

cours Webservices

Frédéric Moal

Version 2020.0-SNAPSHOT

Sommaire

1. Webservices	1
1.1. Le cours	1
1.2. Sommaire du cours	1
1.3. Organisation	1
1.4. Objectifs	2
1.5. Qu'est ce que c'est ?	2
1.6. Buts	2
1.7. Caractéristiques	2
1.8. Une révolutions ?	3
1.9. Finalités	3
2. Webservices : REST	4
2.1. Références	4
2.2. Sommaire	4
2.3. Communications inter-applications	4
2.4. REST	4
2.5. Protocole HTTP	5
2.6. Et REST ?	10
2.7. les Formats d'échange de données	13
2.8. URIs/datas par l'exemple	14
3. Webservices : Client REST	17
3.1. Et côté client ?	17
3.2. Les différentes possibilités	17
3.3. Clients http	17
3.4. Java – JDK 11+	18
3.5. Jersey	21
3.6. Spring MVC	22
3.7. Feign	24
3.8. JS	24
3.9. Outils de test	25
4. Spring Boot	27
4.1. Principes	27
4.2. Composants	29
4.3. Exemple Spring Data	31
4.4. Conclusion	32
5. Webservices : Serveur REST	33
5.1. 2 librairies nécessaires	33
5.2. Mais alors c'est « simple » ?	33
5.3. Spring MVC	33

5.4. JAX-RS	37
6. Webservices : API Design	45
6.1. Je suis maître du monde [des APIs] !	45
6.2. Deux approches “complémentaires”	45
6.3. Deux approches “complémentaires”	45
6.4. Responsabilité du dev	45
6.5. URLs propres ?	46
6.6. URL propre – Bonnes pratiques	46
6.7. Création d’une ressource REST	47
6.8. Bonnes pratiques - GET	47
6.9. Bonnes pratiques - POST	47
6.10. Bonnes pratiques – POST vs PUT	47
6.11. Négociation de contenu	47
6.12. Ton API tu versionneras	48
6.13. Et les interfaces ?	50
7. Webservices : Sérialisation	71
7.1. JAXB	71
7.2. Complément Sérialisation	82
8. Webservices : Sécurité	88
8.1. Et la sécurité ?	88
8.2. Exemple Spring Sec JWT	89
8.3. Spring Security	93
8.4. bonnes pratiques Default	101
8.5. Petite conclusions ?	104

Chapitre 1. Webservices

1.1. Le cours

Repo git sur Bitbucket :

<https://pdicost.univ-orleans.fr/git/projects/WSI22/repos/cours/>



Code source du cours en AsciiDoc

pour générer les slides ET les supports pdf : `mvn`

pour prendre des notes : papier & crayon et/ou modifier le source ; par exemple avec une balise [NOTE.speaker]



Une note .speaker est incluse dans le pdf généré mais n'apparaît pas explicitement dans les slides

1.2. Sommaire du cours

- Deux architectures
 - REST
 - [SOAP]
- Deux implémentations
 - Spring [SpringBoot]
 - JEE [ReastEasy,Quarkus]
- Deux approches
 - API first
 - Code first

1.3. Organisation

- 12h de Cours
- 18h (6h+12h) de TD/TP, moi, yohan, matthieu
- Du travail perso...
 - Beaucoup
 - Beaucoup, Beaucoup
 - Si si !
- Un CC + Un examen

1.4. Objectifs

- REST
 - Comprendre l'architecture des applications REST
 - Exposer/Consommer des services REST
 - Sécurisation des APIs
 - Design d'API
 - Intégrer des services externes (twitter, facebook, google...)
- SOAP
 - Initiation au protocole et aux Web services SOAP
 - Créer et tester des services SOAP

1.5. Qu'est ce que c'est ?

Un Web Service est une « unité logique applicative » (ensemble de services) accessible en utilisant les protocoles standard

Une «librairie» [API] fournissant des données et des services à d'autres applications

Ils s'appuient sur un ensemble de standards



CE N'EST PAS CE QUE L'ON APPELLE TRADITIONNELLEMENT LE « WEB » (Page html/css...) : on ne peut pas y accéder par un navigateur !!!

1.6. Buts

- But : communication **inter-applications**
 1. Technologie *non adaptée* à une interrogation directe par un utilisateur
 2. Couplage faible
- Trois besoins :
 1. Echange de données
 2. Description des services d'échanges
 3. Découverte des services

1.7. Caractéristiques

Les Web services sont « simples » et « interopérables »

- Normalisés (W3C, Oasis)
- Indépendamment de :
 - la plate-forme (UNIX, Windows, Smartphone...)

- leur implémentation (Java, C++, Visual Basic,...)
- l'architecture sous-jacente (.NET, JEE,...)

1.8. Une révolutions ?

- Non, reposent sur des technologies « anciennes » (XML, annuaires,...) ou détournées (JSON)
- D'autres « technos » équivalentes : CORBA, RMI, DCOM
- Mais la normalisation et l'utilisation de standards permettent l'interopérabilité entre frameworks/plateformes/systèmes (Java, .NET,VMS...) des clients et des serveurs

1.9. Finalités

- Faire interagir des composants hétérogènes, distants et indépendants avec un protocole standard (SOAP,http..)
- Permettre à une application de trouver automatiquement sur un réseau le service dont elle a besoin et d'échanger des données avec lui
- Les Web Services ont été conçus pour intégrer la dimension d'Internet, et la standardisation des échanges
- Les services Web permettent d'interconnecter :
 - Différentes entreprises
 - Différents matériels
 - Différentes applications
 - Différents clients
- Dédiés aux applications B2B (Business to Business), EAI (Enterprise Application Integration) ; plus généralement M2M (Machine to Machine)

En réalité, en entreprise :

- interop Front (JEE/.NET) – Back (Cobol) (eg UR) [ESB],
- Architectures clients légers/mobiles (jQuery, AngularJS, vueJS...)
- Approche SOA / micro-services

Chapitre 2. Webservices : REST

2.1. Références

- Back Java
 - baeldung.com
 - spring.io
 - dzone.com
 - json.org
 - jwt.io
- Veille
 - Les cast codeurs
 - Twitter
 - Blogs OCTO/Ippon...
- pour JS : Michel Buffa, MIAAGE de Nice

2.2. Sommaire

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

2.3. 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 : échanges niveau HTTP, on exploite à 100% le protocole
- SOAP : au dessus d'HTTP ou SMTP ou ..., on encapsule les messages

2.4. REST

- Principe simple → Exploitation de la mécanique HTTP
- Inspiration pour définir l'« architecture » d'accès (interface) à une application

2.4.1. Web Service REST

- Acronyme de REpresentational State Transfert 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 fortement basé sur le protocole HTTP

2.4.2. un Web Service REST...

- 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

2.4.3. REST - utilisation

Utilisé dans le développement des applications orientées 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]

2.5. Protocole HTTP

Pigure de rappel sur le protocole

2.5.1. Le Protocole HTTP

HyperText Transfer Protocol

Protocole d'échanges d'information sur le "web"

Basé sur TCP ⇒ connecté

Version courante 1.1 [2.0]

2.5.2. URL/URI

Unique Resource Location

Identifie les ressources de manière unique

composée de 5 parties :

1. Protocole (http, ftp, smtp, ...)
2. Host (google.com)
3. Port (8080, 80)
4. Path (Chemin vers la ressource sur le serveur)
5. Paramètres

Table 1. Décomposition d'une URL

https://	server:port	/path/to/resource	?p1=v1&p2=v2
Protocole	Nom/ip du serveur et port	Chemin de la ressource	Paramètres

2.5.3. request HTTP

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 envoyée par un client http vers un serveur

Format d'une requête

```
<Méthode> <URI> HTTP/<version>
[<Champ d'entête> : <valeur>]
Accept: <Types MIME du contenu>
Ligne blanche
[Corps de la requête]
```

<Méthode> de la requête : GET, POST, PUT, DELETE, PATCH, ...

<URI> de la ressource (avec paramètres éventuels)

<version> du protocole utilisée : 1.0, 1.1 ou 2.0

<Champ d'entête> : <valeur> : informations sur le contexte de la requête (cookies, localisation,)

Exemple d'entête, "Accept: application/json,application/xml", par ordre de préférence sur le type de contenu souhaité dans la réponse.

<Corps de la requête> : données envoyées au serveur, prise en compte pour les requêtes de type POST ou PUT ou PATCH

2.5.4. response HTTP

Réponse du serveur à une requête d'un client :

- Response Message Header
 - Response Line
 - Response Headers
- Response Message [Optional]

Réponse du serveur à la requête HTTP

Format d'une réponse

```
HTTP / <Version> <Statut> <Commentaire>  
[<Champ d'entête>: <valeur>]  
Content-Type: <Type MIME du contenu>  
Ligne blanche  
<Contenu>
```

<Version> du protocole utilisée : 1.0 ou 1.1 ou 2.0

<Statut> de la réponse, caractérisé par des codes prédéfinis par le protocole http : 200/404/500...

<Commentaire> Information descriptive sur le statut de la réponse, eg "OK"

<Champ d'entête> : <valeur> : informations sur le contexte de la réponse (cookies, localisation, serveur ...)

Exemple d'entête, **Content-Type: application/xml; charset=UTF-8**, qui définit le type MIME du contenu de la réponse.

