

# WEB SERVICES :

## REST / RESTFUL

# Références

## □ Sources :

- <http://olivier.coupelon.net/t/cours-webservices>
- Michel Buffa, MIAAGE de Nice
- Blogs OCTO/Ippon/Xebia/...

□ dzone.com

□ json.org

□ Les cast codeurs

□ Twitter

□ Jwt.io

□ .....

□ Jboss, Glassfish, RestEasy, Jersey, Spring.io ...

- REST : introduction
- Pique de « rappel » HTTP
- RESTful et HTTP
- API design
- Programmation REST en Java

# Communications inter-applications

Plusieurs **niveaux** de communication :

- API : on embarque directement l'application à appeler (dépendance directe, eg JAR dans pom)
- RPC/RMI/Corba : on embarque les interfaces des services
- EJB : on appelle l'application avec un protocole de niveau transport (TCP/IP, RMI) ou directe (si même JVM)
- REST : échange niveau HTTP, on exploite à 100% le protocole
- SOAP : au dessus d'HTTP ou SMTP ou ..., on encapsule les messages



# Commençons par la fin !

REST, what else !

Principe simple → Exploitation de la mécanique HTTP

Inspiration pour définir l'« architecture » d'accès (interface) à une application

## Définition

- ❑ Acronyme de **RE**presentational **S**tate **T**ransfert défini dans la thèse de Roy Fielding en 2000 :

<http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>

- ❑ REST n'est pas un protocole ou un format, contrairement à SOAP, HTTP ou RCP, mais un **style** d'architecture inspiré de l'architecture du web fortement basé sur le protocole HTTP
- ❑ Il n'est pas dépendant uniquement du web et peut utiliser d'autre protocoles que HTTP

Ce qu'il est :

- Un système d'architecture
- Une approche pour construire une application

Ce qu'il n'est pas

- Un protocole
- Un format
- Un standard



- Utiliser dans le développement des applications orientés ressources (ROA) ou orientées données (DOA)
- Les applications respectant l'architecture REST sont dites RESTful
- Attention, REST et RESTful sont devenus des termes marketing ! Certains services Web se réclamant de REST ne le sont pas (eg utilisent le protocole HTTP de manière un peu plus conventionnelle) => Utilisation du terme HATEOAS, [Hypermedia as the Engine of Application State]

## REST - qq Fournisseurs

facebook



Google



# Utilisation de REST

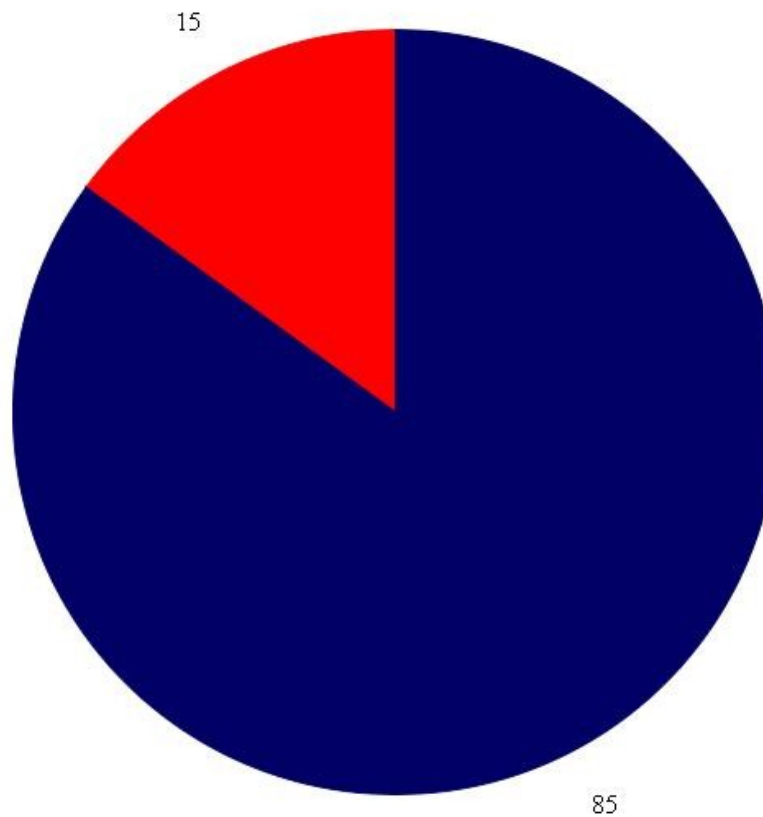
Paul Downey, BT



# REST - Statistiques

## Statistique d'utilisation des services web REST et SOAP chez AMAZON

■ REST ■ SOAP

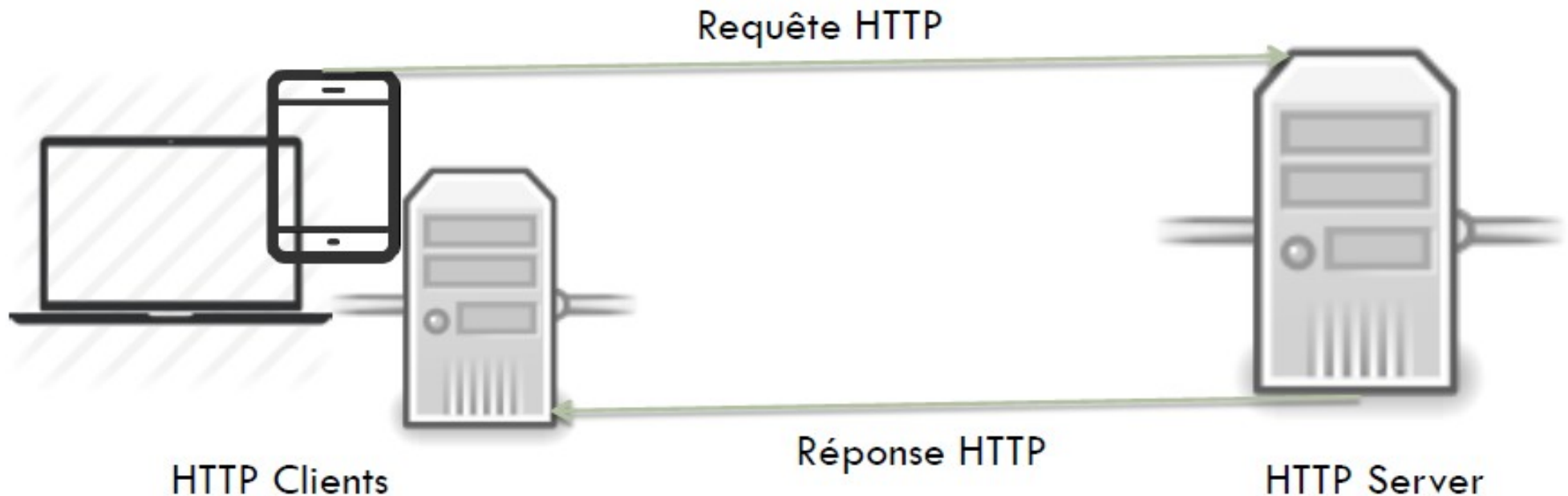




# Protocole HTTP

Piqûre de rappel sur le protocole

- HyperText Transfer Protocol
- Protocole d'échanges d'information sur le web
- Basé sur TCP
- Version courante 1.1 [2.0]



# URL

- Unique Resource Location
- Identifie les ressources de manière unique sur le Web
- 5 parties
  - Protocole (http, ftp, mail, ...)
  - Host (google.com)
  - Port (8080, 80)
  - Path (Chemin vers la ressource sur le serveur)
  - Paramètres

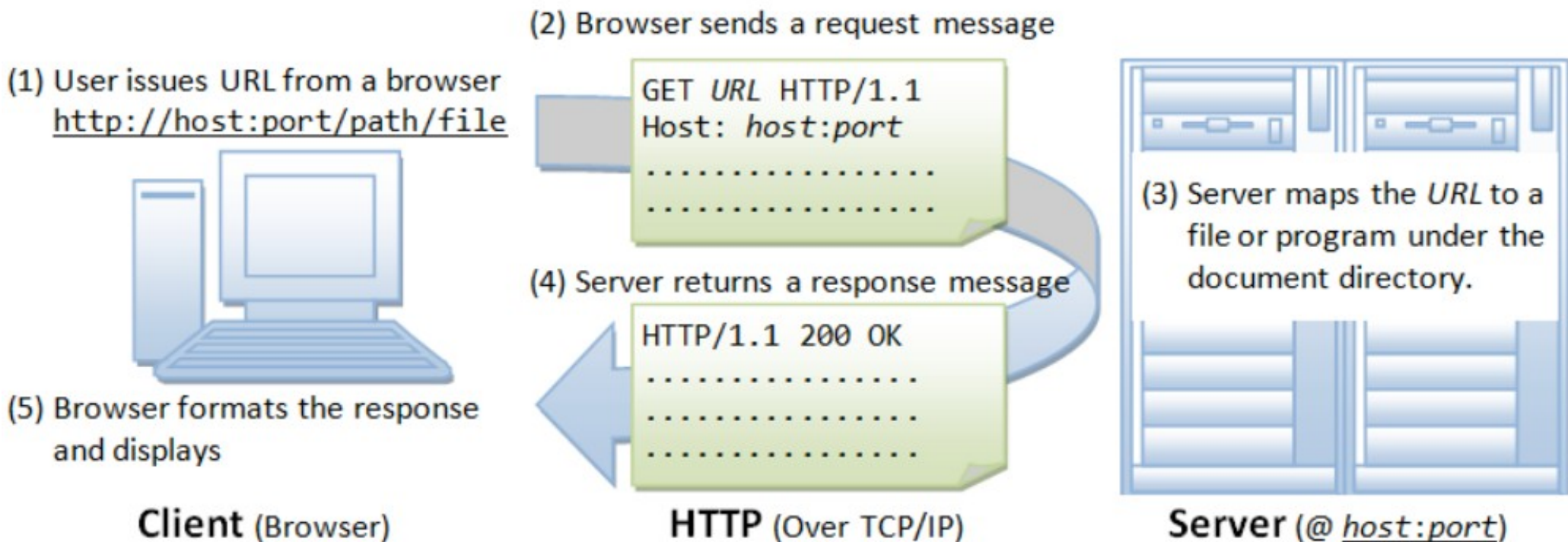
**http://server:port/path/to/resource?p1=v1&p2=v2**

The diagram illustrates the structure of the URL **http://server:port/path/to/resource?p1=v1&p2=v2** by grouping its parts into four categories with curly braces underneath:

- Protocole**: Points to the **http** part of the URL.
- Nom du serveur et port**: Points to the **server:port** part of the URL.
- Chemin de la ressource**: Points to the **/path/to/resource** part of the URL.
- Paramètres**: Points to the **?p1=v1&p2=v2** part of the URL.



# Enchainements Client - Serveur

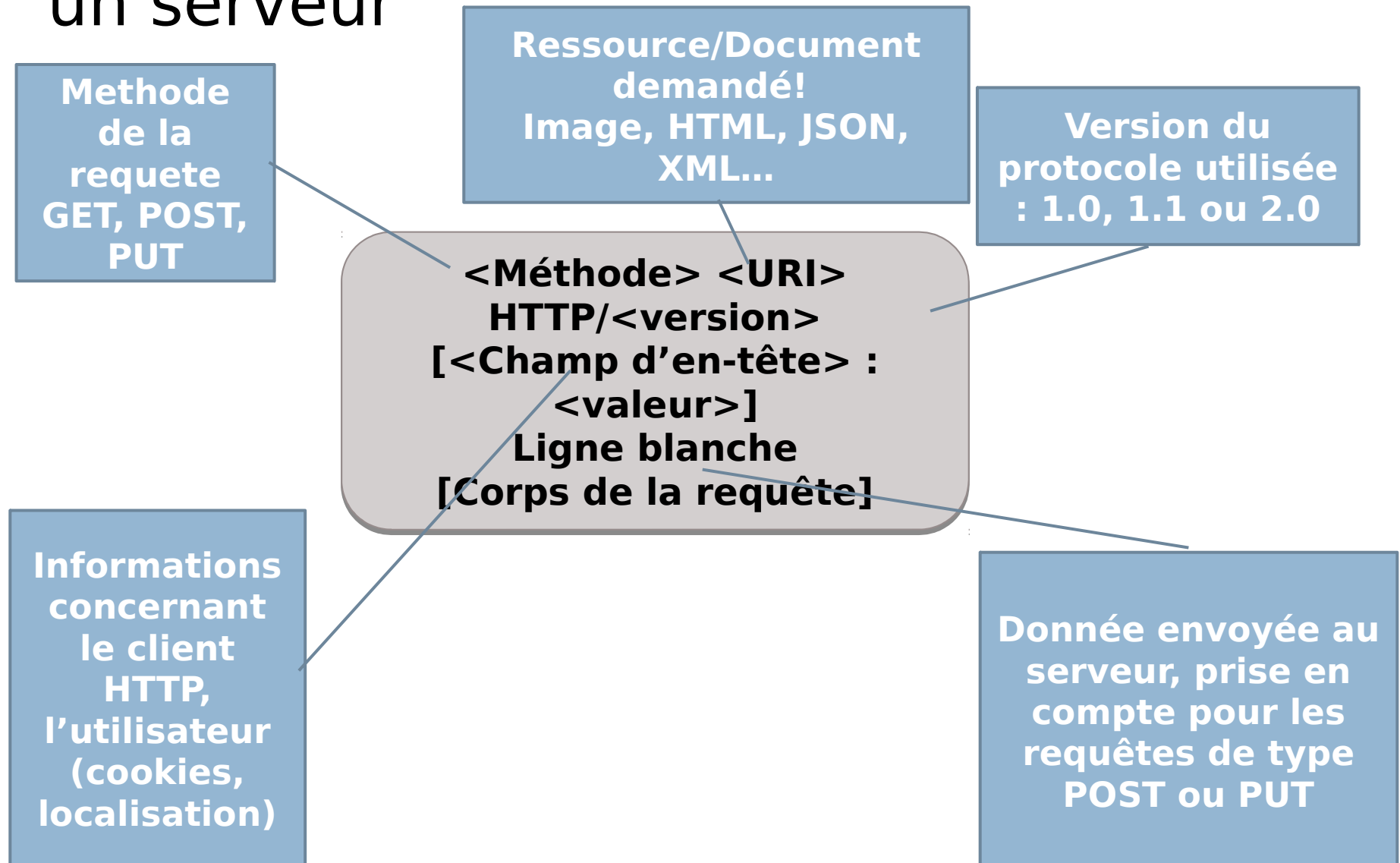




- Permet à un client d'envoyer des messages à un serveur
- Format d'un message HTTP
  - Request Message Header
    - Request Line
    - Request Headers [Optional]
  - Request Message Body

