docs.oracle.com/javaee/6/api/javax/ws/rs/ext/Provider.html)** sont ensuite rendues disponibles à JAX-RS par la classe représentant l'application (c'est-à-dire la classe qui étend **Application (http://docs.oracle.com/javaee/6/api/javax/ws/rs/core/Application.html)**).

Déclaration d'un provider dans une application :

```
@ApplicationPath("api")
public class MyApplication extends javax.ws.rs.core.Application {

  @Override
  public Set<Class<?>> getClasses() {
    Set<Class<?>> classes = new HashSet<Class<?>>();
    classes.add(ImageBodyWriter.class);
    return classes;
  }

}
```

LES CONTRAINTES REST

- client/serveur
- interface uniforme (uniform interface)
- sans état (stateless)
- mise en cache
- layered system
- code-on-demand

client/serveur

L'architecture **Client/Serveur (http://en.wikipedia.org/wiki/Client_server)** est une architecture logique centralisée pour le développement d'applications distribuées.

Un programme serveur se met en attente de requêtes de programmes clients vers lesquels il retourne une réponse.

- *Web*

  Le Web est en grande partie conçu à partir de cette contrainte. HTTP est un protocole élaboré à partir du modèle client/serveur.

- *JAX-RS*

  JAX-RS repose sur le conteneur Web Java EE pour rendre les services accessibles en HTTP.

interface uniforme (uniform interface)

Les composants d'une architecture REST doivent tous utiliser la même interface uniforme et restreinte. Limiter l'interface permet de conserver une architecture simplifiée (plus simple à comprendre et plus simple à faire évoluer).

- *Web*

  L'interface est contrainte par la méthode HTTP, l'URI et le content-type

- *JAX-RS*

  Associe une méthode Java à une méthode HTTP (@GET, @POST, ...), au chemin de ressource (@Path) et au content-type de la requête (@Consumes) et de la réponse (@Produces).

sans état (stateless)

Une requête doit contenir l'ensemble de l'information nécessaire à son traitement. Chaque requête est donc "sans mémoire" (indépendante de l'ordre des requêtes précédentes). Les informations de session sont conservées uniquement par l'application cliente.

Le respect de cette contrainte permet de construire des architectures plus sûres et plus scalables.

- *Web*

  Il existe des techniques très largement répandues qui enfreignent cette contrainte : utilisation des cookies pour identifier une session utilisateur temporaire sur le serveur.

- *JAX-RS*

  JAX-RS est une API sans état. Elle n'inclue pas de mécanisme permettant d'implémenter une session cliente côté serveur. Il est néanmoins possible de développer des applications Web Java EE avec état (stateful).

mise en cache

La mise en cache est nécessaire pour améliorer l'efficacité des échanges réseaux. Cela implique que la représentation d'une ressource doit pouvoir indiquée si elle peut être mise en cache par le client ou un système intermédiaire.

- *Web*

  Le Web dispose de multiples composants pour la mise en cache (navigateurs Web, proxies...). De plus HTTP supporte la définition des stratégies de cache et les requêtes conditionnelles grâce aux en-têtes **Cache-control (http://fr.wikipedia.org/wiki/Cache-Control)** , **ETag (http://en.wikipedia.org/wiki/HTTP_ETag)** , **If-None-Match (http://en.wikipedia.org/wiki/HTTP_ETag)** , If-Modified-Since.

- *JAX-RS*

  JAX-RS intègre l'évaluation des préconditions d'une requête HTTP grâce aux méthodes **Request.evaluatePreconditions() (http://docs.oracle.com/javaee/6/api/javax/ws/rs/core/Request.html)**

Gestion des préconditions avec JAX-RS

```
@Path("/bill/{ref}")
@GET
public Response getBill(@PathParam("ref") String ref, @Context Request req){

  Bill bill = billRepository.get(ref);

  EntityTag etag = new EntityTag(Integer.toString(bill.hashCode()));

  ResponseBuilder builder = req.evaluatePreconditions(etag);

  // Si la précondition échoue, builder sera null
  if(builder == null){
    // L'etag n'est pas celui transmis par le client
    // (ou le client n'en a pas fourni), il faut retourner les données.
    builder = Response.ok(bill);

    // On retourne l'Etag au client
    builder.tag(etag);
  }

  return builder.build();
}
```

layered system

Une architecture REST doit être stratifiée en couches hiérarchiques. Un composant ne peut dialoguer qu'avec ses voisins immédiats, c'est-à-dire les composants appartenant aux couches adjacentes. Cette contrainte permet d'éviter un système centralisé et favorise les techniques de mise en cache et de load balancing.

- *Web*

  Le Web est largement conçu selon cette contrainte. Des composants spécialisés comme les proxy et les firewalls permettent de renforcer une architecture en couche.