2.5.5. Exemple de 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
```

- Request Message Body

```
j_idt13:nom:Toto
j_idt13:description:Hello
```

2.5.6. Exemple de Response

- Response Line **HTTP/1.1 200 OK**
- Response Headers

```
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/javax.faces.resource/theme.css?
ln=primefaces-aristo" />
<link type="text/css" rel="stylesheet"
href="/bibliotheque/faces/javax.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>
```

2.5.7. Retours d'information : Status code

Status code **obligatoire**

- 100-level (Informational) — Server acknowledges a request
- 200-level (Success) — Server completed the request as expected
- 300-level (Redirection) — Client needs to perform further actions to complete the request
- 400-level (Client error) — Client sent an invalid request
- 500-level (Server error) — Server failed to fulfill a valid request due to an error with server

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

2.5.8. Status code

Une fiche mémo :

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

2.5.9. De nombreuses autres méthodes...

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

2.6. Et REST ?

Les caractéristiques d'un service REST héritent en partie du protocole http sur lequel il s'appuie

2.6.1. 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/PATCH, DELETE en général)

Les architectures RESTful sont construites à partir de ressources uniquement identifiées par des URI(s)

2.6.2. Requêtes REST : 3 caractéristiques

- 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,...)





2.6.3. 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>

2.6.4. Méthodes (Verbes)

Une ressource peut subir quatre opérations de bases CRUD correspondant aux quatre principaux types de requêtes HTTP (GET, PUT, POST, DELETE)

2.6.5. 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

2.6.6. Méthode GET

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

request

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

response

```
Statut : 200
Message : OK
En-tête : ...
Représentation : XML, JSON, html, ... (type MIME associé)
```

2.6.7. Méthode POST

La méthode POST crée une nouvelle ressource sur le système

request

```
POST http://amazon.fr/bookstore/books
Corps de la requête
Représentation : XML, JSON
```

response

```
Statut : 201, 204  
Message : Create, No content  
En-tête : URI de la ressource créée
```

2.6.8. Méthode DELETE

Supprime la ressource identifiée par l'URI sur le serveur

request

```
DELETE http://amazon.fr/bookstore/books/156  
[Identifiant de la ressource sur le serveur]
```

response

```
Statut : 200 ou 204  
Message : OK, No content  
En-tête : ...
```

2.6.9. Méthode PUT

Mise à jour de la ressource sur le système

Identifiant de la ressource sur le serveur

request

```
PUT http://amazon.fr/bookstore/books/156  
En-tête : .....  
Corps de la requête : XML, JSON,...
```

response

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

2.6.10. 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/PATCH, POST : Le client envoie l'état d'une ressource au serveur

Les données peuvent être sous différents formats [MIME]: JSON, XML, XHTML, CSV, Text/plain, ...

2.7. les Formats d'échange de données

JSON et ses amis

Deux formats "Standards" de fait :

- XML
- JSON

2.7.1. JSON

JSON « JavaScript Object Notation » est un format d'échange de données, facile à lire par un humain et interpréter par une machine.

Deux structures : * Une collection de clefs/valeurs : Object * Une collection ordonnée d'objets : Array

2.7.2. JSON : 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": null
}
```

2.7.3. JSON : ARRAY

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


```
[
  { "id": 51,
    "nom": "Mathematiques 1",
    "resume": "Resume of math ",
    "isbn": "123654",
    "quantite": 42
  },
  { "id": 102,
    "nom": "Mathematiques 4",
    "resume": "Resume of math ",
    "isbn": "12365444455",
    "quantite": 18
  }
]
```

2.7.4. JSON : Value

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

2.7.5. XML vs JSON

- XML est standard
- Manipulation aisée
- Verbeux
- Schéma XSD, DTD, XSLT, ...
- JSON est créé pour les navigateurs web (JS !)
- Interprété nativement
- Supporté par tous les langages moyennant une API spécifique (Jackson, JAXB, ...)

Pour les deux, cf RefCardz sur dzone.com

2.8. URIs/datas par l'exemple

2.8.1. Un 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

2.8.2. 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>
```

2.8.3. 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>
```

2.8.4. 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>
```

2.8.5. Exemple : Suppression d'un sondage

```
DELETE /sondage/672609683
```

```
200 OK
```

```
GET /sondage/672609683
```

```
404 Not Found
```

2.8.6. Asynchronisme

HTTP est un protocole synchrone

On peut simuler l'asynchrone en utilisant des files d'attente :

```
POST /queue
```

```
202 Accepted
Location: /queue/message/1
```

Et ensuite lecture des résultats sur :

```
GET /queue/message/1
```

Chapitre 3. Webservices : Client REST

3.1. Et côté client ?

Comment on appelle une API REST ?

3.2. Les différentes possibilités

- Client http “génériques”
 - Telnet (hum !)
 - Curl
 - Httpie
 - Soapui
 - Karaté
 - IntelliJ, Eclipse, ...
- Codage client [Java, .NET, ...]
 - Java (JDK natif)
 - Java : Jersey
 - Java : Spring MVC
 - JS natif ou JQuery ou ...

3.3. Clients http

Quelques exemples... cf man

3.3.1. cUrl

Le + utilisé ; dans un script bash, avec **jq** pour le parsing du json

```
curl -X POST -H "Content-Type: application/json"
      -d @body.json httpbin.org/post
curl -v -u user:pwd httpbin.org/basic-auth/user/pwd
curl -k monsitededevquinapasdecertificathttps.fr
curl example.com --resolve example.com:80:127.0.0.1
curl -s httpbin.org/get | jq .headers

# get authorization token header
TOKEN=`curl -i -s -X POST -d login=fred -d password=fred http://localhost:8000/login |
grep Authorization:`
# request with token header
curl -H "$TOKEN" http://localhost:8000/checkToken
```

3.3.2. httpie

en python ; syntaxe simplifiée ; JSON par défaut

Exemple

```
# recup liste des sondages [default : JSON]
http GET :8080/sondage

# recup liste des sondages en XML
http GET :8080/sondage Accept:application/xml

http GET :8080/sondage/1 Accept:application/xml

# creation d'un sondage [passage par forms avec -f]
http -f POST :8080/sondage question="oui ou non ?" propositions=oui,non,ptet

# vote sur le sondage précédent
http -f POST :8080/sondage/2/vote votant=fred choix=ptet
```

3.3.3. Karaté

Suite complète permettant d'écrire des tests (eg en TDD)

<https://github.com/intuit/karate>

Exemple

```
Feature: karate 'hello world' example
Scenario: create and retrieve a cat

Given url 'http://myhost.com/v1/cats'
And request { name: 'Billie' }
When method post
Then status 201
And match response == { id: '#notnull', name: 'Billie' }

Given path response.id
When method get
Then status 200
```

3.4. Java – JDK 11+

La nouvelle API HTTP Client propose un support des versions 1.1 et 2 du protocole HTTP ainsi que les WebSockets côté client.

L'API est fournie en standard dans Java 11+

Nouvelle API + “moderne”

- utilisation du design pattern builder
- utilisation de fabriques pour obtenir des instances des builder mais aussi d'implémentations de certaines interfaces pour des usages courants
- utilisation de l'API Flow (reactive streams) pour fournir les données du body d'une requête (Flow.Publisher) et consommer le body d'une réponse (Flow.Subscriber)

3.4.1. Java – HttpClient

L'obtention d'une instance de type HttpClient peut se faire de deux manières :

- utilisation de la fabrique `newHttpClient()` de la classe HttpClient pour obtenir une instance avec la configuration par défaut

```
HttpClient client = HttpClient.newHttpClient();
```

- utilisation de l'interface HttpClient.Builder qui est un builder pour configurer l'instance obtenue. Une instance du Builder est obtenue en utilisant la fabrique `newBuilder()` de la classe HttpClient

```
HttpClient httpClient = HttpClient.newBuilder()  
    .version(HttpClient.Version.HTTP_1_1)  
    .build();
```

3.4.2. Java – HttpRequest

Elle encapsule une requête à envoyer à un serveur.

Une instance de type HttpRequest est obtenue en utilisant un builder de type HttpRequest.Builder. Une instance de l'interface HttpRequest.Builder est obtenue en utilisant la fabrique `newBuilder()` de la classe HttpRequest.

Plusieurs méthodes permettent de configurer les éléments de la requête : l'uri, le verbe HTTP, les headers, un timeout.

```
HttpRequest request = HttpRequest.newBuilder()  
    .uri(URI.create("http://www.oxiane.com/"))  
    .GET()  
    .build();
```

3.4.3. Java – HttpRequest

Plusieurs méthodes permettent de configurer les éléments de la requête : l'uri, le verbe HTTP, les headers, un timeout.

```
HttpRequest requetePost = HttpRequest.newBuilder()
    .uri(URI.create("http://demo.com/api/ex"))
    .setHeader("Content-Type", "application/json")
    .POST(BodyPublishers.ofString("{\"cle1\":\"valeur1\",\"cle2\":\"valeur2\"}"))
    .build();
```

3.4.4. Java – send

Envoi en mode **synchrone**

- méthode `send()` de la classe `HttpClient`. Elle attend en paramètres :
 - la requête
 - une instance de type `HttpResponse.BodyHandler` qui permet de traiter le body selon la réponse
- La classe `HttpBodyHandlers` propose des fabriques pour obtenir des instances de `BodyHandler` pour des usages courants.

```
HttpResponse response;
try {
    response = httpClient.send(requete, BodyHandlers.ofString());
    System.out.println("Status: " + response.statusCode());
    System.out.println("Headers: " + response.headers());
    System.out.println("Body: " + response.body());
} catch (IOException | InterruptedException e) {
    ...
}
```

3.4.5. Java – sendAsync

Envoi en mode **asynchrone**

- Il se fait en utilisant la méthode `sendAsync()` de la classe `HttpClient`. Elle attend en paramètres :
 - la requête
 - une instance de type `HttpResponse.BodyHandler` qui permet de traiter le body selon la réponse
- Elle renvoie un `CompletableFuture` qui permet de définir les traitements à exécuter à la réception de la réponse.

```
httpClient.sendAsync(requete, BodyHandlers.ofString())
    .thenAccept(response -> {
        System.out.println("Status: " + response.statusCode());
        System.out.println("Headers: " + response.headers());
        System.out.println("Body: " + response.body());
    })
    );
```

3.5. Jersey

Librairie Java(EE) client et serveur pour REST

3.5.1. Exemple de Client Jersey

- 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");
```

3.5.2. Exemple de Client Jersey 2

- 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);
}

```

3.5.3. 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);

```

3.6. Spring MVC

Librairie proposée par Spring pour application Web (<http>)

3.6.1. Exemple de client Spring

- Création du client (opération lourde) :

```

RestTemplate restTemplate = new RestTemplate();

```

- Opérations sur les ressources :

```

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

```

3.6.2. 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);
```

3.6.3. Spring 5

Depuis Spring 5, WebClient "remplace" RestTemplate

Attention, **Reactive Stream** pour l'API

```
String URL = "http://localhost:8080/api/v3/pet/findByStatus?status=available";

WebClient.RequestHeadersUriSpec<?> request = WebClient.create(URL).get();
ClientResponse response = request.exchange().block();

Mono<Pet[]> body = response.bodyToMono(Pet[].class);

System.out.println(response.statusCode());
System.out.println(body.block()[0].name);
```

Pour l'activer, rajouter les dépendances dans le pom.xml :

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-webflux</artifactId>
  <version>5.2.3.RELEASE</version>
</dependency>
<dependency>
  <groupId>io.projectreactor.netty</groupId>
  <artifactId>reactor-netty</artifactId>
  <version>0.9.4.RELEASE</version>
</dependency>
```



```

<script language="javascript">
    var xmlhttp;
    function initialisation() {
        // put more code here in case you are concerned
        // about browsers that do not provide XMLHttpRequest object directly
        xmlhttp = new XMLHttpRequest();
        requete();
    }

    function requete() {
        var url = "http://localhost:8080/sondages";
        xmlhttp.open('GET',url,true);
        xmlhttp.send(null);

        xmlhttp.onreadystatechange = function() {
            if (xmlhttp.readyState == 4) {
                if ( xmlhttp.status == 200) {
                    var listeSondages = xmlhttp.responseText;
                    var listeIdentifiants = JSON.parse(listeSondages);
                    var select = document.getElementById("mesSondages");
                    ...
                }
            }
        };
    }

```

3.8.2. JQuery

```

<script src="/webjars/jquery/jquery.min.js"></script>

<script language="javascript">
    $(document).ready(function() {
        $.ajax({
            url: "http://localhost:8080/sondages/1", method:'POST', data:'YEP'
        }).then(function(data) {
            $('#sondage-id').append(data.id);
            $('#sondage-libelle').append(data.question);
        });
    });
    ...