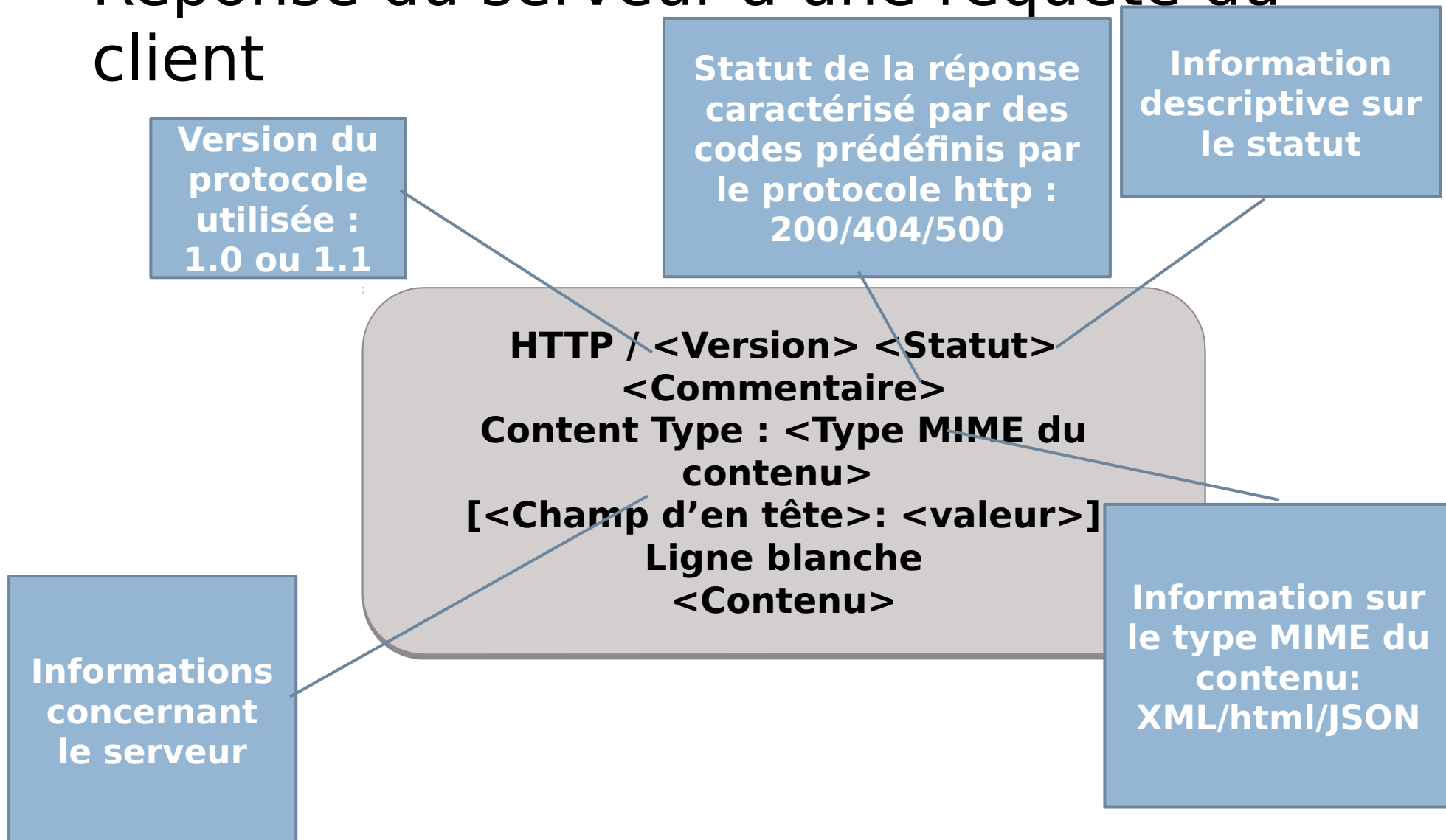
# Requête HTTP

- Requête envoyée par un client http vers un serveur



- Réponse du serveur au client :
- Response Message Header
  - Response Line
  - Response Headers
- Response Message [Optional]

## □ Réponse du serveur à une requête du client



### Démo

- docker ! <https://www.docker.com/>
- nginx
- telnet
- protocole http

# Autre exemple : Request

## □ Request Line

POST /bibliotheque/faces/views/categorie/Create.xhtml HTTP/1.1

## □ Request Headers

Host: localhost:8080

Connection: keep-alive

Content-Length: 176

Cache-Control: max-age=0

Accept:

text/html,application/xhtml+xml,application/xml;q=0.9,image/webp, \*/  
\*;q=0.8

Origin: http://localhost:8080

User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_10\_0)

AppleWebKit/537.36 (KHTML, like Gecko) Chrome/  
39.0.2171.65 Safari/537.36

Content-Type: application/x-www-form-urlencoded

Referer:

http://localhost:8080/bibliotheque/faces/views/categorie/List.xhtml

Accept-Encoding: gzip, deflate

Accept-Language: fr,fr-FR;q=0.8,en;q=0.6

Cookie: JSESSIONID=d64a9484e61761662575b5d14af1

## Autre exemple : Request

- Request Message Body

j\_idt13:nom:Toto

j\_idt13:description:Hello

## Response Message Header

- Response Line

HTTP/1.1 200 OK

- Response Headers

HTTP/1.1 200 OK

X-Powered-By: Servlet/3.1 JSP/2.3 (GlassFish  
Server Open Source Edition 4.0 Java/Oracle  
Corporation/1.8)

Server: GlassFish Server Open Source Edition  
4.0

Content-Type: text/html; charset=UTF-8

Date: Sun, 23 Nov 2014 16:05:39 GMT

Content-Length: 2274



# Response Message Body

## □ Response Body

```
<html xmlns="http://www.w3.org/1999/xhtml"><html
  xmlns="http://www.w3.org/1999/xhtml">
<head><link
type="text/css" rel="stylesheet"
  href="/bibliotheque/faces/javafx.faces.resource/theme.css?
  ln=primefaces-aristo" />
<link type="text/css" rel="stylesheet"
  href="/bibliotheque/faces/javafx.faces.resource/css/jsfcrud.css" />
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Create New Categorie</title></head><body>
<h1>Create New Categorie
</h1>
<p><div id="messagePanel"><table><tr style="color: green"><td>Categorie
  was successfully created. </td>
</tr></table></div>
</html>
```

- Une fiche mémo :

<http://authoritylabs.com/blog/common-http-response-codes-and-what-they-mean/>

## HTTP Status Codes Cheat Sheet

✓ Successful Requests		🔄 Redirects	
200 OK	Request was successful.	300 Multiple Choices	Response indicating that what was requested has moved or that there are multiple options that match the request.
201 Created	Request was successful and something new was created based on that request.	301 Moved Permanently	Requested document was moved permanently and response contains the URI for that new location. Important to use when changing domains names or URLs of existing documents.
202 Accepted	Request was received successfully, but may not be acted on immediately.	302 Found	Requested document was temporarily moved to a different location. The URI of the new location is sent back with the response. 303 and 307 are more specific.
203 Non-authoritative Information	Request was successful, but information sent to the client about the response comes from a 3rd party server.		
204 No Content	Request was successful, but no data is sent back.		
205 Reset Content	Request from the server to reset the information sent.		

- Doivent être interprétés correctement par le **client**, et émis correctement par le serveur !!!

- HTTP définit un ensemble de méthodes qui permet de caractériser les types de requêtes
  - GET : Récupérer des ressources à un serveur
  - POST : Envoyer des données à un serveur
  - PUT/PATCH : Modifier des données
  - DELETE : Suppression de données
  - OPTIONS : Demander la liste des méthodes supportées par un serveur
  - Autres : HEAD, TRACE, CONNECT



Et REST dans tout ça ?

# Communication REST



- Les services REST sont sans états (Stateless)
  - Chaque requête envoyée au serveur doit contenir toutes les informations relatives à son état et est traitée indépendamment de toutes autres requêtes
  - Minimisation des ressources systèmes (pas de gestion de session, ni d'état)
- Interface uniforme basée sur les méthodes HTTP (GET, POST, PUT, DELETE en général)
- Les architectures RESTful sont construites à partir de ressources uniquement identifiées par des URI(s)

### □ Ressources

- Identifiée par une URI

<http://univ-orleans.fr/cursus/master/miage/sir/2>

### □ Méthodes (verbes) permettant de manipuler les ressources (identifiants)

- Méthodes limitées strictement à HTTP : GET, POST, PUT/PATCH, DELETE

### □ Représentation : Vue sur l'état de la ressource

- Format d'échanges entre le client et le serveur (XML, JSON, text/plain,...)

# Ressources

- Une ressource est un objet identifiable sur le système

→ Livre, Catégorie, Client, Prêt

Une ressource n'est pas forcément un objet matérialisé (Prêt, Consultation, Facture...)

- Une ressource est identifiée par une URI :  
Une URI identifie uniquement une ressource sur le système (une ressource peut avoir plusieurs identifiants)




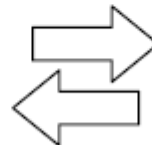
→ <http://amazon.fr/bookstore/books/2934>

Clef primaire de  
la ressource  
dans la BD



- Une ressource peut subir quatre opérations de bases CRUD correspondant aux quatre principaux types de requêtes HTTP (GET, PUT, POST, DELETE)
- REST s'appuie sur le protocole HTTP pour effectuer ces opérations sur les objets
  - CREATE      POST
  - RETRIEVE    GET
  - UPDATE      PUT/PATCH
  - DELETE      DELETE

# CRUD

CRUD	REST		
CREATE	POST		Create a sub resource
READ	GET		Retrieve the current state of the resource
UPDATE	PUT		Initialize or update the state of a resource at the given URI
DELETE	DELETE		Clear a resource, after the URI is no longer valid

# Méthode GET

- La méthode GET renvoie une représentation de la ressource tel qu'elle est sur le système

GET `http://amazon.fr/bookstore/books/156`



Client



Statut : 200

Message : OK

En-tête : ...

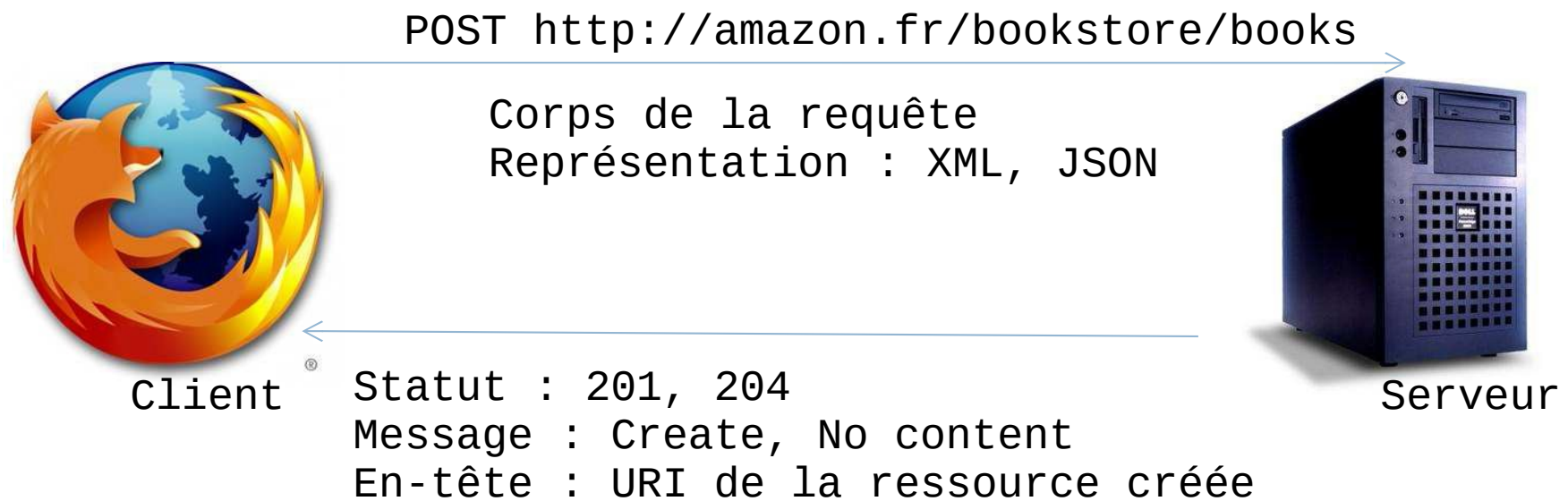
Représentation : XML, JSON, html,...