- *JAX-RS*

  Cette contrainte n'a pas de répercussion sur l'API JAX-RS.

code-on-demand

Il s'agit d'une contrainte optionnelle de l'architecture REST. Un client peut étendre ses fonctionnalités en téléchargeant du code exécutable.

- *Web*

  Le code-on-demand est trés largement utilisé dans le Web pour les humains avec en particulier le support de JavaScript pour étendre les fonctionnalités du navigateur Web.

- *JAX-RS*

  Cette contrainte n'est présente pas dans l'API JAX-RS. Cependant, elle peut être implémentée facilement grâce au **MessageBodyWriter (http://docs.oracle.com/javaee/6/api/javax/ws/rs/ext/MessageBodyWriter.html)** .

HATEOAS

Dans sa thèse, Roy Fielding cite une contrainte spécifique à l'interface uniforme :

> *"hypermedia as the engine of application state"*

Cette contrainte est couramment abrégée en HATEOAS. Mais que signifie-t-elle ?

Un service Web RESTful n'est pas qu'un ensemble d'URI permettant de manipuler des représentations de ressources. Les représentations de ressources sont des *hypermédia* : elles contiennent également des liens (URI) vers d'autres ressources.

Leonard Richardson utilise le terme *connexité* (plutôt que HATEOAS) pour décrire cette particularité des services Web RESTful.

L'implémentation de la connexité peut se faire :

- dans l'en-tête de la réponse grâce à l'en-tête *Link*.
- dans la représentation de la ressource elle-même.

L'en-tête **Link (http://tools.ietf.org/html/rfc5988)** permet de spécifier une URI et des paramètres spécifiant le lien.

```
Link: [URI]; [paramètres]
```

Le paramètre *rel* permet de définir le type de relation.

Le paramètre *title* permet de définir le titre du lien pour afficher à un utilisateur.

Un exemple de lien

```
Link: http://epsi.fr/i04/webservices; rel="cours"; title="Web services"
```

La spécification d'un lien dans la représentation dépend du format des données. Certains formats proposent des façons plus ou moins normalisées de spécifier des liens.

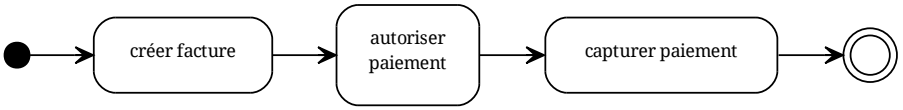
- HTML dispose des balises <link> <a> <form>
- ATOM dispose de la balise <link>
- XML dispose de l'extension **XLink (http://www.yoyodesign.org/doc/w3c/xlink/index.html)**
- JSON dispose de l'extension **HAL (http://stateless.co/hal_specification.html)** (Hypertext Application Language)

Les avantages de la connexité :

- Le service Web s'auto-décrit et nécessite une documentation minimale
- Le client n'a pas besoin de connaître les règles de construction des URI. Il est plus robuste pour supporter les éventuelles évolutions du service.
- L'implémentation de services sans état est rendue plus simple : l'état du client se traduit par les liens suivis plutôt que par une session stockée sur le serveur.

EXEMPLE : PAIEMENT DE FACTURES

Activité d'un système de paiement de factures



Comment modéliser un système de paiement de factures RESTful ?

Création d'une facture

```
PUT /utilisateur/moi/facture/fact-2013005689 HTTP/1.1
Host: facture-en-ligne.fr
Content-type: application/x-www-form-urlencoded; charset=utf-8
[...]

montant=2100&client=MonClient
```

```
HTTP/1.1 201 CREATED
[...]
```

Consultation d'une facture (XML avec négociation de contenu)

```
GET /utilisateur/moi/facture/fact-2013005689 HTTP/1.1
Host: facture-en-ligne.fr
Accept: application/xml
[...]
```

```
HTTP/1.1 200 OK
Content-type: application/xml; charset=utf-8
Link: http://facture-en-ligne.fr/utilisateur/moi/facture/fact-2013005689; rel="modification"
Link: http://facture-en-ligne.fr/utilisateur/moi/facture/fact-2013005689/autorisation; rel="autorisation"
[...]

<?xml version="1.0" ?>
<facture>
  <reference>fact-2013005689</reference>
  <client>MonClient</client>
  <montant>2300</montant>
  <date>2013-11-01T18:30:10Z</date>
  <link href="http://facture-en-ligne.fr/utilisateur/moi/facture/fact-2013005689" method="PUT" rel="modification"/>
  <link href="http://facture-en-ligne.fr/utilisateur/moi/facture/fact-2013005689/autorisation" method="POST" rel="autorisation"/>
</facture>
```