```

3.9. Outils de test

Il existe de nombreux outils [en ligne] permettant de tester les services 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, Karaté

Chapitre 4. Spring Boot

Spring sans les mains !

4.1. Principes

principe de base : COC

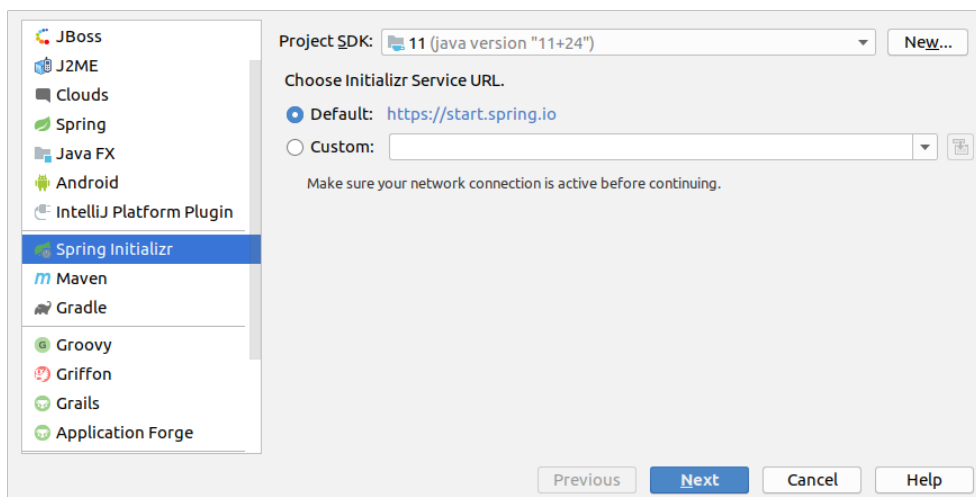
- Utilisation d'un pom parent [gestion des versions]
- Auto-configuration de Spring
 - à partir du classpath [dépendances du pom]
 - auto-scan des annotations
- Création d'un JAR

4.1.1. Démarrage

Création d'un projet :

- Page web <http://start.spring.io>
- Depuis une commande curl
- en utilisant Spring Boot command-line interface
- new project "Spring Initializr" avec STS, IntelliJ IDEA ou NetBeans

4.1.2. Exemple avec IntelliJ



Project Metadata

Group:

Artifact:

Type:

Language:

Packaging:

Java Version:

Version:

Name:

Description:

Package:

Dependencies

Dependencies	Selected Dependencies
Core	Core
Web	Web
Template Engines	SQL
SQL	Ops
NoSQL	
Integration	
Cloud Core	
Cloud Support	
Cloud Config	
Cloud Discovery	
Cloud Routing	
Cloud Circuit Breaker	
Cloud Tracing	
Cloud Messaging	
Cloud AWS	
Cloud Contract	
Pivotal Cloud Foundry	
Azure	
Spring Cloud GCP	
I/O	

4.1.3. Résultat

Initializr génère un squelette de projet avec les choix effectués

- langage (java/kotlin/groovy)
- build maven/gradle
- composants sélectionnés (et répertoires de contexte, eg web)
- un main (Java) et un main de Test
- un plugin `spring-boot-maven-plugin`

4.1.4. Packaging

Le packaging par défaut de Spring Boot est un (fat)jar, même quand on fait une application web !

On peut demander un war a deploy, mais c'est déconseillé.

4.1.5. Lancement

Run du projet

- directement du main depuis l'IDE
- en utilisant le plugin :
 - spring-boot:run
 - spring-boot:start
- `mvn package && java -jar target/{project_id}-{version}.jar`

Logs de démarrage dans la console

4.2. Composants

4.2.1. Fonctionnalités

De base, fournit l'autoconfiguration et les starters pour les dépendances ⇒ simplifie le pom et la configuration

Ajoute des fonctionnalités par composants (starters) eg

- Actuator : info de runtime (métriques, status, env...)
- DevTools : push des modifs dans le navigateur [plugin Live Reload à installer dans le navigateur]
- Properties faciles à définir
- Support étendu natif pour les tests

4.2.2. Composants thématiques

En plus de Spring Core, Web, ...

- Spring Data : interfaces de DAO
- Spring Security
- Spring integration & Spring Batch
- Spring Cloud [micro-services]
- Reactive Spring

⇒ starters

4.2.3. Configuration par défaut :


```
@SpringBootApplication
public class DemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
}
```

Auto-scan le code (annotations, @Configuration...) par défaut dans les **sous-packages** de la classe principale

4.2.4. Annotation @SpringBootApplication

Ajoute automatiquement :

- @Configuration tags la classe comme la source des définitions de bean pour le contexte.
- @EnableAutoConfiguration dis à Spring Boot d'ajouter des beans et des settings en fonction du classpath.
- @ComponentScan active le scan des composants, configurations, et des services dans le package.

4.2.5. application.properties

Fichier de configuration à la racine de ressources (pour configuration BD, ...)

clés de propriétés prédéfinies utilisables

On peut utiliser un yaml à la place

4.2.6. Exemple Data Jpa

```
spring.jpa.hibernate.ddl-auto=create
spring.datasource.url=jdbc:mysql://localhost:3306/banque
spring.datasource.username=springuser
spring.datasource.password=ThePassword
```

4.2.7. Remplacement de composants par défaut

Exemple avec Hibernate ⇒ EclipseLink

```

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
  <exclusions>
    <exclusion>
      <artifactId>hibernate-entitymanager</artifactId>
      <groupId>org.hibernate</groupId>
    </exclusion>
  </exclusions>
</dependency>

<dependency>
  <groupId>org.eclipse.persistence</groupId>
  <artifactId>eclipselink</artifactId>
  <version>2.5.2</version>
</dependency>

```

4.3. Exemple Spring Data

Projet Spring réunissant de nombreux sous-projets, suivant la base de données

- Spring Data JPA
 - JPA persistence against a relational database
- Spring Data MongoDB
- Spring Data Neo4j
- Spring Data Redis
- Spring Data Cassandra

4.3.1. Configuration dans un projet

Utilise SpringBoot, ajout d'une dépendance starter, eg JPA :

```

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>

```

4.3.2. Ajout driver JDBC

Autoscan fournit le driver de la datasource

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
</dependency>
```

4.3.3. Dao en interface

Opération CRUD générée + méthodes par nommage !

```
public interface ClientRepository extends CrudRepository<Client,Long> {
    List<Client> findByNom(String nom);
}
```

4.3.4. Exemple d'utilisation

```
@Bean
public CommandLineRunner demo(ClientRepository repository) {
    return (args) -> {
        repository.save(new Client("Jack", "Bauer"));
        repository.save(new Client("Chloe", "O'Brian"));
        repository.save(new Client("Kim", "Bauer"));

        for (Client client : repository.findAll()) {
            log.info(client.toString());
        }
        repository.findByNom("Bauer").forEach(bauer -> {
            log.info(bauer.toString());
        });
    };
};
```

4.4. Conclusion

Spring Boot indispensable sur tout nouveau projet Spring !

cf <https://spring.io/guides>

Chapitre 5. Webservices : Serveur REST

Développer des Web Services REST avec JAVA

5.1. 2 librairies nécessaires

- 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 / Spring MVC : comment déployer des services REST en Java (mapping requête → méthode)
 - JAXB / Jackson : comment traduire des objets java en xml/json/... de manière « automatique »

5.2. Mais alors c'est « simple » ?

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

5.3. Spring MVC

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

5.3.1. Annotations Spring MVC

- `@RestController`
- `@RequestMapping(method=?)` : classe et/ou méthode, avec raccourcis :
 - `@GetMapping`
 - `@PostMapping`
 - `@PutMapping`
 - `@PatchMapping`
 - `@DeleteMapping`

5.3.2. Annotations Spring MVC

- Paramètres (injection dans paramètres):
 - `@RequestParam`

- @PathVariable
- @...

Démo ; Cf <https://projects.spring.io/spring-framework/> dans la bonne version

5.3.3. Exemple 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();
    }
}
```

5.3.4. 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;
    }
    ...
}
```

5.3.5. 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);
    }
    ...
}

```

5.3.6. Création de données : KO

```

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

```

5.3.7. Création de données : demi-OK sans Location !!!

```

@PostMapping(consumes="application/json")
public ResponseEntity<Taco> postTaco(@RequestBody Taco taco) {
    if (OK) {
        Taco save = tacoRepo.save(taco);
        return new ResponseEntity<>(save, HttpStatus.CREATED);
    }
    return new ResponseEntity<>(null, HttpStatus.BAD_REQUEST);
}

```

5.3.8. Création de données : OK avec Location

```

@PostMapping("/user")
public ResponseEntity<Utilisateur> inscrire(
    @RequestBody Utilisateur user,
    UriComponentsBuilder base) {
    try {
        user.setNom(user.getNom().toLowerCase());
        facade.inscrire(user);
    } catch (UtilisateurDejaExistantException e) {
        return ResponseEntity.status(HttpStatus.CONFLICT).build();
    }
    URI location = base.path("/user/{nom}")
        .buildAndExpand(user.getNom()).toUri();

    return ResponseEntity.created(location).body(user);
}

```

5.3.9. MAJ de données

Option 1 :

```

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

```

5.3.10. MAJ de données

Option 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);
}

```

5.3.11. DEL de données

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

5.3.12. Annotations Spring MVC

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 + "'.");
    }
}
```

5.3.13. Déploiement

- Utilisation de SpringBoot
- Les starters à regarder : spring-bootstarter-*
 - Web !
 - Actuator
 - Devtools
 - Jackson-dataformat-*
 - Hateoas
 - Security

5.4. JAX-RS

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 backend
- Son architecture se repose sur l'utilisation des classes et des annotations pour développer les services web

5.4.1. JAX-RS - Implémentation

JAX-RS est une **spécification** et autour de cette spécification sont développés plusieurs implémentations :

- JERSEY : implémentation de référence (<https://eclipse-ee4j.github.io/jersey/>)
- CXF : Apache (<http://cfx.apache.org>)
- RESTEasy : JBoss
- RESTLET : L'un des premiers framework implémentant REST pour Java

5.4.2. JERSEY

Version actuelle 2.30 implémentant les spécifications de JAX-RS 2.1

- Intégré dans Glassfish et l'implémentation Java EE (6,7,8)
- Anciennement supportés dans Netbeans/Glassfish
- maintenant Jakarta EE