Serveur

# Méthode POST

- La méthode POST crée une nouvelle ressource sur le système



# Méthode DELETE

- Supprime la ressource identifiée par l'URI sur le serveur

Identifiant de la ressource sur le serveur

DELETE http://amazon.fr/bookstore/books/156



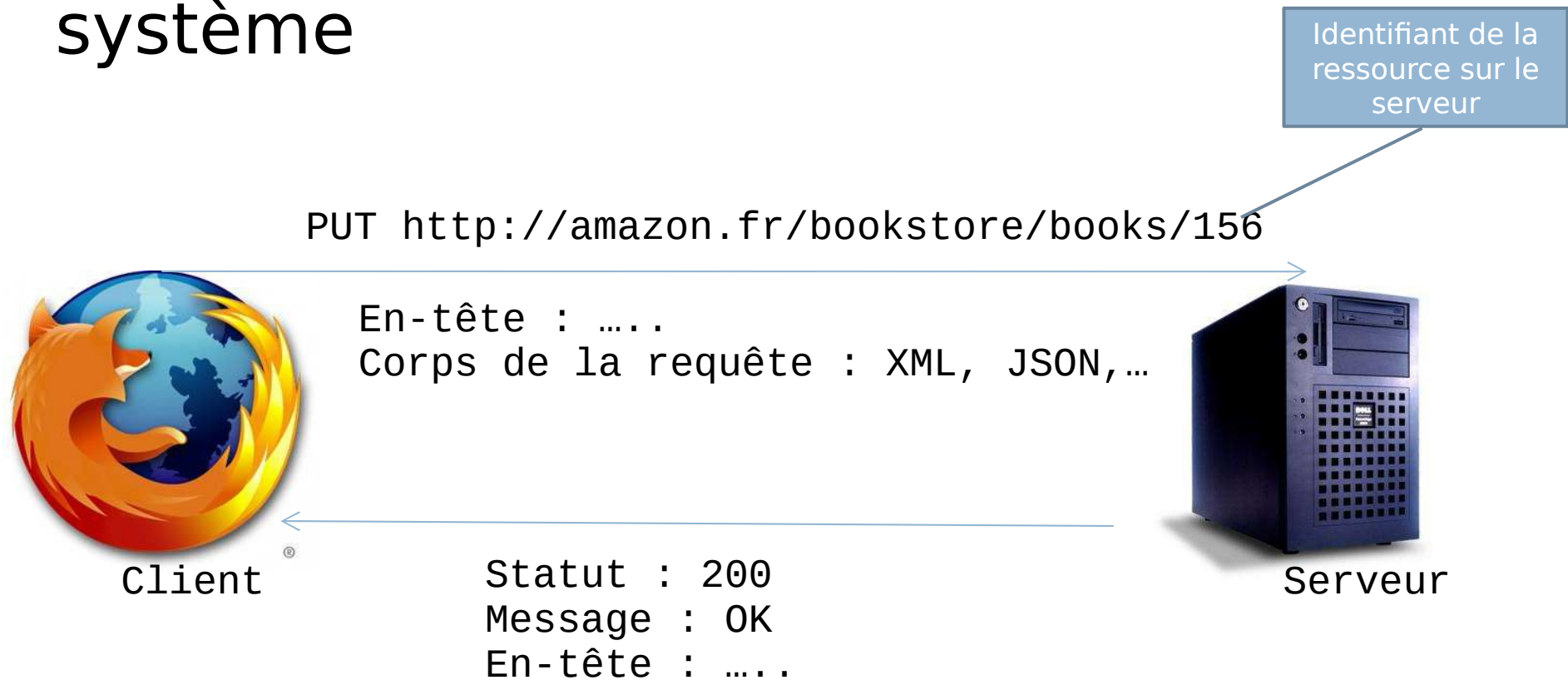
Client



Serveur

Statut : 200  
Message : OK  
En-tête : ....

## □ Mise à jour de la ressource sur le système



# Représentation

Une représentation désigne les données échangées entre le client et le serveur pour une ressource:

- HTTP GET : Le serveur renvoie au client l'état de la ressource
- PUT, POST : Le client envoie l'état d'une ressource au serveur

Les données peuvent être sous différents formats :

- JSON
- XML
- XHTML
- CSV
- Text/plain
- .....

- Contrôle **total** des URI et des contenus
- Le bon dev il dev et le mauvais dev il dev aussi
- Définition d'une API : effet coulée de lave !
  - La coulée de lave se produit lorsqu'une partie de code encore immature est mise en production, forçant la lave à avancer et en empêchant sa modification





URLs propres ?

<http://map.search.ch/lugano>



<http://maps.google.com/maps?f=q&hl=en&q=lugano,+switzerland&layer=&ie=UTF8&z=12&om=1&iwloc=addr>



# Formats d'échange de données

JSON et ses amis

JSON « **J**ava**S**cript **O**bject **N**otation » est un format d'échange de données, facile à lire par un humain et interpréter par une machine.

Basé sur JavaScript, il est complètement indépendant des langages de programmation mais utilise des conventions qui sont communes à toutes les langages de programmation (C, C++, Perl, Python, Java, C#, VB, JavaScript,....)

Deux structures :

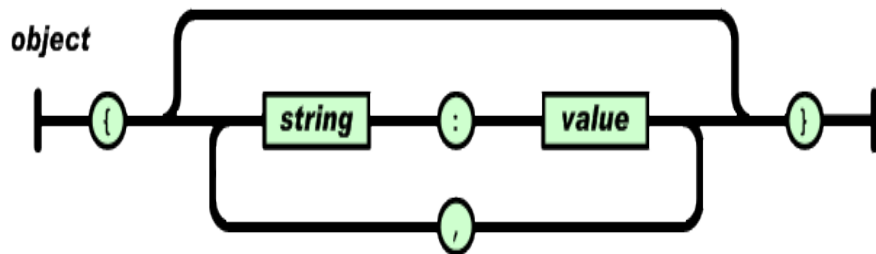
- Une collection de clefs/valeurs □ Object
- Une collection ordonnée d'objets □ Array

# Exemple de JSON

```
{ "menu": {  
  "id": "file",  
  "value": "File",  
  "popup": {  
    "menuitem": [  
      { "value": "New", "onclick": "CreateNewDoc() " },  
      { "value": "Open", "onclick": "OpenDoc() " },  
      { "value": "Close", "onclick": "CloseDoc() " }  
    ]  
  }  
}}
```

## Objet

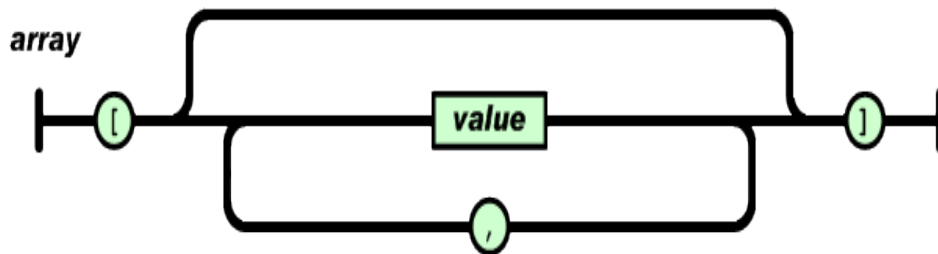
Commence par un « { » et se termine par « } » et composé d'une liste non ordonnée de paire clefs/valeurs. Une clef est suivie de « : » et les paires clef/valeur sont séparés par « , »



```
{ "id": 51,
  "nom": "Mathematiques 1", "resume":
  "Resume of math ", "isbn": "123654",
  "categorie":
    {
      "id": 2, "nom": "Mathematiques",
      "description": "Description of
      mathematiques "
    },
  "quantite": 42,
  "photo": ""
}
```

## ARRAY

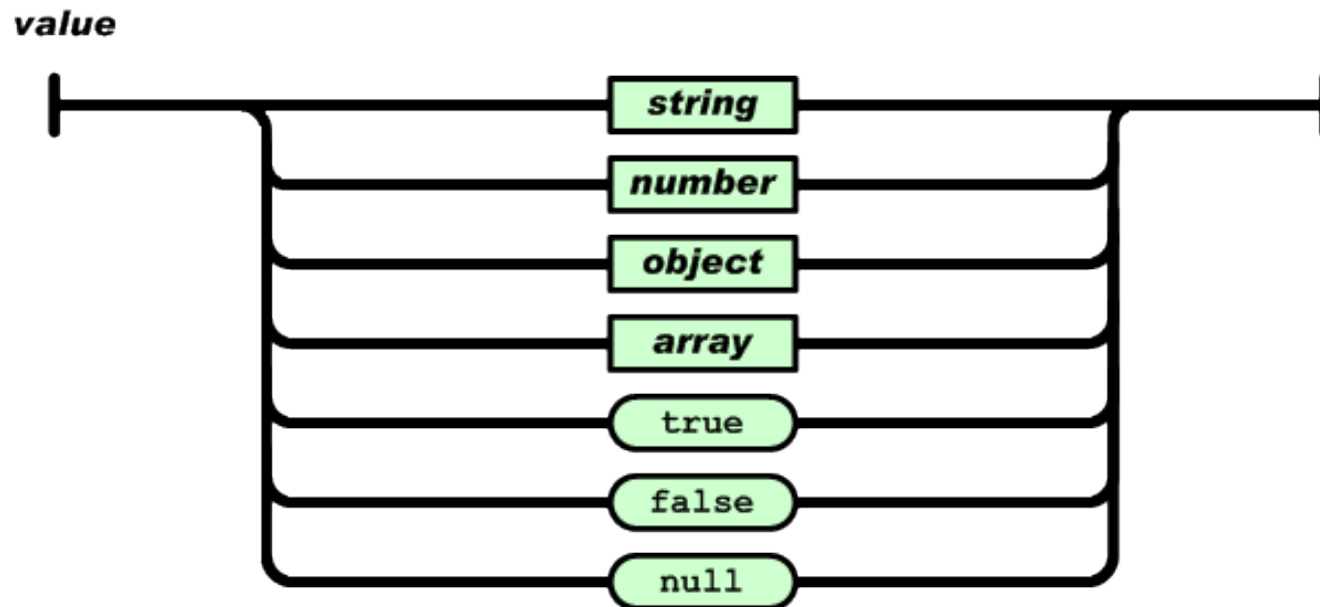
Liste ordonnée d'objets commençant par « [ » et se terminant par « ] », les objets sont séparés l'un de l'autre



```
{ "id": 51,  
  "nom": "Mathematiques 1",  
  "resume": "Resume of math ",  
  "isbn": "123654",  
  "quantite": 42,  
  "photo": ""  
},  
{ "id": 102,  
  "nom": "Mathematiques 1",  
  "resume": "Resume of math ",  
  "isbn": "12365444455",  
  "quantite": 42,  
  "photo": ""  
}  
]
```

## Value

Un objet peut être soit un string entre « "" » ou un nombre (entier, décimal) ou un boolean (true, false) ou null ou un objet.



- JSON est une structure de donnée initialement proposée en JavaScript
- Pour l'interpréter, on utilisait la fonction JavaScript `eval()`, dangereuse :  

```
var donnees =  
eval( '(' + donnees_json + ')' );
```
- Natif dans les navigateurs depuis 2009 (IE8, Firefox 3.5, Opera 10.5) :  

```
var donnees = JSON.parse(donnees_json);
```



- XML est standard
- Manipulation aisée
- Verbeux
- Schéma XSD !
- JSON est crée pour les navigateurs web (JS !)
- Interprété nativement, donc performant
- Supporté par la plupart des langages moyennant une API spécifique (Jackson, JAXB, ...) eg [JSON](#)

Pour les deux, cf RefCardz sur [dzone.com](http://dzone.com)



URIs/datas par l'exemple

## Exemple

- Sondages des questions et plusieurs réponses
- Toutes les opérations CRUD sur un sondage
- Sur chaque sondage des votes, avec une réponse
- Toutes les opérations CRUD sur un vote
- Questions :
  - proposer une interface EJB
  - proposer une interface REST

## Exemple : Création d'un sondage

**POST /sondage**

```
<question>Lequel  
?</question>  
<options>A, B, C</options>
```

**201 Created**

Location:  
/sondage/672609683

**GET /sondage/672609683**

**200 OK**

```
<question>Lequel  
?</question>  
<options>A, B, C</options>
```

## Exemple : Vote au sondage

POST /sondage/672609683/vote

<name>J. Martins</name>

<choice>B</choice>

201 Created

Location: /sondage/672609683/vote/1

GET /sondage/672609683

200 OK

<question>Lequel ?</question>

<options>A,B,C</options>

<votes>

<vote id='1'>

<name>J. Martins</name>

<choice>B</choice>

</vote>

</votes>

## Exemple : Modification d'un sondage

PUT  
/sondage/672609683/vote/1

<name>J. Martins</name>

<choice>C</choice>

200 OK

GET /sondage/672609683

200 OK

<options>A,B,C</options>

<votes>

<vote id='1'>

<name>J. Martins</name>

<choice>C</choice>

</vote>

</votes>

## Exemple : Suppression d'un sondage

```
DELETE  
/sondage/672609683
```

```
200 OK
```

```
GET /sondage/672609683
```

```
404 Not Found
```

Je suis maître du monde !