Consultation d'une facture (JSON avec négociation de contenu)

```
GET /utilisateur/moi/facture/fact-2013005689 HTTP/1.1
Host: facture-en-ligne.fr
Accept: application/json
[...]
```

```
HTTP/1.1 200 OK
Content-type: application/json; charset=utf-8
Link: http://facture-en-ligne.fr/utilisateur/moi/facture/fact-2013005689; rel="modification"
Link: http://facture-en-ligne.fr/utilisateur/moi/facture/fact-2013005689/autorisation; rel="autorisation"
[...]

{'reference':'fact-2013005689',
 'montant':2300,
 'client':'MonClient',
 'date':'2013-11-01T18:30:10Z',
 'link': [
   {'href':'http://facture-en-ligne.fr/utilisateur/moi/facture/fact-2013005689',
     'method':'PUT',
     'rel':'modification'},
   {'href':'http://facture-en-ligne.fr/utilisateur/moi/facture/fact-2013005689/autorisation',
     'method':'POST',
     'rel':'autorisation'}
  ]
 }
```

Création d'une autorisation de paiement

```
POST /utilisateur/moi/facture/fact-2013005689/autorisation HTTP/1.1
Host: facture-en-ligne.fr
Content-type: application/json
[...]

{'cardOwner':'',
 'cardNumber':'1111222233334444',
 'cardExpiryDate':'122014',
 'cardSecurityNumber':'123'}
```

```
HTTP/1.1 201 CREATED
Location: http://facture-en-ligne.fr/utilisateur/moi/facture/fact-2013005689/autorisation/3F2504E0
[...]
```

Modification d'une facture

```
PUT /utilisateur/moi/facture/fact-2013005689 HTTP/1.1
Host: facture-en-ligne.fr
Content-type: application/x-www-form-urlencoded; charset=utf-8
[...]

montant=2300&client=MonClient
```

```
HTTP/1.1 200 OK
[...]
```

Modification d'une facture après autorisation

```
PUT /utilisateur/moi/facture/fact-2013005689 HTTP/1.1
Host: facture-en-ligne.fr
Content-type: application/x-www-form-urlencoded; charset=utf-8
[...]

montant=2312&client=MonClient
```

```
HTTP/1.1 409 CONFLICT
Content-type: plain/text; charset=utf-8
[...]

La facture fact-2013005689 ne peut pas être modifiée car une autorisation
de paiement a déjà été effectuée avec succès.
```

En fonction de l'état des resources, certaines opérations peuvent ne plus être possibles et entraîner un rejet pour conflit.

Consultation d'une facture après autorisation

```
GET /utilisateur/moi/facture/fact-2013005689 HTTP/1.1
Host: facture-en-ligne.fr
Accept: application/xml
[...]
```

```
HTTP/1.1 200 OK
Content-type: application/xml; charset=utf-8
Link: http://facture-en-ligne.fr/utilisateur/moi/facture/fact-2013005689/capture; rel="capture"
[...]

<?xml version="1.0" ?>
<facture>
  <reference>fact-2013005689</reference>
  <client>MonClient</client>
  <montant>2300</montant>
  <date>2013-11-01T18:30:10Z</date>
  <link href="http://facture-en-ligne.fr/utilisateur/moi/facture/fact-2013005689/capture" method="POST" rel="capture"/>
</facture>
```

Après la création de l'autorisation, la représentation de la facture ne contient plus les liens de modification et d'autorisation. Par contre, le lien pour poster une capture (c'est-à-dire la validation du paiement) est présent.

Ceci est une illustration de la connexité entre les ressources. Elle permet de décrire un workflow et guide le client dans l'utilisation du service.

RÉFÉRENCES REST ET SERVICES WEB RESTFUL

REST cookbook
http://restcookbook.com/

Une étude de cas très complète (commander un café en REST)
http://www.infoq.com/articles/webber-rest-workflow

Le modèle de maturité de Richardson des web services RESTful
http://martinfowler.com/articles/richardsonMaturityModel.html

La thèse de Roy Fielding incluant la présentation de REST
http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

RÉFÉRENCES JAX-RS

La spécification JAX-RS 2.0 en téléchargement
http://download.oracle.com/otndocs/jcp/jaxrs-2_0-fr-eval-spec/index.html

Tutoriels Java EE pour JAX-RS
http://docs.oracle.com/javaee/6/tutorial/doc/giepu.html

Résumé des principales annotations JAX-RS
http://www.techferry.com/articles/RESTful-web-services-JAX-RS-annotations.html

La documentation de Jersey (implémentation de référence de JAX-RS)
https://jersey.java.net/documentation/latest/

LIVRES

RESTful Web Services
Leonard Richardson & Sam Ruby - O'Reilly 2007

Java EE 7 Essentials
Arun Gupta - O'Reilly 2013

ANNEXES

- Services Web RESTful et Ajax
- Sécurité et navigateur Web : JSON-P et COTS
- Utilisation avancée de JAX-RS : l'annotation **@Context (http://docs.oracle.com/javaee/6/api/javax/ws/rs/core/Context.html)**