5.4.3. JAX-RS : Développement

- 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

5.4.4. Annotations

JAX-RS utilise des annotations Java pour exposer les services REST

- @Path : Chemin d'accès à la ressource
- @GET □ @PUT, @POST □ @DELETE : Type de requête (verbe http)
- @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

5.4.5. JAX-RS : contextes URI concaténés

Les URIs des ressources sont générées par concaténation des contextes

- de déploiement : le serveur
- de l'application, `@ApplicationPath`
- des ressources locales `@Path`

5.4.6. JAX-RS : `@ApplicationPath` contexte

Configuration du contexte global de l'application

- `@ApplicationPath("api")`
- Concaténé au contexte de deploy
- Racine des ressources `@Path`

5.4.7. 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("categorie")
public class CategoryFacade {
    ...
}
```

5.4.8. JAX-RS : contextes concaténés

Exemple : URI <http://localhost:8080/Bibliotheque/api/categorie>

Contexte de déploiement de l'application : <http://localhost:8080/Bibliotheque/> (serveur)

Contexte de l'application JAX-RS : **api** * `@ApplicationPath("api")`

Contexte de la ressource : **categorie** `@Path("categorie")`

5.4.9. 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 `@Path` de la classe et celle de la méthode

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

<http://localhost:8080/Bibliotheque/api/categorie/hello>

5.4.10. JAX-RS : @Path dynamique

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;
}
```

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

5.4.11. JAX-RS : @Path dynamique

On peut limiter le match des paths dynamique avec des expressions régulières

Exemple :

```
@Path("users/{username: [a-zA-Z][a-zA-Z_0-9]*}")
```

5.4.12. @GET, @POST, @PUT, @DELETE

Restriction des mappings par verbe HTTP sur les méthodes Java

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

`curl -X GET http://localhost:8080/Bibliotheque/api/categorie/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;
}
```

`curl -X GET http://localhost:8080/Bibliotheque/api/categorie/fred`

5.4.13. @GET, @POST, @PUT, @DELETE

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

Exemple :

Table 2. URIs du service

URI	GET	POST	PUT	DELETE
/livre	Liste des livres	Créer un nouveau livre	X	X
/livre/{id}	Renvoie le Livre identifié par l'id	X	Mise à jour du livre identifié par id	Supprimer le livre identifié par id

5.4.14. @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();
    }
    ...
}

```

5.4.15. @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);
}

```

5.4.16. @GET, @POST, @PUT, @DELETE

Pensez au range pour les collections !

```

@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});
}
}

```

5.4.17. 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

5.4.18. 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

5.4.19. Paramètres des requêtes

Pour les objets en paramètre, il faut :

1. être un type primitif,
2. avoir un constructeur(String)
3. avoir une méthode `static` `valueOf(String)` ou `fromString(String)` (eg `Integer.valueOf(String)` `java.util.UUID.fromString(String)`);
4. avoir une (register) implémentation de `javax.ws.rs.ext.ParamConverterProvider` qui retourne une instance de `javax.ws.rs.ext.ParamConverter` capable de conversion de/vers String
5. être une collection `List<T>`, `Set<T>` or `SortedSet<T>`, où T satisfait 2 ou 3

5.4.20. Et la response ?

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

5.4.21. De nombreuses autres subtilités

Cf Doc Jersey

<https://eclipse-ee4j.github.io/jersey.github.io/documentation/latest/index.html>

Chapitre 6. Webservices : API Design

6.1. Je suis maître du monde [des APIs] !

Design de Web Services REST By example

6.2. Deux approches “complémentaires”

- 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...

6.3. 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)
- ...

6.4. Responsabilité du dev

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 !



Figure 1. La coulée de lave se produit lorsqu'une partie de code encore immature est mise en production, forçant la lave à se solidifier en empêchant sa modification

6.5. 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>

6.6. URL propre – Bonnes pratiques

- 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

6.7. Création d'une ressource REST

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

6.8. Bonnes pratiques - GET

- Opération de **lecture**
- Idempotente
- Possibilité de cache

6.9. Bonnes pratiques - POST

- Opération de création
- Effet de bord possible sur le serveur
- Positionné le header **Location** dans la response

6.10. Bonnes pratiques – POST vs PUT

- Identifier une ressource créée
- Préferer

```
POST /sondage/672609683/vote
```

```
201 CREATED
```

```
Location: /sondage/672609683/vote/1
```

à

```
PUT /sondage/672609683/vote/1
```

```
200 OK
```

6.11. Négociation de contenu

On **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
```

6.12. Ton API tu versionneras

Les APIs doivent être versionnées

pour leurs permettre d'évoluer dans le temps [vs coulée de lave]

Deux solutions usuelles :

- ajouter la version de l'API dans l'URL (/api/v1/...)
- préciser la version dans le type mime du header **Accept** de la requête [eg Github]

6.12.1. Des noms tu utiliseras, jamais de verbes

Utilisez des **NOMS** pour les URIs, jamais de **VERBES** !

Voici quelques **contre-exemples INTERDITS** :

```
/getProducts
/listOrders
/retrieveClientByOrder?orderId=1
```

utilisez à la place :

```
GET /products : will return the list of all products
POST /products : will add a product to the collection
GET /products/4 : will retrieve product #4
PATCH/PUT /products/4 : will update product #4
```

6.12.2. GET et HEAD tu respecteras

RFC2616 définit clairement que les opérations HEAD et GET ne **doivent pas** altérer l'état du système.

Très mauvaise pratique : **GET** /deleteProduct?id=1

Imaginez le résultat de l'indexation des URLs par un moteur de recherche !

6.12.3. Des sous-ressources tu utiliseras

Si vous voulez récupérer une sous-collection d'un objet, utilisez une sous-uri pour cette ressource.

Par exemple, pour obtenir la liste des albums d'un artiste, utilisez `GET /artists/8/albums`

6.12.4. Des résultats paginés tu renverras

Ne pas retourner, sans contrôle, des collections qui peuvent être grandes !

Cela pose des problèmes de latence sur le transport des données, mais aussi de sérialisation/désérialisation sur le serveur et le client.

la solution est de **paginer** systématiquement les requêtes (cf Facebook, Twitter, Github,...) avec des paramètres du type from/to, start/size...

Une autre bonne pratique est d'indiquer dans l'entête HTTP de la réponse l'URI de la page précédente et de la page suivante.

6.12.5. 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 permanently deleted and will never be available again.
 - 422 Unprocessable entity Could be used when there was a validation error while creating an object.

liste complète dans la RFC2616 : <https://tools.ietf.org/html/rfc2616#section-10>

6.12.6. 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-801202e7b89859ce"
}
```

6.12.7. Gestion de version : URLs et MIME types

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"
```

6.13. Et les interfaces ?

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

6.13.1. Description de l'interface d'un WS REST

3 possibilités :

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

6.13.2. WADL

Web Application Definition Language est un langage de description des services REST, au format XML.

- une spécification du W3C initiée par SUN (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

6.13.3. WADL

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)

6.13.4. WADL : exemple

6.13.5. WADL

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

6.13.6. Swagger / Open API

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

6.13.7. Swagger / Open API

```
swagger: "2.0"
info:
  description: "This is a sample server Petstore server. You can find out more about
  Swagger
  at [http://swagger.io](http://swagger.io) or on [irc.freenode.net,
  #swagger](http://swagger.io/irc/).
  For this sample, you can use the api key `special-key` to
  test the authorization
  filters."
  version: "1.0.0"
  title: "Swagger Petstore"
  termsOfService: "http://swagger.io/terms/"
  contact:
    email: "apiteam@swagger.io"
  license:
    name: "Apache 2.0"
    url: "http://www.apache.org/licenses/LICENSE-2.0.html"
  host: "petstore.swagger.io"
  basePath: "/v2"
  tags:
    - name: "pet"
      description: "Everything about your Pets"
      externalDocs:
        description: "Find out more"
        url: "http://swagger.io"
    - name: "store"
      description: "Access to Petstore orders"
    - name: "user"
      description: "Operations about user"
      externalDocs:
        description: "Find out more about our store"
        url: "http://swagger.io"
  schemes:
    - "http"
```

6.13.8. Swagger / Open API

```
paths:
  /pet:
    post:
      tags:
        - "pet"
      summary: "Add a new pet to the store"
      description: ""
      operationId: "addPet"
      consumes:
        - "application/json"
        - "application/xml"
      produces:
        - "application/xml"
        - "application/json"
      parameters:
        - in: "body"
          name: "body"
          description: "Pet object that needs to be added to the store"
          required: true
          schema:
            $ref: "#/definitions/Pet"
      responses:
        405:
          description: "Invalid input"
      security:
        - petstore_auth:
            - "write:pets"
            - "read:pets"
```

6.13.9. Swagger / Open API


```
put:
tags:
- "pet"
summary: "Update an existing pet"
description: ""
operationId: "updatePet"
consumes:
- "application/json"
- "application/xml"
produces:
- "application/xml"
- "application/json"
parameters:
- in: "body"
name: "body"
description: "Pet object that needs to be added to the store"
required: true
schema:
$ref: "#/definitions/Pet"
responses:
400:
description: "Invalid ID supplied"
404:
description: "Pet not found"
405:
description: "Validation exception"
security:
- petstore_auth:
- "write:pets"
- "read:pets"
```

6.13.10. Swagger / Open API

```

/pet/findByTags:
  get:
    tags:
      - "pet"
    summary: "Finds Pets by tags"
    description: "Muliple tags can be provided with comma separated strings. Use
tag2, tag3 for testing."
    operationId: "findPetsByTags"
    produces:
      - "application/xml"
      - "application/json"
    parameters:
      - name: "tags"
        in: "query"
        description: "Tags to filter by"
        required: true
        type: "array"
        items:
          type: "string"
        collectionFormat: "multi"
    responses:
      200:
        description: "successful operation"
        schema:
          type: "array"
          items:
            $ref: "#/definitions/Pet"
      400:
        description: "Invalid tag value"
    security:
      - petstore_auth:
        - "write:pets"
        - "read:pets"
    deprecated: true

```

tag1,

6.13.11. Swagger / Open API

```
/pet/{petId}:
get:
tags:
- "pet"
summary: "Find pet by ID"
description: "Returns a single pet"
operationId: "getPetById"
produces:
- "application/xml"
- "application/json"
parameters:
- name: "petId"
in: "path"
description: "ID of pet to return"
required: true
type: "integer"
format: "int64"
responses:
200:
description: "successful operation"
schema:
$ref: "#/definitions/Pet"
400:
description: "Invalid ID supplied"
404:
description: "Pet not found"
security:
- api_key: []
```