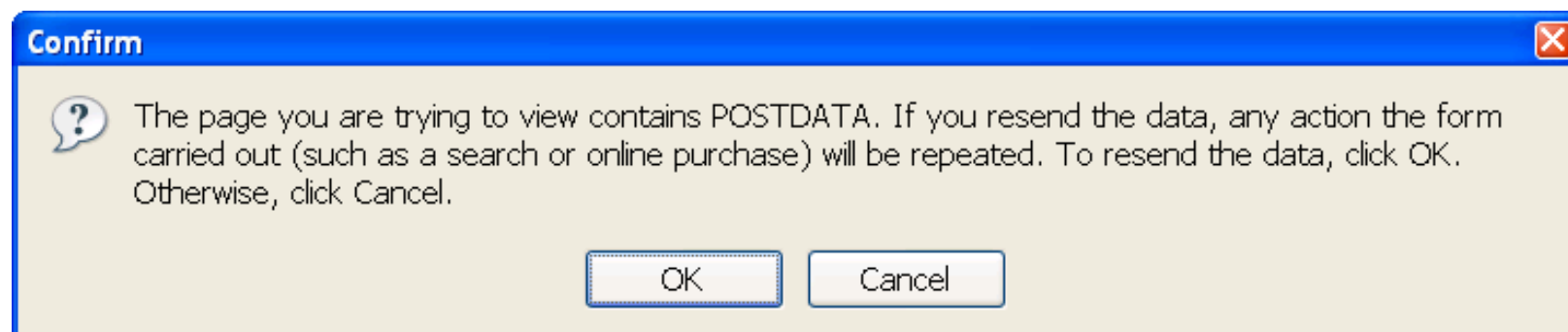


- Utiliser le chemin de la ressource pour spécifier les paramètres
- Une fois définie, une URL ne doit plus changer
- REST utilise des URL opaques :  
L'accès à une nouvelle ressource ne doit pas être construite par le client, mais transmise par le serveur
- UTILISER les bons codes HTTP de retour

1. Identification des ressources
2. Définition d'url propres
3. Pour chaque url, prévoir les actions pour GET, POST, PUT & DELETE
4. Documenter chaque ressource
5. Définir les liens entre ressources
6. Déployer et tester chaque ressources sur un serveur « http »

- Operation de lecture
- Idempotente
- Possibilité de cache

- Operation de création
- Effet de bord possible sur le serveur



- Identifier une ressource créée

- Préferer

POST /sondage/672609683/vote  
201 CREATED

Location: /sondage/672609683/vote/1

à

PUT /sondage/672609683/vote/1  
200 OK

- On ~~peut~~ **DOIT** ajouter en entête dans la requête le type de contenu qu'on souhaite recevoir, par ordre de référence :
- GET /resource
- Accept: application/xml, application/json
- 200 OK
- Content-Type: application/json

- API versions should be mandatory. This way, you will be futureproof as the API changes through time. One way to do is to pass the API version in the URL (/api/v1/...)
- One other neat trick is to use the Accept HTTP header to pass the version desired.  
[Github does that](#)
- Using versions will allow you to change the API structure without breaking compatibility with older clients
- Cf exemple application...



# Des noms tu utiliseras, jamais de verbes

- One thing I often see is people using verbs instead of nouns in their resources name. Here are some bad examples :
  - /getProducts
  - /listOrders
  - /retreiveClientByOrder?orderId=1
- For a clean and concise structure, you should *always* use nouns. Moreover, a good use of HTTP methods will allow you to remove actions from your resources names.
  - GET /products : will return the list of all products
  - POST /products : will add a product to the collectionn
  - GET /products/4 : will retreive product #4
  - PATCH/PUT /products/4 : will update product #





- [RFC2616](#) clearly states that HEAD and GET methods should always be safe to call (in other words, the state should not be altered)
- Here is a bad example : GET /deleteProduct?id=1
- Imagine a search engine indexing that page...



- If you want to get a sub collection (collection of an other one), use nested routing for a cleaner design
- For instance, if you want to get a list of all albums of a particular artist, you would want to use GET /artists/8/albums



## Des résultats paginés tu renverras

- Returning a very large resultset over HTTP is not a very good idea neither. You will eventually run into performance issues as serializing a large JSON may quickly become expensive.
- An option to get around that would be to paginate your results. Facebook, Twitter, Github, etc. does that. It is much more efficient to make more calls that takes little time to complete, than a big one that is very slow to execute.
- Also, if you are using pagination, one good way to indicate the next and previous pages links is through the Link HTTP header.



# Les bons codes HTTP tu respecteras

- Always use proper HTTP status codes when returning content (for both successful and error requests). Here a quick list of non common codes that you may want to use in your application.
- **Success codes**
  - 201 Created should be used when creating content (INSERT),
  - 202 Accepted should be used when a request is queued for background processing (async tasks),
  - 204 No Content should be used when the request was properly executed but no content was returned (a good example would be when you delete something).
- **Client error codes**
  - 400 Bad Request should be used when there was an error while processing the request payload (malformed JSON, for instance).
  - 401 Unauthorized should be used when a request is not authenticated (wrong access token, or username or password).
  - 403 Forbidden should be used when the request is successfully authenticated (see 401), but the action was forbidden.
  - 406 Not Acceptable should be used when the requested format is not available (for instance, when requesting an XML resource from a JSON only server).
  - 410 Gone Should be returned when the requested resource is permenant and will never be available again.
  - 422 Unprocesable entity Could be used when there was a validation error creating an object.
- A more complete list of status codes can be found in [RFC2616](#)



## Toujours une erreur tu préciseras

- When an exception is raised, you should always return a consistent payload describing the error. This way, it will be easier for other to parse the error message (the structure will always be the same, whatever the error is).
- Here one I often use in my web applications. It is clear, simple and self descriptive :  
HTTP/1.1 401 Unauthorized  
{ "status": "Unauthorized",  
"message": "No access token provided.",  
"request\_id": "594600f4-7eec-47ca-8012-02e7b89859ce"  
}

- Gestion de versions de l'API utilisée
  - **Dans la base URL : /v2/api**
  - **Dans le type MIME :**
    - application/vnd.github.v3+json
    - application/vnd.acme.user+xml;v=1
  - Autre solution : RFC 6381
    - Accept: application/json;  
profiles="http://profiles.acme.com/user/v/1"



# Design de Web Services REST

By example

## Deux approches “complémentaires”

- Contract first
- Code first
  
- Contract First :
  - Spécifier complètement l'API dans un langage de description adapté
  - WADL (rest), WSDL (Soap) ou Open API (swagger)
- Code First :
  - j'allume le PC et hop...



## Deux approches “complémentaires”

- En pratique :
  - Spécification informelle de l'API
  - Ecriture de l'API ou de l'implémentation
  - Génération de la spec de l'API ou du code de l'API
  
- Outillage :
  - Swagger <https://editor.swagger.io/> (<->)
  - Spring REST docs (code->docs)
  - ...



# Développer des Web Services REST avec JAVA

- 2 parties nécessaires pour déployer des WS-REST :
  - La gestion des requêtes-réponses
  - Le mapping objets Java – données sérialisées ( xml, json, ...)
- 2 implémentations de librairies Java
  - JAX-RS : comment déployer des services REST en Java (mapping requête <-> méthode) ou Spring MVC
  - JAXB : comment traduire des objets java en xml/json/... de manière « automatique », Jackson

Mais alors c'est « simple » ?

- Le + utilisé : Spring MVC
- Annotations légèrement différentes par rapport à la “norme” JAX-RS  
@RequestMapping
- Utilisation de Spring Boot

- @RestController
- @RequestMapping : classe et/ou méthode
- Paramètres (injection dans paramètres):
  - @RequestParam
  - @PathVariable
  - @...
- Démo ; Cf <https://projects.spring.io/spring-framework/>  
dans la bonne version

# Spring MVC

- Annotations dans le code
  - Sur les classes
  - Sur les méthodes

Annotation	HTTP method	Typical use <sup>a</sup>
@GetMapping	HTTP GET requests	Reading resource data
@PostMapping	HTTP POST requests	Creating a resource
@PutMapping	HTTP PUT requests	Updating a resource
@PatchMapping	HTTP PATCH requests	Updating a resource
@DeleteMapping	HTTP DELETE requests	Deleting a resource
@RequestMapping	General purpose request handling; HTTP method specified in the <code>method</code> attribute	

# Example GET

```
@RestController
@RequestMapping(path="/design",produces="application/json")
@CrossOrigin(origins="*")
public class DesignTacoController {
    private TacoRepository tacoRepo;
    @Autowired
    EntityLinks entityLinks;

    public DesignTacoController(TacoRepository tacoRepo) {
        this.tacoRepo = tacoRepo;
    }
    @GetMapping("/recent")
    public Iterable<Taco> recentTacos() {
        PageRequest page = PageRequest.of(
            0, 12, Sort.by("createdAt").descending());
        return tacoRepo.findAll(page).getContent();
    }
}
```

# Exemple GET paramétré

```
@RestController
@RequestMapping(path="/design",produces="application/json")
@CrossOrigin(origins="*")
public class DesignTacoController {
    ...

    @GetMapping("/{id}")
    public Taco tacoById(@PathVariable("id") Long id) {
        Optional<Taco> optTaco = tacoRepo.findById(id);
        if (optTaco.isPresent()) {
            return optTaco.get();
        }
        return null;
    }
    ...
}
```



# Exemple GET paramétré+return

```
@RestController
```

```
@RequestMapping(path="/design",produces="application/json")
```

```
@CrossOrigin(origins="*")
```

```
public class DesignTacoController {
```

```
...
```

```
    @GetMapping("/{id}")
```

```
    public ResponseEntity<Taco> tacoById(@PathVariable("id") Long id) {
```

```
        Optional<Taco> optTaco = tacoRepo.findById(id);
```

```
        if (optTaco.isPresent()) {
```

```
            return new ResponseEntity<>(optTaco.get(), HttpStatus.OK);
```

```
        }
```

```
        return new ResponseEntity<>(null, HttpStatus.NOT_FOUND);
```

```
    }
```

```
...
```

```
}
```

# Création de données

```
@PostMapping(consumes="application/json")
@ResponseStatus(HttpStatus.CREATED)
public Taco postTaco(@RequestBody Taco taco) {
    return tacoRepo.save(taco);
}
```

# MAJ de données

```
@PostMapping("/{orderId}")  
public Order putOrder(@RequestBody Order order)  
{  
    return repo.save(order);  
}
```

# MAJ de données 2

```
@PatchMapping(path="/{orderId}", consumes="application/json")
public Order patchOrder(@PathVariable("orderId") Long orderId,
    @RequestBody Order patch) {
    Order order = repo.findById(orderId).get();
    if (patch.getDeliveryName() != null) {
        order.setDeliveryName(patch.getDeliveryName());
    }
    if (patch.getDeliveryStreet() != null) {
        order.setDeliveryStreet(patch.getDeliveryStreet());
    }
    ...
    if (patch.getCcExpiration() != null) {
        order.setCcExpiration(patch.getCcExpiration());
    }
    return repo.save(order);
}
```

# DEL de données

```
@DeleteMapping("/{orderId}")
@ResponseStatus(code=HttpStatus.NO_CONTENT)
public void deleteOrder(@PathVariable("orderId") Long orderId)
{
    try {
        repo.deleteById(orderId);
    } catch (EmptyResultDataAccessException e) {}
}
```

- Cf TP
- Gestion des erreurs : mapping auto exception ↔ HttpStatus

```
ResponseStatus(HttpStatus.NOT_FOUND)  
class UserNotFoundException extends RuntimeException {  
  
    public UserNotFoundException(String userId) {  
        super("could not find user '" + userId + "'.");  
    }  
}
```



- Acronyme de Java API for RestFul Web Services
- Version courante 2.0 décrite par JSR 339
- Depuis la version 1.1, il fait partie intégrante de la spécification Java EE 6+
- Décrit la mise en œuvre des services REST web coté client
- Son architecture se repose sur l'utilisation des classes et des annotations pour développer les services web



- JAX-RS est une spécification et autour de cette spécification sont développés plusieurs implémentations
  - JERSEY : implémentation de référence fournie par Oracle ( <http://jersey.java.net> )
  - CXF : Fournie par Apache ( <http://cfx.apache.org> )
  - RESTEasy : fournie par JBOSS
  - RESTLET : L'un des premiers framework implémentant REST pour Java

- Version actuelle 2.15 implémentant les spécifications de JAX-RS 2.0
- Intégré dans Glassfish et l'implémentation Java EE (6,7)
- Supportés dans Netbeans/Glassfish

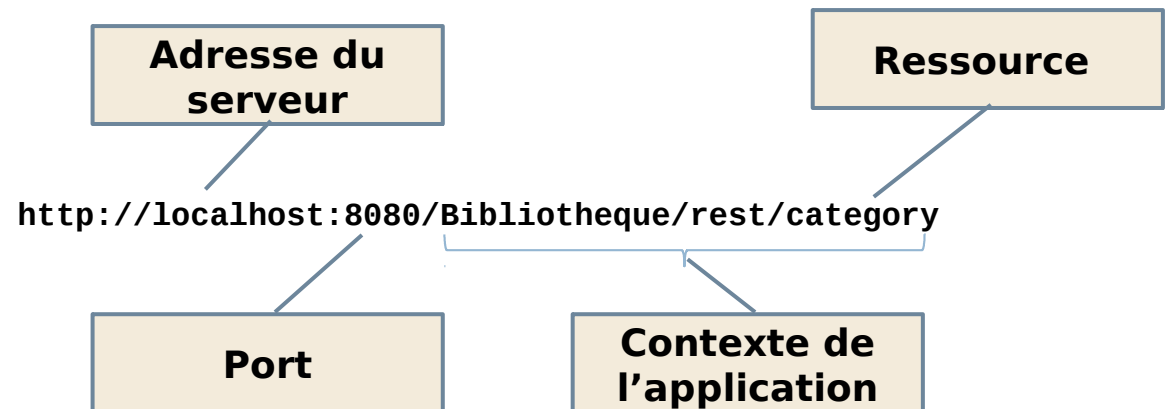
- Basé sur POJO (Plain Old Java Object) en utilisant des annotations spécifiques JAX-RS
- Pas de modifications dans les fichiers de configuration
- Le service est déployé dans une application web (ou EJB)
- Pas de possibilité de développer le service à partir d'un WADL contrairement à SOAP
- Approche Bottom/Up
  - Développer et annoter les classes
  - Le WADL est automatiquement généré par l'API

- JAX-RS utilise des annotations Java pour exposer les services
- Masque tous les traitements HTTP
- @Path : Chemin d'accès à la ressource
- @GET , @PUT, @POST , @DELETE : Type de requête
- @Consumes : Type de donnée en entrée
- @Produces : Type de donnée en sortie
- @PathParam, @QueryParam, @HeaderParam, @CookieParam, @MatrixParam, @FormParam
- + JAXB permet aussi de traiter des Objets (en paramètres d'entrée ou en retour) en Objet - XML ou JSON

## JAX-RS : @ApplicationPath contexte

- ❑ Configuration du contexte global de l'application
- ❑ @ApplicationPath("**rest**")
- ❑ Concaténé au contexte de deploy
- ❑ Racine des ressources @Path

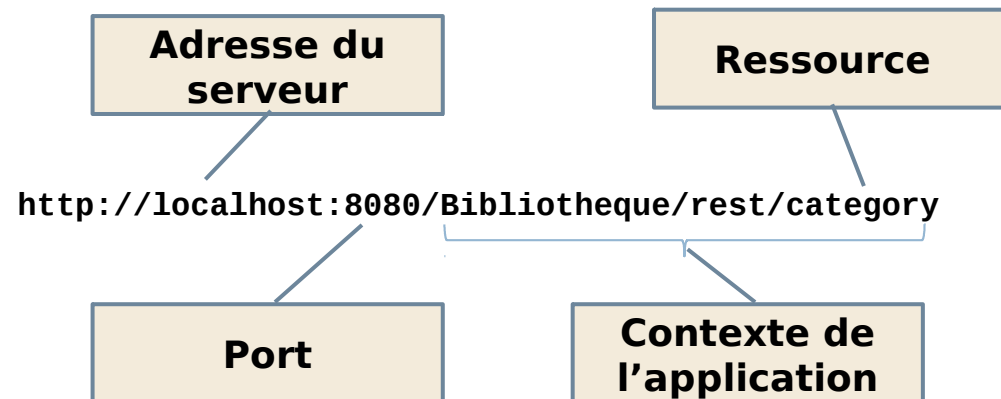
```
@Path("category")
public class CategoryFacade {
.....
}
```



## JAX-RS : @Path sur une classe

- ❑ L'annotation permet de rendre une classe accessible par une requête HTTP
- ❑ Elle définit la racine des ressources (Root Racine Ressources)
- ❑ La valeur donnée correspond à l'uri relative de la ressource

```
@Path("category")
public class CategoryFacade {
.....
}
```



## JAX-RS : @Path sur une méthode

- L'annotation peut être utilisée pour annoter des méthodes d'une classe
- L'URI résultante est la concaténation entre le valeur de @pat de la classe et celle de la méthode

```
@Path("category")
public class CategoryFacade {
    @GET
    @Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
    @Path("hello")
    public String hello() {
        return "Hello World!";
    }
}
..
}
```

<http://localhost:8080/Bibliotheque/webresources/category/hello>

- ❑ La valeur définie dans l'annotation @Path n'est forcément une constante, elle peut être variable.
- ❑ Possibilité de définir des expressions plus complexes, appelées Template Parameters
- ❑ Les contenus complexes sont délimités par « {} »
- ❑ Possibilité de mixer dans la valeur @Path des expressions

```
@GET
@Consumes ({MediaType.APPLICATION_JSON,
MediaType.APPLICATION_XML}) @Produces
({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
@Path("/{nom}")
public String hello (@PathParam("nom") String nom){
return "Hello " + nom;
}
```



- ❑ On peut limiter le match des paths dynamique avec des expressions régulières
- ❑ Exemple :  
`@Path("users/{username: [a-zA-Z][a-zA-Z_0-9]*}")`

# @GET, @POST, @PUT, @DELETE

- Permettent de mapper une méthode à un type de requête HTTP
- Ne sont utilisables que sur des méthodes
- Le nom de la méthode n'a pas d'importance, JAX détermine la méthode à exécuter en fonction de la requête

```
@Path("category")
public class CategoryFacade {
    @GET
    @Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
    @Path("test")
    public String hello()
    {
        return "Hello World!";
    }
    ..
}
```

<http://localhost:8080/Bibliotheque/webresources/category/test>

```
@GET
@Consumes ({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML}) @Produces
({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML}) @Path("{nom}")
public String hello (@PathParam("nom") String nom){
    return "Hello " + nom;
}
```

<http://localhost:8080/Bibliotheque/webresources/category/Master2>

- Les opérations CRUD sur les ressources sont réalisées au travers des méthodes de la requête HTTP



GET, POST  
PUT, DELETE



**/books**

**GET : Liste des livres**

**POST : Créer un nouveau livre**

**/books/{id}**

**GET : Livre identifié par l'id**

**PUT: Mis à jour du livre  
identifié par id**

**DELETE : Supprimer le livre  
identifié par id**

## @GET, @POST, @PUT, @DELETE

```
@Path("livre")
public class LivreFacadeREST extends AbstractFacade<Livre> {

    @POST
    @Consumes({"application/xml", "application/json"})
    public void create(Livre entity) {
        super.create(entity);
    }

    @GET
    @Produces({"application/xml", "application/json"})
    public List<Livre> findAll() {
        return super.findAll();
    }

    ...
}
```

# @GET, @POST, @PUT, @DELETE

```
@PUT
@Path("/{id}")
@Consumes({"application/xml", "application/json"})
public void edit(@PathParam("id") Long id , Livre entity) {
    super.edit(entity);
}
```

```
@DELETE
@Path("/{id}")
public void remove(@PathParam("id") Long id) {
    super.remove(super.find(id));
}
```

```
@GET
@Path("/{id}")
@Produces({"application/xml", "application/json"})
public Livre find(@PathParam("id") Long id) {
    return super.find(id);
}
```

```
@GET @Path("{from}/{to}") @Produces({"application/xml", "application/json"})
public List<Livre> findRange(@PathParam("from") Integer from, @PathParam("to")
Integer to) {
    return super.findRange(new int[]{from, to});
}
}
```

# Paramètres des requêtes

- JAX-RS fournit des mécanismes pour extraire des paramètres dans la requête
- Utilisés sur les paramètres des méthodes des ressources pour réaliser des injections de contenu
- Différentes annotations :
  - @PathParam : valeurs dans templates parameters
  - @QueryParam : valeurs des paramètres de la requête
  - @FormParam : Valeurs des paramètres de formulaire
  - @HeaderParam: Valeurs dans l'en tête de la requête
  - @CookieParam : Valeurs des cookies
  - @Context : Informations liés au contexte de la ressource

# Paramètres des requêtes

- Exemple avec Default :

```
@Path("smooth")
```

```
@GET
```

```
public Response smooth(  
    @DefaultValue("2") @QueryParam("step") int step,  
    @DefaultValue("true") @QueryParam("min-m") boolean hasMin,  
    @DefaultValue("true") @QueryParam("max-m") boolean hasMax,  
    @DefaultValue("true") @QueryParam("last-m") boolean hasLast,  
    @DefaultValue("blue") @QueryParam("min-color") ColorParam  
    minColor,  
    @DefaultValue("green") @QueryParam("max-color") ColorParam  
    maxColor,  
    @DefaultValue("red") @QueryParam("last-color") ColorParam  
    lastColor) {  
    ...  
}
```

- Remarque : si parsing échoue, 404

## Paramètres des requêtes

- Pour les objets en param, il faut :
  - Be a primitive type;
  - Have a constructor that accepts a single String argument;
  - Have a static method named `valueOf` or `fromString` that accepts a single String argument (see, for example, `Integer.valueOf(String)` and `java.util.UUID.fromString(String)`);
  - Have a registered implementation of `javax.ws.rs.ext.ParamConverterProvider` JAX-RS extension SPI that returns a `javax.ws.rs.ext.ParamConverter` instance capable of a "from string" conversion for the type. or
  - Be `List<T>`, `Set<T>` or `SortedSet<T>`, where T satisfies 2 or 3 above. The resulting collection is read-only.



## Et la response ?

- Type primitif
- String
- Objets java [!]
- Autre : Response
- Permet de construire une réponse HTTP complète [status, entete, ...]
- Eg POST avec Location, cf exemple Sondage

- Cf Doc Jersey
- <https://jersey.java.net/documentation/latest/jaxrs-resources.html>



# Et côté client ?

Comment on appelle ces trucs ?

## □ Client http

- Telnet (hum !)
- Curl
- Httpie
- Soapui
- Karaté
- ...

## □ Codage client [Java, .NET, ...]

- Jersey
- Spring MVC
- ...

- Création du client (opération lourde) :

```
Client client = Client.create();
```

- Accès à une ressource :

```
WebResource webResource =  
    client.resource("http://example.com/base");
```

- Opérations sur les ressources :

```
String s =  
webResource.accept("text/plain").get(String.class);
```

```
ClientResponse response =  
webResource.type("text/plain").put(ClientResponse.class  
, "foo:bar");
```

### □ Fluent API

```
client = ClientBuilder.newClient();
```

```
Response response = client  
    .target(BASE_REST_URI)  
    .path(projectKey)  
    .path("repos")  
    .queryParams("limit", 200)  
    .request(MediaType.APPLICATION_JSON)  
    .get();  
if (response.getStatus() == 200) {  
    return response.readEntity(Repos.class);  
}
```

# Exemple de Client Jersey 3

## □ Config client

```
ClientConfig clientConfig = new ClientConfig();
HttpAuthenticationFeature feature = HttpAuthenticationFeature.basic("login", "motpasse");
clientConfig.register(feature);
ObjectMapper om = new ObjectMapper()
    .registerModule(new JavaTimeModule())
    .configure(SerializationFeature.WRITE_DATE_TIMESTAMPS_AS_NANOSECONDS, false)
    .configure(SerializationFeature.FAIL_ON_EMPTY_BEANS, false)
    .configure(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES, false)
    .configure(DeserializationFeature.READ_DATE_TIMESTAMPS_AS_NANOSECONDS, false)
    .setSerializationInclusion(JsonInclude.Include.NON_NULL);
clientConfig.register(om);

client = ClientBuilder.newClient(clientConfig);
```

- Création du client (opération lourde) :  
`RestTemplate restTemplate = new RestTemplate();`
- Accès à une ressource :

- Opérations sur les ressources :

```
Quote quote =  
    restTemplate.getForObject("http://serveur.io/api/random",  
        Quote.class);
```



## Exemple de client Spring (2)

### □ Avec header, body :

```
// headers
HttpHeaders httpHeaders = new HttpHeaders();
//httpHeaders.set("Accept", "*/*");

// body
MultiValueMap<String, String> map
    = new LinkedMultiValueMap<String, String>();
map.add("attribut", valeur);

HttpEntity<MultiValueMap<String,String>> httpEntity
    = new HttpEntity<MultiValueMap<String,String>>(map , httpHeaders);

String url = "http://serveur:port/urllapi";

String res = restTemplate.postForObject(url,httpEntity,String.class);
```

- Le client REST [text/html] le plus commun est le navigateur Web
- Pour faire communiquer des applications REST ou effectuer des tests unitaires, des implémentations clientes existent.

le plus connu/utilisé : SoapUI

- Il existe de nombreux outils [en ligne] permettant de tester les services Web REST
- Certains sont disponibles sous forme d'extensions que vous pouvez installer dans les navigateurs
  - RestConsole
  - PostMan
- D'autres directement (ou plugins) dans l'EDI (IntelliJ, Netbeans, ...)
- Ou sous la forme d'outils indépendants : SoapUI

# Et les interfaces ?

Ben quand tu définis une API, tu donnes la spec eh cong !