6.13.12. Swagger / Open API

```

post:
  tags:
  - "pet"
  summary: "Updates a pet in the store with form data"
  description: ""
  operationId: "updatePetWithForm"
  consumes:
  - "application/x-www-form-urlencoded"
  produces:
  - "application/xml"
  - "application/json"
  parameters:
  - name: "petId"
    in: "path"
    description: "ID of pet that needs to be updated"
    required: true
    type: "integer"
    format: "int64"
  - name: "name"
    in: "formData"
    description: "Updated name of the pet"
    required: false
    type: "string"
  - name: "status"
    in: "formData"
    description: "Updated status of the pet"
    required: false
    type: "string"
  responses:
    405:
      description: "Invalid input"
  security:
  - petstore_auth:
  - "write:pets"
  - "read:pets"

```

6.13.13. Swagger / Open API

```
securityDefinitions:
  petstore_auth:
    type: "oauth2"
    authorizationUrl: "http://petstore.swagger.io/oauth/dialog"
    flow: "implicit"
    scopes:
      write:pets: "modify pets in your account"
      read:pets: "read your pets"
  api_key:
    type: "apiKey"
    name: "api_key"
    in: "header"
```

6.13.14. Swagger / Open API

```

definitions:
  Order:
    type: "object"
    properties:
      id:
        type: "integer"
        format: "int64"
      petId:
        type: "integer"
        format: "int64"
      quantity:
        type: "integer"
        format: "int32"
      shipDate:
        type: "string"
        format: "date-time"
      status:
        type: "string"
        description: "Order Status"
        enum:
          - "placed"
          - "approved"
          - "delivered"
      complete:
        type: "boolean"
        default: false
    xml:
      name: "Order"
  Category:
    type: "object"
    properties:
      id:
        type: "integer"
        format: "int64"
      name:
        type: "string"

```

6.13.15. Exemple code → docs/spec

Il existe des bibliothèques capables de scanner les annotations REST dans le code pour générer la spécification en OpenAPI ou Swagger.

Avec Spring boot : springdoc-openapi génère une spécification en OpenAPI 3.0

```

<dependency>
  <groupId>org.springdoc</groupId>
  <artifactId>springdoc-openapi</artifactId>
  <version>1.3.0</version>
</dependency>
<!-- pour avoir l'UI Swagger en plus -->
<dependency>
  <groupId>org.springdoc</groupId>
  <artifactId>springdoc-openapi-ui</artifactId>
  <version>1.3.0</version>
</dependency>
<!-- version pour webflux -->
<dependency>
  <groupId>org.springdoc</groupId>
  <artifactId>springdoc-openapi-webflux-ui</artifactId>
  <version>1.3.0</version>
</dependency>

```

Des URLs sont automatiquement déployées :

la spec en JSON : <http://localhost:8080/v3/api-docs/>

la spec en Yaml : <http://localhost:8080/v3/api-docs.yaml>

Swagger UI, ihm web : <http://localhost:8080/swagger-ui.html>

6.13.16. OpenAPI tuning

Pour modifier ces URLs par défaut, on ajoute dans la config (properties) :

```

springdoc.api-docs.path=/api-docs
springdoc.swagger-ui.path=/swagger-ui-custom.html

```

6.13.17. Compléter cette documentation automatique

On peut(doit) ajouter des annotations dans le code pour ajouter des précisions/infos/descriptions dans la spécification générée.

Par exemple :

```

@Tag(name = "Doudeule", description = "API de gestion des rendez-vous")
@Operation(description = "renvoie la liste de tous les rendez-vous")
@Parameter(name = "rdv", description = "le rendez-vous à créer, sans son id")

```

cf documentation en ligne : <https://springdoc.org/>

6.13.18. HATEOAS

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

6.13.19. HATEOAS exemples

```
{
  "content": "Hello, World!",
  "_links": {
    "self": {
      "href": "http://localhost:8080/greeting?name=World"
    }
  }
}
```

6.13.20. HATEOAS : ressource

```
public class Greeting extends RepresentationModel<Greeting> {

    private final String content;

    @JsonCreator
    public Greeting(@JsonProperty("content") String content) {
        this.content = content;
    }

    public String getContent() {
        return content;
    }
}
```

6.13.21. HATEOAS : controller


```

@RestController
public class GreetingController {

    private static final String TEMPLATE = "Hello, %s!";

    @RequestMapping("/greeting")
    public HttpEntity<Greeting> greeting(
        @RequestParam(value = "name", defaultValue = "World") String name) {

        Greeting greeting = new Greeting(String.format(TEMPLATE, name));

        greeting.add(linkTo(methodOn(GreetingController.class).greeting(name)).withSelfRel());

        return new ResponseEntity<>(greeting, HttpStatus.OK);
    }
}

```

6.13.22. RETOUR RECAP

CODING BY EXAMPLE :

Imaginons qu'une société veuille gérer les appareils qu'elle confie à ses collaborateurs afin de savoir qui utilise quel appareil.

Dans notre hypothèse de départ, nous pouvons imaginer les opérations suivantes :

- Pour la gestion des personnes
 - Créer une personne
 - Récupérer la liste des personnes existantes
 - Récupérer les informations détaillées d'une personne
 - Mettre à jour les informations concernant une personne
 - Supprimer une personne
- Pour la gestion des appareils
 - Créer un appareil
 - Récupérer la liste des appareils
 - Récupérer les informations relatives à un appareil
 - Mettre à jour les informations relatives à un appareil
 - Supprimer un appareil
- Pour la gestion de « qui à quoi »
 - Lier un utilisateur à un appareil
 - Supprimer le lien entre un utilisateur et un appareil

(Par soucis de facilité, nous n'allons pas gérer ici le fait de transférer l'attribution d'un appareil

d'un utilisateur à un autre, il suffit de supprimer le lien et ensuite de le recréer avec un nouvel utilisateur)

Voyons comment mettre en œuvre des APIs sur base de notre exemple de départ et regardons quelles sont les bonnes pratiques à respecter.

6.13.23. Bonne Pratique – Utiliser correctement les verbes HTTP

Bien qu'il n'y ait aucune limitation technique, la règle communément admise pour l'utilisations des verbes http est la suivante :

HTTP VERB	Action	Description
GET	select	Lecture d'une information
POST	insert	Écrire une information
PUT	update	Mettre à jour une information
DELETE	Delete	Supprimer une information

6.13.24. Bonne Pratique – Utiliser des noms et pas de verbes dans votre URL

Prenons en exemple l'opération de récupérer tous les utilisateurs enregistrés.

La bonne pratique est de retourner la liste des utilisateurs pour une requête « GET » (pour rappel, nous lisons une information) pour l'url /users

VERB	URL	Description
GET	/users	Récupération des identifiants de chaque utilisateur.

A ne pas faire :

VERB	URL	Description
GET	/getallusers	Récupération des identifiants de chaque utilisateur.
GET	/allusers	Récupération des identifiants de chaque utilisateur.

Sur base de cette première API, nous pouvons en déduire 2 nouvelles bonnes pratiques.

6.13.25. Bonne Pratique – Utiliser le pluriel pour nommer vos ressources

Lorsque l'URL se rapporte à un groupe de ressources (ici nous parlons bien de plusieurs utilisateurs et non pas d'un seul), le pluriel s'impose pour indiquer aux développeurs que l'URL se rapporte à une collection de ressources.

6.13.26. Bonne Pratique – Utiliser correctement les codes de retour

Le protocole http nous offre une grande flexibilité dans les codes de retour que notre API peut envoyer en réponse, il faut les utiliser là où ils ont un sens.

Code de retour	Message	Description et utilisation
200	OK	Le serveur a traité la requête avec succès.
201	CREATED	Une nouvelle ressource a été créée.
204	No Content	Peut être utilisée en réponse à une requête DELETE effectuée avec succès.
206	Partial Content	En réponse à une requête demandant une réponse trop lourde pour être envoyée en une seule fois. De la pagination va être nécessaire pour récupérer l'ensemble des informations
304	Not Modified	Le client peut utiliser les données en cache car elles n'ont pas été modifiées depuis la date spécifiée.
400	Bad Request	La requête est invalide et ne peut pas être traitée par le serveur.
401	Unauthorized	La requête nécessite que le client soit identifié.
403	Forbidden	Le serveur a compris la requête mais l'utilisateur n'est pas autorisé à accéder à cette API.
404	Not Found	La ressource demandée n'existe pas.
500	Internal Server Error	Votre code ne devrait jamais renvoyer cette erreur. Cette erreur devrait être récupérée par votre code et traitée, pour ensuite renvoyer une réponse adéquate au client.

Prenons l'exemple de la création d'un utilisateur, la requête devrait être un POST sur l'URL /users

VERB	URL	Code de retour	Description
POST	/users	201 Created	Création d'un utilisateur

6.13.27. Bonne Pratique – Utiliser des sous-ressources pour identifier un élément unique

Si vous souhaitez récupérer les informations relatives à un utilisateurs précis, votre requête devrait utiliser le verbe « GET » et l'URL devrait avoir la forme /users/{userId} où {userId} est l'ID de l'utilisateur spécifique.

VERB	URL	Code de retour	Description
GET	/users/{userId}	200 OK	Récupération des informations spécifiques de l'utilisateur

6.13.28. Bonne Pratique – Utiliser des sous-ressources pour identifier une relation

Si vous souhaitez récupérer la liste des appareils liés à un utilisateur spécifique, votre requête utilisera le verbe « GET » et l'URL devrait avoir la forme `/users/{user-ID}/devices`.

VERB	URL	Code de retour	Description
GET	<code>/users/{userId}/devices</code>	200 OK	Récupération des appareils liés à un utilisateur

De la même façon, si vous voulez lier un appareil à un utilisateur, la requête utilisera le verbe « POST » et l'URL aura la forme `/users/{user-ID}/devices/{device-ID}`. Dans ce cas, `{user-ID}` et `{device-ID}` sont respectivement l'ID de l'utilisateur qui reçoit l'appareil et l'ID de l'appareil qu'il reçoit.

VERB	URL	Code de retour	Description
POST	<code>/users/{userId}/devices/{deviceId}</code>	201 Created	Création du lien entre l'utilisateur et le l'appareil.

6.13.29. Bonne Pratique – Spécifiez le format de vos données

Il est important que le développeur (et l'application qu'il développe) qui consomme vos API connaisse le format de votre réponse. Le header `Content-Type` existe justement pour répondre à ce besoin. Utilisez-le toujours dans les réponses de vos APIs.

N'utilisez le `Content-Type` « `text/plain` » que si la réponse est effectivement du texte.

Le content type le plus utilisé, pour ne pas dire le content type standard des API REST est le JSON « `application/json` ».

Il est également très important de spécifier le « `charset` » que vous utilisez. Dans la majorité des cas, et sauf contre-indication, le `charset` que vous utilisez est le « `UTF-8` ».