3 possibilités :

- Descripteur WADL
- Descripteur Swagger / OpenAPI Spec. (OAS)
- HATEOAS

- Web Application Definition Language est un langage de description des services REST au format XML. Il est une spécification du W3C initiée par SUN ([www.w.org/Submission/wadl](http://www.w.org/Submission/wadl))
- Il décrit les éléments à partir de leur type (Ressources, Verbes, Paramètre, type de requête, Réponse)
- Il fournit les informations descriptives d'un service permettant de construire des applications clientes exploitant les services REST

- **Web Application Description Language**
- Équivalent du WSDL pour services SOAP
- Interface décrivant les opérations disponibles d'un service REST, ses entrées/sorties, les types, les MIME type acceptés/renvoyés/...
- Créé automatiquement (en statique ou dynamique) par ***certaines*** des librairies à partir de la définition des services (comme WSDL)

# WADL : exemple

```
<application xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://research.sun.com/wadl/2006/10 wadl.xsd"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:ex="http://www.example.org/types"
  xmlns="http://research.sun.com/wadl/2006/10">

  <resources base="http://www.example.org/services/">
    <resource path="getStockQuote">
      <method name="GET">
        <request>
          <param name="symbol" style="query" type="xsd:string"/>
        </request>
        <response>
          <representation mediaType="application/xml"
            element="ex:quoteResponse"/>
          <fault status="400" mediaType="application/xml"
            element="ex:error"/>
        </response>
      </method>
    </resource>
  </resources>
</application>
```



- Génération auto (code first) uniquement pour certaines implémentations
- Utilisation avec SoapUI

- editor.swagger.io
- Fonctionne dans les 2 sens :
  - API first → génération d'un squelette de serveur ET de client, multi-plateforme
  - Code First → Génération de la spec en OpenAPI / Swagger
- Utilisable avec SoapUI
- Documentation en HTML avec tests d'appel

# Exemple code → docs/spec

## □ Avec Spring boot : springfox

```
<dependency>
  <groupId>io.springfox</groupId>
  <artifactId>springfox-swagger2</artifactId>
  <version>2.8.0</version>
</dependency>
<dependency>
  <groupId>io.springfox</groupId>
  <artifactId>springfox-swagger-ui</artifactId>
  <version>2.8.0</version>
</dependency>
```

## □ Dans le code : @EnableSwagger2 +

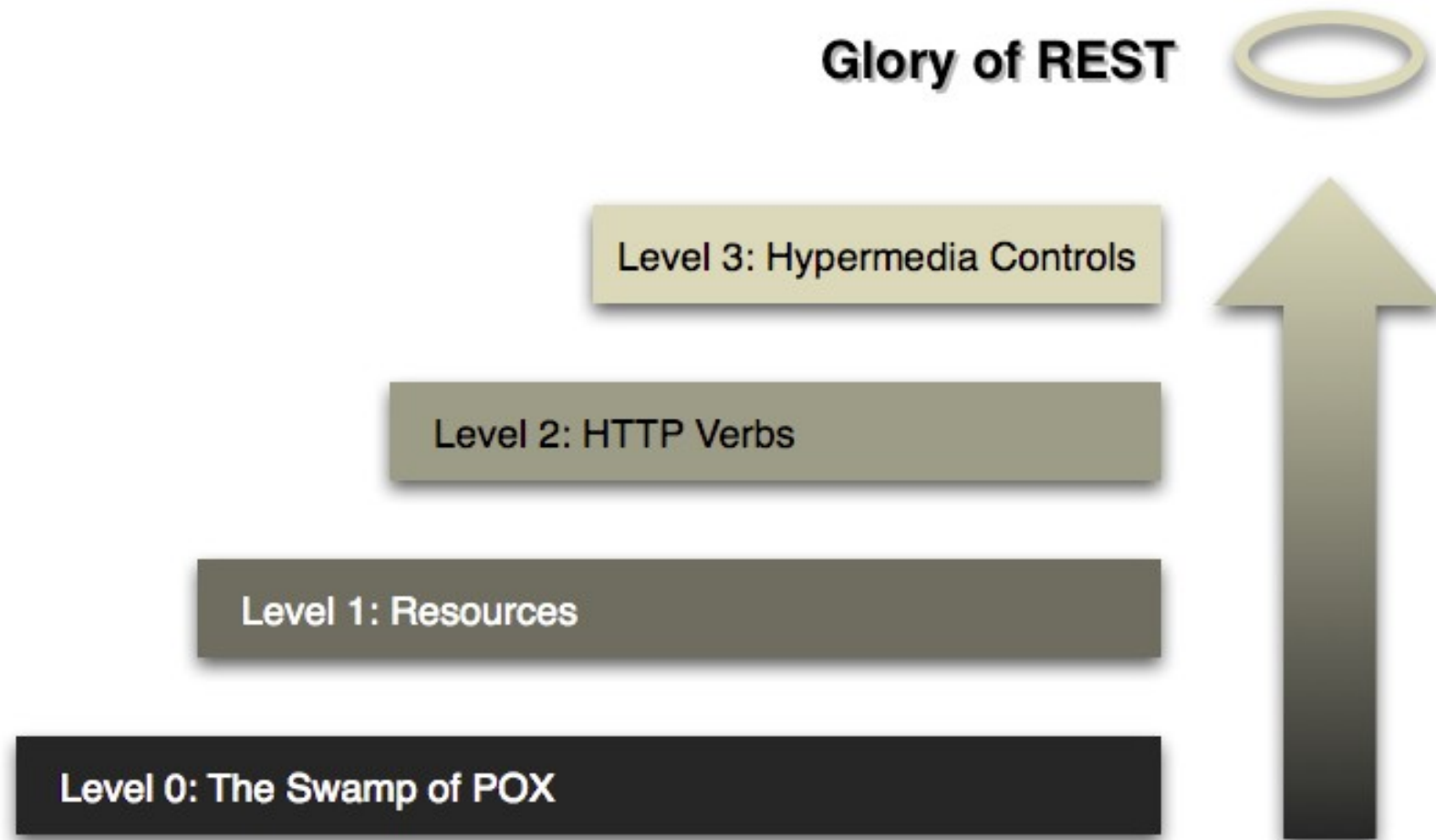
```
@Bean
public Docket api() {
    return new Docket(DocumentationType.SWAGGER_2)
        .select()
        .apis(RequestHandlerSelectors.any())
        .paths(PathSelectors.any())
        .build();
}
```

Démo : boot-swagger,

<http://localhost:8080/v2/api-docs>

<http://localhost:8080/swagger-ui.html>

## □ Level UP



- Niveau 2 : OK, mais le client doit connaître l'API pour l'utiliser ; toute modification plante le client
- Idée de “Hypermedia as the engine of application state” : casser ce couplage
- Le client “navigue” dans les URLs fournies dynamiquement par le serveur
- Cf Spring HATEOAS

PLUS QU'1 [GROS] PB :  
ET LES OBJETS JAVA ?

JAXB -  
XML(JSON) ET JAVA

- JAXB (Java API for XML Binding) est un *framework* qui permet d'associer un modèle objet écrit en Java à un modèle objet écrit en XML
- Il fait le mapping (dans les deux sens) entre classes Java et des schémas XSD
  - Génération de classes à partir d'un schéma XSD
  - Génération d'un schéma à partir de classes
- Il permet ensuite de passer automatiquement d'instances (objets) à des documents XML conformes à un schéma.

# Premier exemple

Annotations dans le code Java pour  
générer un schéma et des documents  
XML

API JAXB intégrée dans Java SE 6+  
[pas import]



## Exemple

```
@XmlElement(name="marin",
    namespace="http://orleans.miage.fr/cours-wsi")
@XmlAccessorType(XmlAccessType.FIELD)
@Entity
public class Marin implements Serializable {
    @XmlAttribute(name="id")
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String nom ;
    @Enumerated(EnumType.STRING)
    private Grade grade ;
    ...
}
```

- `@XmlRootElement` : indique que les objets seront des objets racine des documents XML que nous allons générer. Le nom passé en attribut correspond au nom de l'élément racine du document XML généré. Le namespace est lui l'espace de noms utilisé
- `@XmlAccessorType` : indique que les champs de la classe `Marin` seront lus directement, sans passer par les *getters* . [ ou `XmlAccessType.PROPERTY` ]
- `@XmlAttribute` : indique que ce champ doit être écrit dans un attribut plutôt qu'un sous-élément. Le nom de cet attribut est précisé en tant qu'attribut de cette annotation.

# Création document XML

```
public static void main(String[] args) throws JAXBException {  
    // création d'un objet de type Marin  
    Marin marin = new Marin() ;  
    marin.setId(15L) ; marin.setNom("Surcouf") ;  
    marin.setGrade(Grade.PACHA) ;  
    // création d'un contexte JAXB sur la classe Marin  
    JAXBContext context = JAXBContext.newInstance(Marin.class) ;  
    // création d'un marshaller à partir de ce contexte  
    Marshaller marshaller = context.createMarshaller() ;  
  
    marshaller.setProperty("jaxb.encoding", "UTF-8") ; // on choisit UTF-8  
    marshaller.setProperty("jaxb.formatted.output", true) ; // formater ce  
    fichier  
  
    // écriture finale du document XML dans un fichier surcouf.xml  
    marshaller.marshal(marin, new File("surcouf.xml")) ;  
}
```

```
<?xml version="1.0" encoding="UTF-8"
  standalone="yes"?>
<ns2:marin
  xmlns:ns2="http://orleans.miage.fr/cours-
  wsi" id="15">
  <nom>Surcouf</nom>
  <grade>PACHA</grade>
</ns2:marin>
```

```
public class FromXML {  
    public static void main(String[] args) throws  
    JAXBException {  
        // création d'un contexte JAXB sur la classe Marin  
        JAXBContext context =  
        JAXBContext.newInstance(Marin.class) ;  
        // création d'un unmarshaller  
        Unmarshaller unmarshaller =  
        context.createUnmarshaller() ;  
        Marin surcouf = (Marin)unmarshaller.unmarshal(new  
        File("surcouf.xml")) ;  
        System.out.println("Id = " + surcouf.getId()) ;  
        System.out.println("Nom = " + surcouf.getNom()) ;  
    }
```

## Exemple

- Annotation des classes du « modèle »
- Lecture/écriture de documents XML très simple en JAXB pour des cas simples de classes/mapping
- Intégration dans JEE5 / JEE6 / JEE7...
- Possibilité de traiter des cas plus complexes

## 1er problème

### □ Changer le type généré en JSON ?

```
marshaller.setProperty(MarshallerProperties.MEDIA_  
TYPE, "application/json");  
marshaller.setProperty(MarshallerProperties.JSON_I  
NCLUDE_ROOT, false);
```

### □ Résultat :

```
Exception in thread "main" Exception in thread  
"main" javax.xml.bind.PropertyException: name:  
eclipseLink.media-type value: application/json
```

## Comment faire du JSON ?

- Par défaut, pas de Provider JSON
- Utiliser une librairie une librairie qui le supporte : Jettison, JaxMe, MOXy...
- MOXy inclus dans EclipseLink (JPA !)
  - Changer le Provider par MOXy :
  - Fichier `jaxb.properties` dans package, définir :  
`javax.xml.bind.context.factory=org.eclipse.persistence.jaxb.JAXBContextFactory`
- Pour Jersey, dépendances à ajouter dans le pom

Pas de normalisation => cf Docs



## □ Run :

```
marshaller.setProperty(MarshallerProperties.MEDIA_TYPE,  
    "application/json");  
marshaller.setProperty(MarshallerProperties.JSON_INCLUDE_ROOT,  
    false);  
marshaller.marshal(marin, System.out);
```

## □ Résultat :

```
{  
    "id" : 15,  
    "nom" : "Surcouf",  
    "grade" : "PACHA"  
}
```



# Annotations JAXB

La sérialisation : Je maîtrise !

## But

- un schéma XML permet de contraindre un document XML de façon plus forte qu'une classe ne contraint une instance
- eg la notion d'ordre des champs dans une classe n'existe pas en Java. Au contraire, la notion d'ordre des sous-éléments d'un élément racine ou non existe en XML

=> Compléter les informations de Classe

- De 3 types :
  - Placées sur un package
  - Placées sur les classes ou énumérations
  - Placées dans les classes sur les champs ou getters

- Fixe les schémas XML utilisés pour écrire les documents XML générés
- Annotation sur package
- Fichier « package-info.java », contenant la javadoc, les annotations et juste la déclaration du package

## @XmlSchema : déclaration

### □ Exemple :

// fichier package-info.java

```
@XmlSchema (  
    xmlns = {  
        @XmlNs(prefix="orl",  
            namespaceURI="http://orleans.miage.fr/cours-jaxb"),  
        @XmlNs(prefix="xsd",  
            namespaceURI="http://www.w3.org/2001/XMLSchema") },  
    namespace="http://orleans.miage.fr/cours-jaxb",  
    elementFormDefault=XmlNsForm.QUALIFIED,  
    attributeFormDefault=XmlNsForm.UNQUALIFIED  
)  
package modele;
```

- Déclaration générée :

**<xs:schema**

elementFormDefault="qualified"

attributeFormDefault="unqualified"

targetNamespace="http://orleans.miage.fr/  
cours-jaxb"

xmlns:orl="http://orleans.miage.fr/cours-jaxb"

xmlns:xsd="http://www.w3.org/2001/  
XMLSchema"

version="1.0" >

- `@XmlRootElement` associe la classe annotée avec un nœud racine d'un document XML. Dans les cas simples, c'est la seule annotation que l'on trouve sur une classe ; Suffisante
- `@XmlType` permet plusieurs choses. Le point principal est son attribut `propOrder`, qui permet de fixer l'ordre dans lequel les champs de cette classe doivent être enregistrés dans le document XML



### □ Exemple de @XmlType

```
@XmlRootElement(name="marin")
@XmlType(propOrder={"nom", "prenom", "age"})
public class Marin {
    @XmlAttribute(name="id")
    private long id ;
    private String nom, prenom ;
    private age ; ...
}
```

# Sur les classes

- @XmlAccessorType : par def, tous les champs ou propriétés d'une classe sont pris en compte lors de la génération de XML [sauf ceux qui sont annotés @XmlTransient]
- Par défaut, JAXB prend en compte toutes les propriétés publiques et les champs annotés. Donc si l'on annote un champ privé, sans annoter la classe avec @XmlAccessType.FIELD =>JAXB verra deux fois le même champ : une fois du fait de l'annotation, et la deuxième fois du fait qu'il prend en compte le *getter* public.
- Les valeurs que peut prendre cette annotation sont les suivantes :
  - @XmlAccessType.FIELD : indique que tous les champs non statiques de la classe sont pris en compte ;
  - @XmlAccessType.PROPERTY : indique que toutes les paires de *getters* / *setters* sont prises en compte ;
  - @XmlAccessType.PUBLIC : indique que toutes les paires de *getters* / *setters* et tous les champs publics non statiques seront pris en compte ;
  - @XmlAccessType.NONE : indique qu'aucun champ ou propriété n'est pris en compte