Le Header que vous envoyez en réponse aux requêtes clients devrait normalement être :

Content-Type: `application/json; charset=utf-8`

6.13.30. Bonne Pratique – La gestion des temps (date et heure) :

Le sujet peut sembler trivial mais il est loin de l'être.

En reprenant le sujet de la gestion de nos appareils et de nos utilisateurs, imaginons que nous souhaitions savoir depuis quand un utilisateur a à sa disposition un appareil spécifique.

L'appel que nous allons faire ressemblera à :

VERB	URL	Code de retour	Description
------	-----	----------------	-------------

GET	/users/{userId}/devices/{deviceId}	200 OK	Récupération des informations relatives à l'attribution d'un appareil spécifique pris en charge par un utilisateur spécifique.
-----	------------------------------------	--------	--

La complexité vient du fait que l'attribution de l'appareil a peut-être été enregistrée par une personne dans une time zone X pour un utilisateur dans une time zone Y et doit être visualisée par un utilisateur dans une time zone Z. Comment faire en sorte que tout le monde voit le temps qui corresponde à sa time zone ? Une des solutions à ce problème est de fournir la date et l'heure sous format Timestamp UTC (Petit rappel : le timestamp est le nombre de secondes écoulées depuis le 1er janvier 1970 / L'heure UTC (Universal Time Coordinated), en français Temps Universel Coordonné, est l'heure de référence internationale. Elle correspond aussi à l'heure GMT (Greenwich Mean Time) et à l'heure Z (Zulu)).

La réponse à ce call pourrait ressembler à :

```
{
  "attribution": "enable",
  "timeFrom" : 1557080067
}
```

6.13.31. Bonne Pratique – Utiliser HATEOAS (Hypermedia as the Engine of Application State)

Derrière cet acronyme se cache simplement le fait d'utiliser des liens hypertexte dans les réponses de vos APIs pour indiquer au client de vos APIs où il peut aller chercher plus d'informations concernant une ressource.

L'exemple ci-dessous est un exemple de réponse à la requête qui permet de retrouver quels appareils sont liés à quel utilisateur. Dans la réponse ci-dessous on peut voir que l'appareil avec l'ID 123456 est lié à l'utilisateur avec l'ID « abcde ». La réponse ajoute que des informations relatives à l'appareil (« rel » : « self ») peuvent être trouvées à l'URL spécifiée dans le lien HREF.

```
{
  "user-id": "abcde",
  "devices": [
    {
      "device-ID": 123456,
      "links": [
        {"rel": "self", "href": "/devices/123456"},
        {"rel": "list", "href": "/devices"}
      ]
    }
  ]
}
```

6.13.32. Bonne Pratique – Proposer des filtres, du tri, de la pagination pour les collections

L'utilisabilité de votre API pourra être augmentée par l'ajout de fonctionnalités telles que les filtres, le tri ou la pagination. Si l'on compare les filtres, tri ou pagination au langage SQL, les filtres correspondent à la clause « where », le tri correspond à la clause « sort by » et la pagination correspond aux clauses « limit offset ».

En reprenant notre exemple, si vous souhaitez ne récupérer que la liste des utilisateurs masculins, la requête aura la forme suivante :

VERB	URL
GET	/users?sex=m

Pour ajouter de la pagination, si vous ne souhaitez récupérer que les 10 premiers utilisateurs, la requête aura alors la forme suivante :

VERB	URL
GET	/users?sex=m&limit=10

« limit » donne le nombre d'objets retournés à partir du début de la liste.

Pour récupérer les 10 utilisateurs suivants, la requête aura alors la forme suivante :

VERB	URL
GET	/users?sex=m&offset=10&limit=10

“offset” donne le point de départ du curseur à partir duquel on va prendre en compte les objets.

Et enfin, si vous souhaitez trier la réponse, la requête aura alors la forme suivante :

VERB	URL
GET	/users?sex=m&offset=10&limit=10&sort=+lastName,-birthDate

Dans ce cas-ci, le filtre se fera sur le nom de famille (en ordre alphabétique grâce au « + » indiqué devant « lastName ») et ensuite le filtre se fera sur la date de naissance (en ordre décroissant grâce au « - » indiqué devant le « birthDate »).

6.13.33. Bonne Pratique – Documentez votre API

Si vous souhaitez que votre API soit facilement utilisable et que les développeurs qui auront à la consommer ne vous fagocitent pas votre temps pour comprendre comment l'utiliser, il est INDISPENSABLE de fournir une documentation claire et précise.

Vous aurez donc à documenter :

- les différents chemins à utiliser et leur construction

- Pour chaque chemin, les différentes méthodes (VERB) disponibles
- Pour chaque méthode :
 - Les paramètres acceptés et leur type
 - le format accepté
 - le modèle du body s'il est utilisé
 - Les headers acceptés
 - La/les méthodes d'authentification

Au-delà des descriptions techniques, vous devriez également indiquer aux développeurs votre politique d'utilisation de votre API. Avez-vous un quota d'utilisation (de call) à respecter/à ne pas dépasser, une « Fair Use Policy »,... ?

6.13.34. Bonne Pratique – Versionnez votre API

Comme toute solution informatique, votre API devrait connaître des évolutions, et donc des versions. Il est important d'indiquer aux développeurs qui consomment votre API la façon dont ils peuvent spécifier la version de l'API qu'ils appellent.

Une méthode largement utilisée est d'indiquer le numéro de version dans le PATH de l'URL. Cette méthode a l'avantage d'être simple à mettre en œuvre de votre côté mais l'est peut-être un peu moins pour vos clients (les développeurs qui consomment votre API). En effet, s'il souhaite faire certains appels sur une version N, et d'autres appels sur une version N+1, ils vont devoir gérer deux préfixes de chemins différents.

Une autre solution consiste à faire passer le numéro de version des APIs en paramètres, ou ce qui se fait plus largement, en Header à la requête. Un des avantages de cette solution est de permettre d'avoir une version courante qui évolue au fil du temps. En effet, si le développeur n'utilise pas de paramètre ou de header pour spécifier la version, cela implique qu'il utilise de facto la version courante que vous proposez. Cette technique, bien que séduisante, cache un piège non négligeable, vos versions d'API doivent être compatibles entre elles pour que le développeur qui ne fait pas de modification à son application et appelle continuellement la version courante, ne soit pas mis en défaut lorsque vous ferez une update de la version courante. De la version N à la version N+1. Attention, cette solution vous donnera également plus de travail de gestion.

Pour reprendre notre exemple, nous pourrions avoir les solutions suivantes pour récupérer une liste d'utilisateurs :

VERB	URL
GET	/v1/users
GET	/users?version=1
GET	/users

Headers: "version:1"

6.13.35. Bonne Pratique – Décorrellez les attributs des réponses avec les attributs de votre Data Store

D'un point de vue sécurité, il est bon de ne pas exposer directement le nom des champs de vos bases de données (ou des attributs si l'on parle NoSQL) directement dans les réponses de votre API. En effet, si vous exposez en direct le nom des champs ou des attributs dans vos réponses, vous facilitez le travail des pirates qui auraient envie de vous attaquer. Il est donc important d'avoir une table de mapping entre les informations que vous récupérez depuis votre Data Store (SQL, NoSQL,...) et les informations que votre API envoie en réponse.

Pour revenir à notre exemple, si l'API qui reçoit les coordonnées d'une personne doit envoyer son nom, son prénom, sa date de naissance, on peut imaginer que le retour de ce call-là aura en Body un JSON qui devrait avoir ressembler à ce que l'on trouve ci-dessous :

```
{
  "user" :{
    "firstName" : "Fred",
    "lastName" : "MOAL",
    "birthDate" : 123984000
  }
}
```

6.13.36. Bonne Pratique – Pensez aux applications qui vont consommer vos APIs

En construisant les différents chemins de vos APIs, pensez aux développeurs qui vont devoir les consommer et aux performances de votre Backend si trop de requêtes http sont nécessaires pour remonter une seule donnée.

En reprenant notre exemple initial, une requête en GET sur le chemin /users devrait retourner uniquement une liste de userID, qui permettrait ensuite de retrouver les informations des utilisateurs en faisant une requête GET sur le chemin /users/{userId}. Même si cette solution peut sembler pertinente, si une application veut uniquement afficher le prénom de chaque utilisateur, cela implique que cette application fasse un premier call pour retrouver la liste des userId et ensuite un call pour chaque utilisateur afin de retrouver son prénom, vous multipliez donc énormément le nombre de requêtes nécessaires.

Il peut donc parfois être nécessaire de se montrer flexible sur les réponses que votre API reçoit afin de préserver autant les performances et les coûts de votre plateforme que les performances des applications qui consomment vos APIs.

6.13.37. Bonnes pratiques

- Passage de paramètres
 - @PathVariable, @RequestParam, @RequestBody, @RequestHeader
 - Types des paramètres
- Validation des paramètres

- À la main
- Contraintes de validation @Valid
- Exceptions du modèle

Chapitre 7. Webservices : Sérialisation

ET LES OBJETS JAVA ?

JAXB XML(JSON) ET JAVA

7.1. JAXB

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.

7.1.1. Premier exemple

Annotations dans le code Java pour générer un schéma et des documents XML API JAXB intégrée dans Java SE 6+

```
@XmlRootElement(name="marin",
namespace="http://orleans.univ.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 ;
    ...
}
```

7.1.2. Annotations

- `@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.

7.1.3. 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")) ;
}
```

7.1.4. Résultat

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

7.1.5. Lecture de document XML

```
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()) ;
    }
}
```

7.1.6. 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

7.1.7. 1er problème

Changer le type généré en JSON ?

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

Résultat :

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

7.1.8. 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

7.1.9. Résultat Moxy

- 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"
}
```

7.1.10. Annotations JAXB

La sérialisation : Je maîtrise !

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

7.1.11. Annotations principales

De 3 types :

- Placées sur un package
- Placées sur les classes ou énumérations
- Placées dans les classes sur les champs ou getters

7.1.12. @XmlSchema

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

7.1.13. @XmlSchema : déclaration

Exemple :

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

7.1.14. @XmlSchema : résultat

Déclaration générée :

```
<xs:schema
  elementFormDefault="qualified"
  attributeFormDefault="unqualified"
  targetNamespace="http://orleans.univ.fr/cours-jaxb"
  xmlns:orl="http://orleans.miage.fr/cours-jaxb"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  version="1.0" >
```

7.1.15. Sur les classes

@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

7.1.16. Sur les classes

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 ; ...
}
```

7.1.17. 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

7.1.18. Sur les classes

`@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

7.1.19. 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>
```

7.1.20. 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>
```

7.1.21. Sur un champ ou getter

@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

7.1.22. Sur un champ ou getter

@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

7.1.23. 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>();
```


7.1.24. 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>
```

7.1.25. Sur un champ ou un getter

@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

7.1.26. 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>
```

7.1.27. Sur un champ ou un getter

@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

7.1.28. 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 ;
    ...
}
```

7.1.29. Sur un champ ou getter

Résultat pour un marin :

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<marin>
  <salaire>100</salaire>
</marin>
```

7.1.30. 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'un plugin Maven

7.1.31. 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 ;
    ...
}

@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>() ;
    ...
}

```

7.1.32. 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>

```

```

<?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>

<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>

```

7.1.33. Générer des classes

Génération de classes de schémas

La génération d'un jeu de classes utilise un autre utilitaire fourni dans la distribution JAXB : xjc

On peut utiliser un plugin maven Lecture des xsd et génération des classes correspondantes

A utiliser pour faire un client

7.1.34. xjc

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

7.1.35. ObjectFactory : utilisation

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) ;
}
```

7.1.36. Références

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

7.2. Complément Sérialisation

Objets JSON, XML...

7.2.1. Le problème :

Relations toujours BI-directionnelles en Java

```

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;
}

```

7.2.2. S rialisation XML/JSON ?

S rialisation 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"]
->....