- @XmlEnum permet de préciser la façon dont les valeurs d'une énumération vont être écrites dans le code XML
- @XmlEnum fonctionne comme l'annotation JPA @EnumeratedType, qui permet d'imposer d'écrire en base le nom des valeurs énumérées plutôt que le numéro d'ordre

# Sur les classes

// énumération Grade

*@XmlType*

*@XmlEnum(String.class)*

```
public enum Grade {  
    MATELOT, BOSCO, PACHA, CUISINIER  
}
```

// Schéma XML généré

```
<xsd:simpleType name="Grade">  
    <xsd:restriction base="xsd:string"/>  
    <xsd:enumeration value="MATELOT"/>  
    <xsd:enumeration value="BOSCO"/>  
    <xsd:enumeration value="PACHA"/>  
    <xsd:enumeration value="CUISINIER"/>  
</xsd:simpleType>
```

# Sur les classes

```
// énumération Grade
@XmlType
@XmlEnum(Integer.class)
public enum Grade {
    @XmlEnumValue("10") MATELOT,
    @XmlEnumValue("20") BOSCO,
    @XmlEnumValue("30") PACHA,
    @XmlEnumValue("40") CUISINIER
}
// Schéma XML généré
<xsd:simpleType name="Grade">
  <xsd:restriction base="xsd:int"/>
  <xsd:enumeration value="10"/>
  <xsd:enumeration value="20"/>
  <xsd:enumeration value="30"/>
  <xsd:enumeration value="40"/>
</xsd:simpleType>
```

- **@XmlElement** permet d'associer un champ ou un *getter* à un nœud d'un document XML. Il permet de spécifier le nom de cet élément, son espace de nom, sa valeur par défaut, etc...
- **@XmlTransient** : sur un champ ou un *getter* le retire des éléments pris en compte pour la création des schémas et des documents XML

- **@XmlElementWrapper et @XmlElements**
- Si le champ ou le *getter* que l'on annote est de type List, alors on peut spécifier les choses plus finement
- Une List Java permet d'enregistrer des éléments de différents types [générique ou non]
- En utilisant l'annotation @XmlElements, on peut spécifier que chaque élément de la liste, du fait de sa classe, est associée à un nœud différent

# Sur un champ ou getter

□ Exemple :

```
@XmlRootElement(name="bateau")
@XmlAccessorType(XmlAccessType.FIELD)
public class Bateau {
    @XmlAttribute
    private long id;
    @XmlElement(name = "nom")
    private String nom;
    @XmlElementWrapper(name = "equipage")
    @XmlElements({
        @XmlElement(name = "marin", type = Marin.class),
        @XmlElement(name = "capitaine", type = Capitaine.class),
        @XmlElement(name = "cuisinier", type = Cuisinier.class)
    })
    private List<Marin> equipage = new ArrayList<Marin>();
```



## Sur un champ ou getter

### □ Résultat :

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<bateau xmlns:ns2="http://orleans.miage.fr/cours-jaxb"
  id="3">
  <nom>Altaïr</nom>
  <equipage>
    <marin id="10"> <nom>Surcouf</nom></marin>
    <cuisinier id="20"> <nom>Cook</nom></cuisinier>
    <capitaine id="30">
      <nom>Magellan</nom></capitaine>
    </equipage>
  </bateau>
```

- @XmlList se pose sur des champs ou *getters* de type List, et n'est valide que lorsque ces listes portent des types primitifs Java, des classes enveloppe, ou des chaînes de caractères ( String).
- Dans ce cas, la liste est écrite dans un unique élément XML, et ses éléments sont séparés par des espaces

# Sur un champ ou getter

□ Exemple :

@XmlRootElement

**public class** Tableau {

@XmlElement

@XmlList

**private** List<String> **nombres** = **new** ArrayList<String>();

**public** List<String> getNombres() {

**return this.nombres;**

}

}

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>

**<tableau>**

**<nombres>**un deux trois**</nombres>**

**</tableau>**

- **@XmlAttribute** : Cet élément permet d'écrire le champ annoté dans un attribut XML plutôt que dans un sous-élément de l'élément XML parent. On peut fixer le nom de cet attribut, l'espace de noms auquel il appartient et imposer qu'il soit présent (attribut required)
- **@XmlValue** : Une classe donnée ne peut avoir plus d'un champ ou *getter* annoté par @XmlValue. Indique que les instances de cette classe sont représentées par une valeur simple unique, donnée par le champ qui porte cette annotation. Ainsi, lorsque ces objets seront utilisés ailleurs, ils seront représentés par cette valeur simple

## Sur un champ ou getter

### □ Exemple @XmlValue

```
@XmlAccessorType(XmlAccessType.FIELD)
public class Salaire {
    @XmlValue
    private int montant ; ...
}
```

```
@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Marin {
    private Salaire salaire ; ...
}
```

□ Résultat pour un marin :

```
<?xml version="1.0" encoding="UTF-8"  
  standalone="yes"?>
```

```
<marin>
```

```
  <salaire>100</salaire>
```

```
</marin>
```



Générer un schéma

- Une classe annotée peut être utilisée pour générer un schéma XML. Les documents XML générés seront valides par rapport à ce schéma
- Pour cette génération, il faut utiliser un utilitaire Java qui fait partie de la distribution de JAXB [cf site JAXB] : schemagen
- schemagen peut s'utiliser au travers d'une tâche Ant ou d'un plugin Maven



## Exemple

```
@XmlRootElement(name="marin",  
    namespace="http://orleans.miage.fr/cours-jaxb")  
@XmlAccessorType(XmlAccessType.FIELD)  
@XmlType(propOrder={"nom", "prenom", "grade",  
    "age"})  
public class Marin {  
    @XmlAttribute(name="id") private long id ;  
    @XmlElement(required=true) private String nom ;  
    private String prenom ;  
    private long age ;  
    private Grade grade ; ...}
```

## Exemple

```
@XmlRootElement(name="bateau")
@XmlAccessorType(XmlAccessType.FIELD)
public class Bateau {
    @XmlAttribute private long id ;
    private String nom ;
    @XmlElementWrapper(name="equipage")
    private List<Marin> equipage = new
    ArrayList<Marin>() ;

    ...
}
```

## Deux schémas générés

### □ Marin.xsd

```
<?xml version="1.0" encoding="UTF-8"
  standalone="yes"?>
<xs:schema version="1.0"
  targetNamespace="http://orleans.miage.fr/cours
-jaxb"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  >
  <xs:import schemaLocation="schema2.xsd"/>
  <xs:element name="marin" type="marin"/>
</xs:schema>
```

# Deux schémas générés

## □ Schema2.xsd

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
  <xs:schema version="1.0" xmlns:ns1="http://orleans.miage.fr/cours-jaxb"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:import namespace="http://orleans.miage.fr/cours-jaxb"
      schemaLocation="marin.xsd"/>
    <xs:element name="bateau" type="bateau"/>
    <xs:complexType name="bateau">
      <xs:sequence>
        <xs:element name="nom" type="xs:string" minOccurs="0"/>
        <xs:element name="equipage" minOccurs="0">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="equipage" type="marin" nillable="true"
                minOccurs="0" maxOccurs="unbounded"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute name="id" type="xs:long" use="required"/>
    </xs:complexType>
```

# Deux schémas générés

## □ Schema2.xsd

```
<xs:complexType name="marin">
  <xs:sequence>
    <xs:element name="nom" type="xs:string"/>
    <xs:element name="prenom" type="xs:string" minOccurs="0"/>
    <xs:element name="grade" type="grade" minOccurs="0"/>
    <xs:element name="age" type="xs:long"/>
  </xs:sequence>
  <xs:attribute name="id" type="xs:long" use="required"/>
</xs:complexType>

<xs:simpleType name="grade">
  <xs:restriction base="xs:string">
    <xs:enumeration value="MATELOT"/> <xs:enumeration
value="BOSCO"/> <xs:enumeration value="PACHA"/>
    <xs:enumeration value="CUISINIER"/> </xs:restriction>
  </xs:simpleType>
</xs:schema>
```



# Générer des classes

- La génération d'un jeu de classes utilise un autre utilitaire fourni dans la distribution JAXB : xjc
- On peut utiliser un script Ant ou un plugin maven
- Lecture des xsd et génération des classes correspondantes
- A utiliser pour faire un client

- Génère toutes les classes du modèle ET une classe ObjectFactory
- C'est une classe fabrique, qui expose autant de méthodes que nous avons de classes générées



□ Exemple :

```
public static void main(String... args) {  
    ObjectFactory factory = new ObjectFactory() ;  
  
    Marin marin = factory.createMarin() ;  
    Bateau bateau = factory.createBateau() ;  
  
    Bateau.Equipage equipage =  
    factory.createBateauEquipage() ;  
    bateau.setEquipage(equipage) ;  
    bateau.getEquipage().getEquipage().add(marin) ;  
}
```

- Java le soir, Blog de José Paumard
- JAXB
- MOXy EclipseLink
- Mojo maven plugins pour JAXB

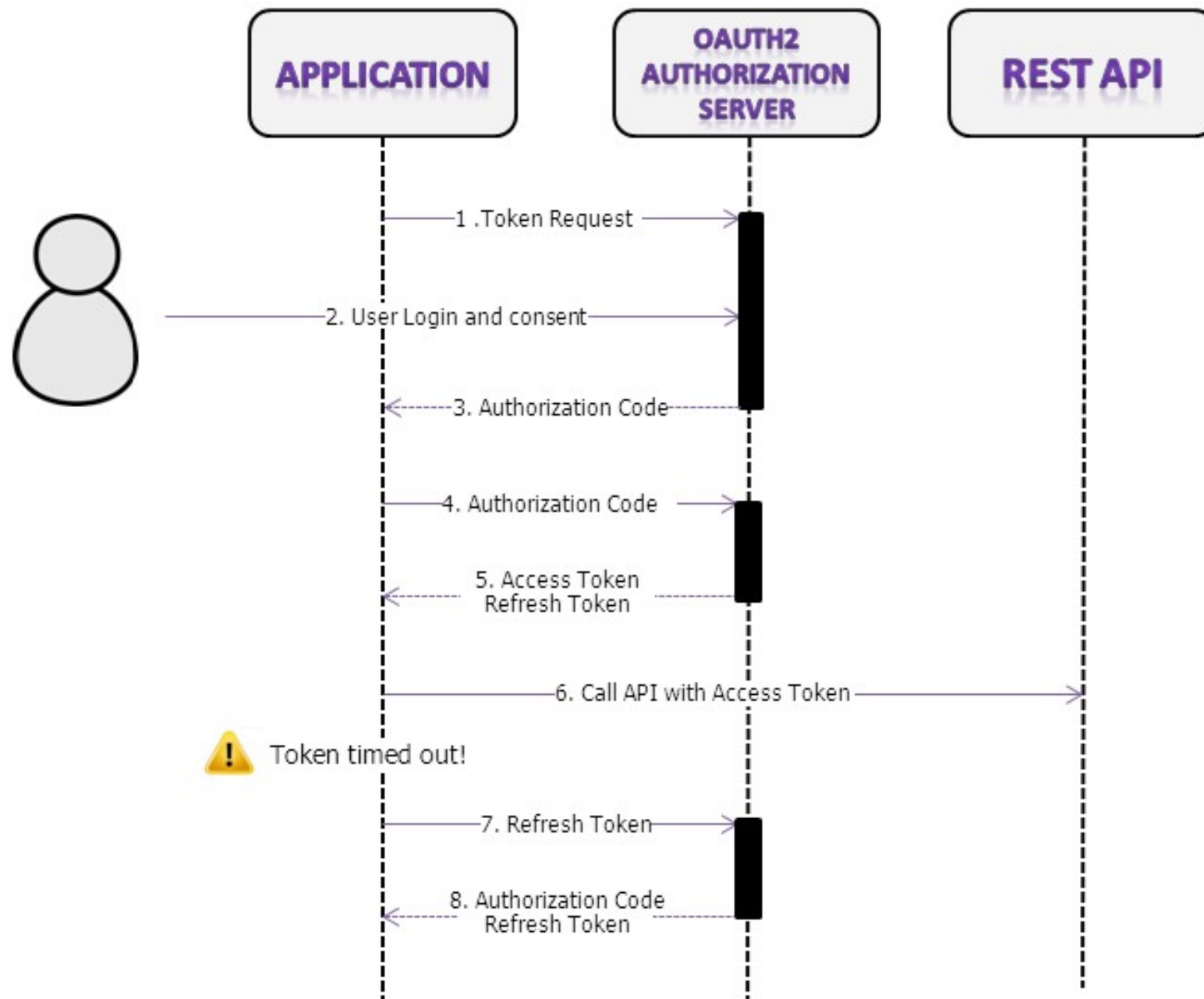


Et la sécurité ?

- Distinguer 2 problèmes :
  - Authentification : vérifier l'identité d'une "personne"
  - Autorisations associées à l'entité authentifiée
- En REST, protection de l'accès aux ressources

- Au niveau transport : https, utilisation de TLS
- Au niveau accès (user) aux contenus :
  - Authentification login/password
  - Éviter de les envoyer à chaque échange : clé d'authentification (token)
  - Protocole(s) de communication login/pass – clé entre client/serveur/serveur tiers
- Deux solutions :
  - Utilisation protocole standard, e.g. OAuth 2.0 [externe]
  - Protocole interne défini

# OAuth 2.0



## Définition d'un protocole

- 2 ou 3 intervenants
- Avantage : sécurité
- Inconvénient : non standard

- Principes récurrents : URL – droits d'accès
- Eg Spring Security :

```
protected void configure(HttpSecurity http)
    throws Exception {
    http
        .antMatcher("/**")
        .authorizeRequests()
        .antMatchers("/", "/login**", "/webjars/**")
        .permitAll()
        .anyRequest()
        .authenticated();
}
```



- En servlet : interception de “toutes” les requêtes entrantes pour vérifier la présence d’un token
- Exemple en Spring boot MVC :

```
@Configuration
public class WebConfig extends WebMvcConfigurerAdapter {
    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new SecurityInterceptor()).addPathPatterns("/**");
    }
}
```

## □ Exemple en Spring boot MVC : Handler

```
public class SecurityInterceptor implements HandlerInterceptor {
    private static final Logger logger = LoggerFactory.getLogger(SecurityInterceptor.class);

    @Override
    public boolean preHandle(HttpServletRequest httpServletRequest, HttpServletResponse
httpServletResponse, Object o) throws Exception {
        logger.info("request");
        if (httpServletRequest.getHeader("jepasse")!=null) {
            return true;
        }
        return false; // blocage de la requête AVANT traitement
    }

    @Override
    public void postHandle(HttpServletRequest httpServletRequest, HttpServletResponse
httpServletResponse, Object o, ModelAndView modelAndView) throws Exception {
        logger.info("response");
    }

    @Override
    public void afterCompletion(HttpServletRequest httpServletRequest, HttpServletResponse
httpServletResponse, Object o, Exception e) throws Exception {
        logger.info("afterComp");
    }
}
```

- 2 techniques pour le Token :
  - Génération aléatoire, puis stockage (BD, map...) d'association
  - Utilisation encryptage pour signature (e.g. JWT)
- En général, passé par Header ("Authorization" ou autre...)

- Génération de tokens signés par clé privée : JSON Web Tokens standard RFC 7519
- [jwt.io](https://jwt.io) : support multi-langages /multi-plateforme
- Stateless ! =>perfs : scalable
- Principe : fabriquer un token avec des données, le signer pour garantir la non-modification

- Exemple de token :

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG91IiwiaWF0IjYWRtaW4iOnRydWV9.TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ
```

- Contient des infos en clair  
(header+payload) décodables en clair  
(base 64) + signature décodable  
uniquement avec la clé privée
- Vérification de signature pour valider les  
claims

# Token JWT : décodage

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjE5ODc2MzQ1NjUwLjJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ
```

## HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

## PAYLOAD: DATA

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

## VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  
) ☐ secret base64 encoded
```

# Token JWT : en Java ?

## □ Librairie jjwt

```
<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt</artifactId>
  <version>0.9.0</version>
</dependency>
```

## □ Génération d'un token :

```
Claims claims = Jwts.claims().setSubject(login);
claims.put("roles", user.getRoles()); ...
String token = Jwts.builder()
    .setClaims(claims)
    .setExpiration(new Date(System.currentTimeMillis() + EXPIRATION_TIME))
    .signWith(SignatureAlgorithm.HS512, SECRET_KEY.getBytes())
    .compact();
```

## □ Décodage du token :

```
Jws<Claims> jwsClaims = Jwts.parser()
    .setSigningKey(SECRET_KEY.getBytes())
    .parseClaimsJws(tokenToCheck.replace(TOKEN_PREFIX, ""));
String login = jwsClaims.getBody().getSubject(); ...
```

- Stateless => toutes les informations stockées dans le token
- Logout ?
- Changement des données ?
- Invalidation d'un token ?





Petite conclusions ?

- HTTP est Stateless
- Ajout d'information dans les ressources retournées représentant les **transitions valides**
- Technique HTTP classiques de sessions

- Utilisation de la sécurité HTTP (S)
  - TLS
  - Authentification, en général basée sur URI [interception par Spring Security par exemple]
  - Oauth (2.0) de plus en plus utilisé sur les API REST (eg Google, Facebook...)

- HTTP est un protocole synchrone
- On peut simuler une file simplement :
- POST /queue
- 202 Accepted
- Location: /queue/message/1
- GET /queue/message/1

- Surtout utilisé pour exposé des services non authentifié sur le web [mais pas seulement]
- Utilisé pour ses « performances »
- Utilisé pour des services orientés « utilisateur final »
- Utilisé pour sa simplicité
- Utilisé avec des librairies JavaScript (jQuery, Dojo, AngularJS...) pour la partie cliente
- En natif sur les serveurs JEE mais aussi

# Services Web étendus VS REST

```
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/" xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Header/>
  <S:Body> <ns2:hello xmlns:ns2="http://services.bibliotheque.univ-orleans.fr/">
    <name>Informatique</name>
  </ns2:hello>
</S:Body>
</S:Envelope>
```



Client

SOAP



Serveur



Client

REST

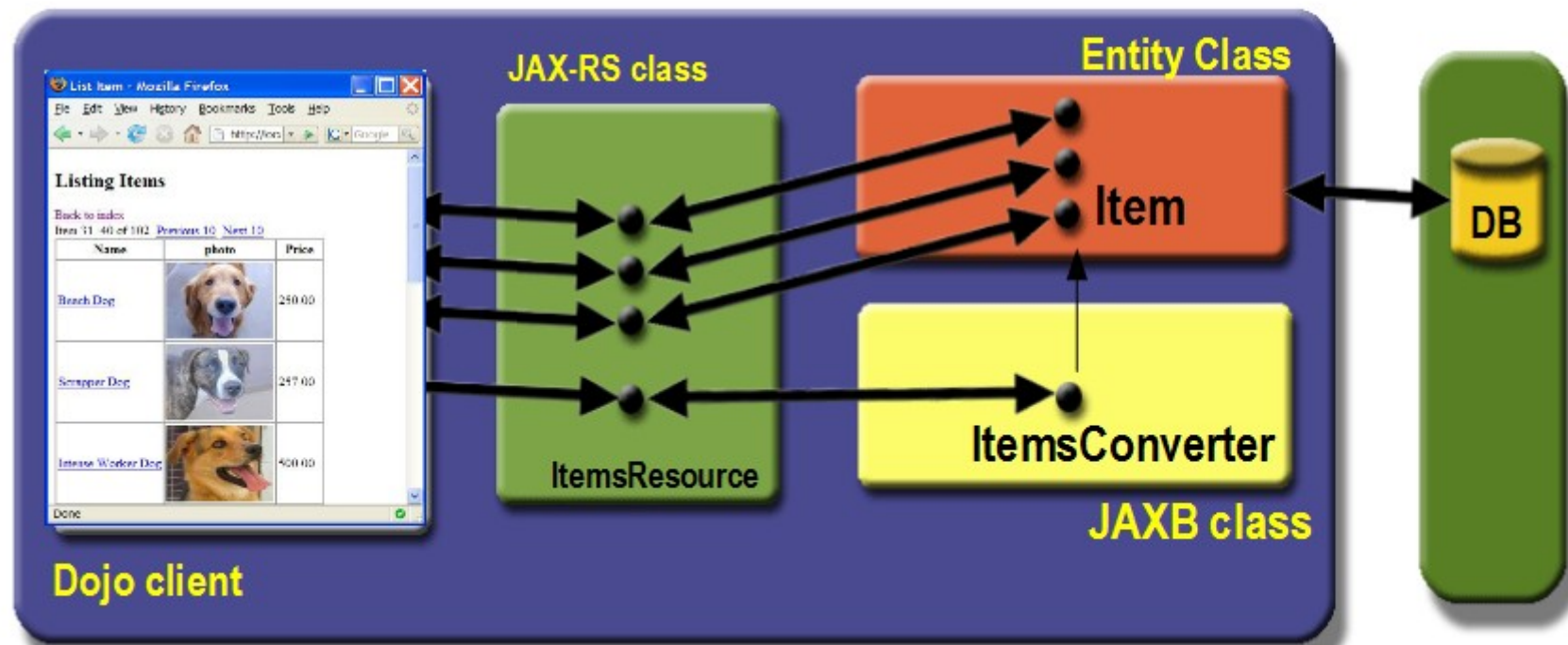


Serveur

<http://localhost:8080/Bibliotheque/webresources/category/informatique>

## RESTful Catalog

- Dojo client, JAX-RS, JAXB, JPA



## SOAP

### → Avantages

- Standardisé
- Interopérabilité
- Sécurité (WS-Security)

### → Inconvénients

- Performances (enveloppe SOAP supplémentaire)
- Complexité, lourdeur
- Cible l'appel de service



## REST

### → Avantages

- Simplicité de mise en œuvre
- Lisibilité par un humain (tests)
- Evolutivité
- Repose sur les principes du web
- Représentations multiples (XML, JSON,...)

### → Inconvénients

- Sécurité restreinte par l'emploi des méthodes HTTP
- Cible l'appel de ressources

# Conclusion

## □ Big WS – SOAP

- Idée : Verbes
- Lourd / complet
- Client lourd
- Fonctionnel
- SOA

## □ WS REST

- Idée : Noms
- Léger / simple
- Client léger et lourd
- ROA / DOA
- A la mode (!) en ce moment dans la cycle client lourd-léger
- Nombreux nouveaux supports natifs (JEE, .NET, NodeJS, RestQL, API réseaux sociaux.....)



# Complément S rialisation Objets

JSON, XML...

## Le problème !

- Relations toujours BI-directionnelles en Java

- Eg :

```
public class User {  
    public int id;  
    public String name;  
    public List<Item> userItems;  
}  
  
public class Item {  
    public int id;  
    public String itemName;  
    public User owner;  
}
```

- Sérialisation XML/JSON ?

## Le problème !

### □ Sériàlisation XML/JSON ?

`com.fasterxml.jackson.databind.JsonMappingException:`

Infinite recursion (StackOverflowError)

(through reference chain:

`org.baeldung.jackson.bidirection.Item["owner"]`

-

`>org.baeldung.jackson.bidirection.User["userItems"]`

`->java.util.ArrayList[0]`

`->org.baeldung.jackson.bidirection.Item["owner"]`

`->.....`

## Solution 1 : on aide Jackson

- *@JsonManagedReference* is the forward part of reference – the one that gets serialized normally.
- *@JsonBackReference* is the back part of reference – it will be omitted from serialization.

```
public class User {  
    public int id;  
    public String name;  
  
    @JsonBackReference  
    public List<Item> userItems;  
}  
  
public class Item {  
    public int id;  
    public String itemName;  
  
    @JsonManagedReference  
    public User owner;  
}
```

## Solution 1 : on aide Jackson

### □ *Résultat :*

```
{  
  "id":2,  
  "itemName":"book",  
  "owner":  
    {  
      "id":1,  
      "name":"John"  
    }  
}
```

## Solution 2 : *@JsonIdentityInfo*

### □ Example :

```
@JsonIdentityInfo(  
    generator =  
    ObjectIdGenerators.PropertyGenerator.class,  
    property = "id")  
public class User { ... }  
  
@JsonIdentityInfo(  
    generator =  
    ObjectIdGenerators.PropertyGenerator.class,  
    property = "id")  
public class Item { ... }
```



### □ Résultat :

```
{  
  "id":2,  
  "itemName":"book",  
  "owner":  
    {  
      "id":1,  
      "name":"John",  
      "userItems":[2]  
    }  
}
```

## Solution 3 : @JsonIgnore

- On ignore l'un des sens de la relation

- Eg

```
public class User {  
    public int id;  
    public String name;  
  
    @JsonIgnore  
    public List<Item> userItems;  
}
```

- Résultat :

```
{  
  "id":2,  
  "itemName":"book",  
  "owner":  
    {  
      "id":1,  
      "name":"John"  
    }  
}
```

## Solution 4 : *@JsonView*

- Des interfaces de marquage (flag)
- Définition dans une classe
- Public ou pas si dans le même package

```
public class Views {  
    public static class Resume {}  
  
    public static class Complet extends Resume {}  
}
```

## Solution 4 : *@JsonView*

### □ Sur les classes :

```
public class User {  
    @JsonView(Views.Resume.class)  
    public int id;  
  
    @JsonView(Views.Resume.class)  
    public String name;  
  
    @JsonView(Views.Complet.class)  
    public List<Item> userItems;  
}  
public class Item {  
    @JsonView(Views.Resume.class)  
    public int id;  
  
    @JsonView(Views.Resume.class)  
    public String itemName;  
  
    @JsonView(Views.Resume.class)  
    public User owner;  
}
```

## Solution 4 : *@JsonView*

- Lors de la sérialisation, on choisit la vue :

```
@JsonView(View.Resume.class)
@RequestMapping(value = "/users", method =
RequestMethod.GET)
public Collection<User> getAllUser() {
    return userRepository.findAll();
}
```

```
@JsonView(View.Complete.class)
@RequestMapping(value = "/users/{id}", method =
RequestMethod.GET)
public User getUserById(@PathVariable Long id) {
    return userRepository.findOne(id);
}
```

## Solution 5 : Custom Serializer/de-

- Contrôle total de la sérialisation/dé-sérialisation
- Eg

```
public class CustomListSerializer extends StdSerializer<List<Item>>{
    public CustomListSerializer() {
        this(null);
    }
    public CustomListSerializer(Class<List> t) {
        super(t);
    }
    @Override
    public void serialize(
        List<Item> items,
        JsonGenerator generator,
        SerializerProvider provider)
        throws IOException, JsonProcessingException {
        List<Integer> ids = new ArrayList<>();
        for (Item item : items) {
            ids.add(item.id);
        }
        generator.writeObject(ids);
    }
}
```