```

7.2.3. 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;
}

```

7.2.4. Solution 1 : on aide Jackson

R sultat :

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

7.2.5. Solution 2 : @JsonIdentityInfo

Exemple :

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

7.2.6. Solution 2 : @JsonIdentityInfo

Résultat :

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

7.2.7. 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"  
  }  
}
```

7.2.8. 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 {}  
}
```

7.2.9. 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;
}

```

7.2.10. Solution 4 : @JsonView

Lors de la sérialisation, on choisit la vue :

```

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

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

```

7.2.11. 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);
    }
}

```

Chapitre 8. Webservices : Sécurité

8.1. 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

8.1.1. pré-Requis

Avant de sécuriser votre application, il faut avant sécuriser le contexte !

- Utilisation de httpS ; Transport Layer Security (TLS) nom officiel de HTTPS
- Let's Encrypt <https://letsencrypt.org/> certificats TLS gratuits
- Normalement, de bout en bout (load balancer, ...)
- Lecture de <https://spring.io/guides/topicals/spring-security-architecture/>

8.1.2. Problèmes spécifiques REST

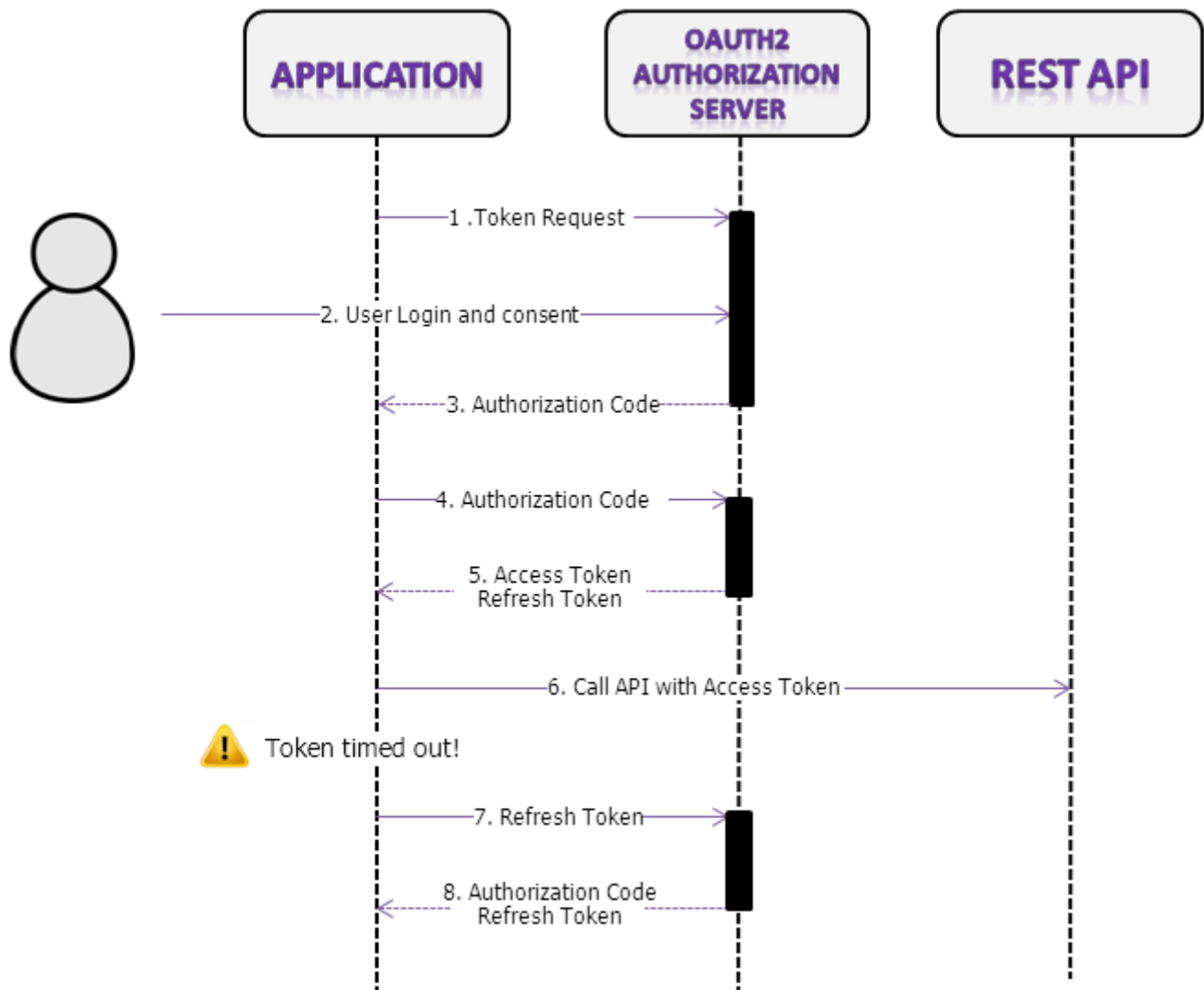
REST ⇒ Http ⇒ Stateless

- 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

8.1.3. OAuth 2.0



8.1.4. Définition d'un protocole

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

8.2. Exemple Spring Sec JWT

Un premier exemple d'utilisation avec des tokens JWT [JOT]

8.2.1. Mapping URL - droits

Principes récurrents : URLs → droits d'accès

Eg avec Spring Security (cf plus bas)

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

8.2.2. Interception URLs

Idée : avec une 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("/**");
    }
}
```

8.2.3. Interception : vérification

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");
    }
}

```

8.2.4. Token

2 techniques pour le Token :

- Génération aléatoire (eg UUID), puis stockage (BD, map...) d'association
- Utilisation encryptage pour signature (e.g. JWT)

En général, passé par Header ("Authorization" ou autre...)

8.2.5. Token : exemple de JWT

Principe : fabriquer un token avec des données, le signer pour garantir sa non modification

- Génération de tokens signés par clé privée : JSON Web Tokens standard RFC 7519
- jwt.io : support multi-langages /multiplateforme
- **Stateless** ⇒ perfs, scalable !

8.2.6. Token JWT

Exemple de token :

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRm91IiwiaWF0Ij0iOnRydWV9.TjVA95OrM7E2cBab30RMhRhDcEfxjoYZgeFONFh7HgQ
```

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

8.2.7. Token JWT

Token JWT : en Java ?

Librairie jjwt, à ajouter à votre pom.xml

```
<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt-api</artifactId>
  <version>0.11.1</version>
</dependency>
<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt-impl</artifactId>
  <version>0.11.1</version>
  <scope>runtime</scope>
</dependency>
<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt-jackson</artifactId> <!-- or jjwt-gson if Gson is preferred -->
  <version>0.11.1</version>
  <scope>runtime</scope>
</dependency>
```

8.2.8. jjwt, encodage

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();
```

8.2.9. jjwt, décodage

Décodage du token :

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

8.2.10. Token JWT : limitations

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

8.3. Spring Security

L'implémentation de référence pour la sécurité en Java/JEE/...

8.3.1. Activation

Avec SpringBoot, activation par un starter security :

```
<dependency>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starter-security</artifactId>  
</dependency>
```

+ éventuellement les starters pour ldap, ...

8.3.2. par Défaut

Authentification de type "Basic" activée par défaut pour toutes les URLs

Au run, log du password du seul utilisateur (user) :

Using generated security password: ea13cfac-5b09-4baa-92c5-df400139a0ea

8.3.3. Configuration

Config par annotations `@Configuration`


```

@Configuration
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .anyRequest().fullyAuthenticated()
            .and()
            .formLogin();
    }
    @Override
    public void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth
            .ldapAuthentication()
            .userDnPatterns("uid={0},ou=people")
            .groupSearchBase("ou=groups")
            .contextSource()
            .url("ldap://localhost:8389/dc=springframework,dc=org")
            .and()
            .passwordCompare()
            .passwordEncoder(new LdapShaPasswordEncoder())
            .passwordAttribute("userPassword");
    }
}

```

8.3.4. Autre exemple avec UserDetails

Redéfinition de `userService` avec `@Bean`

```

@Configuration
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .antMatchers("/", "/home").permitAll()
            .anyRequest().authenticated()
            .and().formLogin().loginPage("/login").permitAll()
            .and().logout().permitAll();
    }
    @Bean
    @Override
    public UserDetailsService userDetailsService() {
        UserDetails user =
            User.withDefaultPasswordEncoder()
                .username("user")
                .password("password")
                .roles("USER")
                .build();
        return new InMemoryUserDetailsManager(user);
    }
}

```

8.3.5. Droits sur les méthodes

Annotation `@Secured` sur les méthodes

```

@SpringBootApplication
@EnableGlobalMethodSecurity(securedEnabled = true)
public class SampleSecureApplication {
}

@Service
public class MyService {
    @Secured("ROLE_USER")
    public String secure() {
        return "Hello Security";
    }
}

```

8.3.6. Droits sur les méthodes : JSR-250

On préférera l'équivalent en JSR-250 (`jsr250Enabled = true`) :

```

@RolesAllowed("ROLE_VIEWER")
public String getUsername2() {
    //...
}
@RolesAllowed({ "ROLE_VIEWER", "ROLE_EDITOR" })
public boolean isValidUsername2(String username) {
    //...
}

```

Pour des contraintes plus complexes, on utilise `@PreAuthorize` et `@PostAuthorize` avec des expressions (`prePostEnabled = true`) :

```

@PreAuthorize("hasRole('ROLE_VIEWER') or hasRole('ROLE_EDITOR')")
public boolean isValidUsername3(String username) {
    //...
}
@PreAuthorize("#username == authentication.principal.username")
public String getMyRoles(String username) {
    //...
}

```

8.3.7. Contexte d'authentification

Une fois authentifié, Spring Security peut injecter un contexte d'authentification dans vos méthodes

```

@RequestMapping("/foo")
public String foo(@AuthenticationPrincipal User user) {
    ... // do stuff with user
}
// ou directement le `Principal`
@RequestMapping("/foo")
public String foo(Principal principal) {
    Authentication authentication = (Authentication) principal;
    User = (User) authentication.getPrincipal();
    ... // do stuff with user
}

```

8.3.8. Filter Tunning

Possibilité d'utiliser ses propres filtres, eg Token JWT !

```

@EnableWebSecurity
public class SecurityConfigurer extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.cors().and().csrf().disable().authorizeRequests()
            .antMatchers(HttpMethod.GET, "/public/*").permitAll()
            .anyRequest().authenticated()
            .and()
            .addFilter(new JwtAuthenticationFilter(authenticationManager()))
            .addFilter(new JwtAuthorizationFilter(authenticationManager()))
            // this disables session creation on Spring Security

        .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);
    }
}

```

```

public class JwtAuthenticationFilter extends UsernamePasswordAuthenticationFilter {

    public JwtAuthenticationFilter(AuthenticationManager authenticationManager) {
        setAuthenticationManager(authenticationManager);
    }

    @Override
    protected void successfulAuthentication(HttpServletRequest req,
                                            HttpServletResponse res,
                                            FilterChain chain,
                                            Authentication auth) throws IOException,
ServletException {
        CustomUserDetails customUserDetails = (CustomUserDetails)auth.getPrincipal();
        // username & password authenticated
        String username = obtainUsername(req);
        String password = obtainPassword(req);
        // record authorities
        Claims claims = Jwts.claims().setSubject(customUserDetails.getUsername());
        claims.put("scopes", auth.getAuthorities().stream().map(s ->
s.toString()).collect(Collectors.toList()));
        claims.put("nom", customUserDetails.getNom());
        claims.put("prenom", customUserDetails.getPrenom());
        claims.put("email", customUserDetails.getEmail());
        // build JWT Token
        String token = Jwts.builder()
            .setClaims(claims)
            .setSubject(customUserDetails.getUsername())
            .setExpiration(new Date(System.currentTimeMillis() + EXPIRATION_TIME))
            .signWith(SignatureAlgorithm.HS512, SECRET.getBytes())
            .compact();
        res.addHeader(HEADER_STRING, TOKEN_PREFIX + token);
    }
}

```

```

public class JwtAuthorizationFilter extends BasicAuthenticationFilter {
    private static final Logger logger =
        LoggerFactory.getLogger(JwtAuthorizationFilter.class);

    public JwtAuthorizationFilter(AuthenticationManager authManager) {
        super(authManager);
    }

    @Override
    protected void doFilterInternal(HttpServletRequest req,
                                    HttpServletResponse res,
                                    FilterChain chain) throws IOException,
        ServletException {
        String header = req.getHeader(HEADER_STRING);
        if (header == null || !header.startsWith(TOKEN_PREFIX)) {
            chain.doFilter(req, res);
            return;
        }
        UsernamePasswordAuthenticationToken authentication = getAuthentication(req);
        SecurityContextHolder.getContext().setAuthentication(authentication);
        chain.doFilter(req, res);
    }

    private UsernamePasswordAuthenticationToken getAuthentication(HttpServletRequest request) {
        String token = request.getHeader(HEADER_STRING);
        if (token != null) {
            // decode token
            Jws<Claims> jwsClaims = Jwts.parser()
                .setSigningKey(SECRET.getBytes())
                .parseClaimsJws(token.replace(TOKEN_PREFIX, ""));
            String user = jwsClaims.getBody().getSubject();
            if (user != null) {
                List<String> scopes = jwsClaims.getBody().get("scopes", List.class);
                List<GrantedAuthority> authorities = scopes.stream()
                    .map(authority ->
                        AuthoritiesRoles.getRoleFromRoleName(authority).getAuthority())
                    .collect(Collectors.toList());
                String nom = jwsClaims.getBody().get("nom", String.class);
                String prenom = jwsClaims.getBody().get("prenom", String.class);
                String email = jwsClaims.getBody().get("email", String.class);
                UsernamePasswordAuthenticationToken userToken = new
                UsernamePasswordAuthenticationToken(user, authorities);
                CustomUserDetails details = new
                CustomUserDetails((UserDetails)userToken.getDetails(), nom, prenom, email);
                userToken.setDetails(details);

                return userToken;
            }
            return null;
        }
    }
}

```

```
        return null;
    }
}
```

8.3.9. Custom Authent : mode classique

Méthode de base pour la définition d'une custom Authent : utiliser l'AuthenticationManagerBuilder

```
@Configuration
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    public void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.
    }
}
```

8.3.10. Custom Auth : AuthenticationProvider

Autre possibilité : créer un bean AuthenticationProvider custom.



This is only used if the AuthenticationManagerBuilder has not been populated

```
@Bean
public MyAuthenticationProvider myAuthenticationProvider() {
    return new MyAuthenticationProvider();
}
...
class MyAuthenticationProvider implements AuthenticationProvider {}
```

8.3.11. Custom Auth : UserDetailsService

Autre possibilité : créer un bean UserDetailsService custom.



This is only used if the AuthenticationManagerBuilder has not been populated and no AuthenticationProviderBean is defined.

```
@Bean
public SpringDataUserDetailsService springDataUserDetailsService() {
    return new SpringDataUserDetailsService();
}

class SpringDataUserDetailsService implements UserDetailsService {
}
```

8.3.12. du Multi http config

Deux configs http différentes : une pour les URLs /api/** (eg REST) et l'autre pour le reste (eg web)

```
@EnableWebSecurity
public class MultiHttpSecurityConfig {
    @Bean
    public UserDetailsService userDetailsService() throws Exception {
        // ensure the passwords are encoded properly
        UserBuilder users = User.withDefaultPasswordEncoder();
        InMemoryUserDetailsManager manager = new InMemoryUserDetailsManager();

        manager.createUser(users.username("user").password("password").roles("USER").build());

        manager.createUser(users.username("admin").password("password").roles("USER", "ADMIN").build());

        return manager;
    }

    @Configuration
    @Order(1)
    public static class ApiWebSecurityConfigurationAdapter extends
WebSecurityConfigurerAdapter {
        protected void configure(HttpSecurity http) throws Exception {
            http
                .antMatcher("/api/**")
                .authorizeRequests()
                    .anyRequest().hasRole("ADMIN")
                    .and()
                .httpBasic();
        }
    }

    @Configuration
    public static class FormLoginWebSecurityConfigurerAdapter extends
WebSecurityConfigurerAdapter {

        @Override
        protected void configure(HttpSecurity http) throws Exception {
            http
                .authorizeRequests()
                    .anyRequest().authenticated()
                    .and()
                .formLogin();
        }
    }
}
```

8.3.13. OAuth2

Support en tant que client (`@EnableOAuth2Sso`) et/ou serveur (`@EnableAuthorizationServer`)

cf <https://spring.io/guides/tutorials/spring-boot-oauth2/>

et la doc en ligne <https://docs.spring.io/spring-security-oauth2-boot/>

8.3.14. OAuth2 Client

Configuration "simple"

```
<dependency>
  <groupId>org.springframework.security.oauth.boot</groupId>
  <artifactId>spring-security-oauth2-autoconfigure</artifactId>
  <version>2.2.0.BUILD-SNAPSHOT</version>
</dependency>
```

```
@SpringBootApplication
@EnableOAuth2Sso
public class SocialApplication {
    ...
}
```

```
security:
  oauth2:
    client:
      clientId: 233668646673605
      clientSecret: 33b17e044ee6a4fa383f46ec6e28ea1d
      accessTokenUri: https://graph.facebook.com/oauth/access_token
      userAuthorizationUri: https://www.facebook.com/dialog/oauth
      tokenName: oauth_token
      authenticationScheme: query
      clientAuthenticationScheme: form
    resource:
      userInfoUri: https://graph.facebook.com/me
```

8.4. bonnes pratiques Default

8.4.1. Activer CSRF


```
@EnableWebSecurity
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .csrf()
            .csrfTokenRepository(
                CookieCsrfTokenRepository.withHttpOnlyFalse());
    }
}
```

8.4.2. Utiliser un CSP

Headers par défaut de Spring Sec :

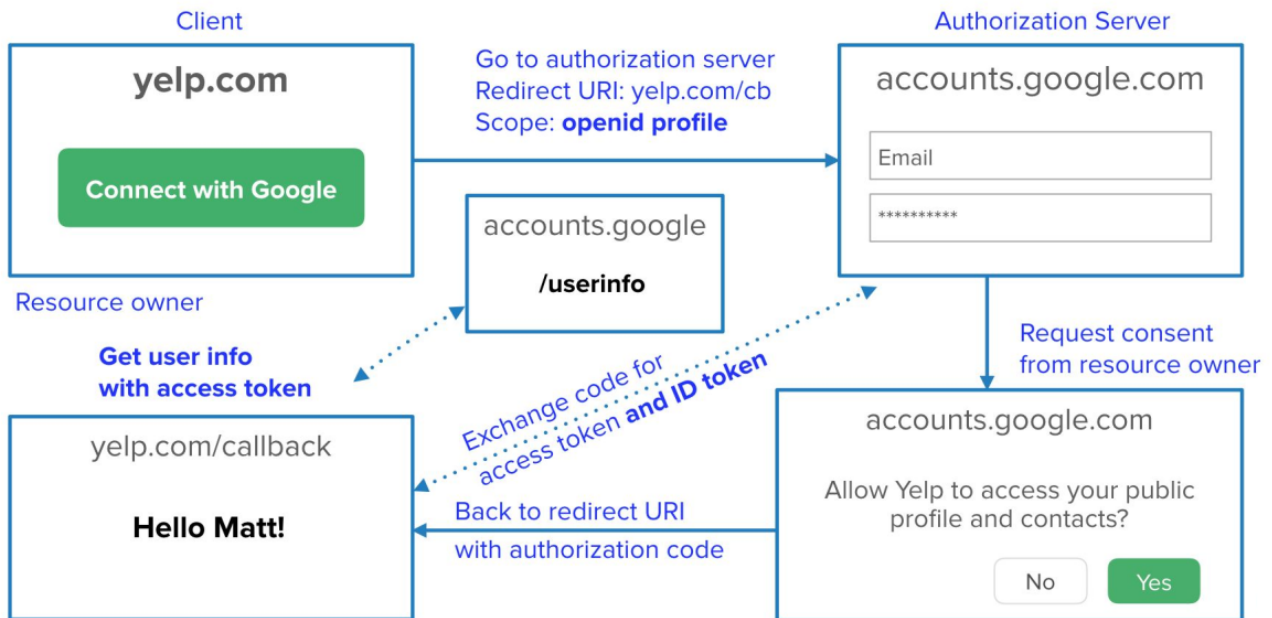
```
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: 0
X-Content-Type-Options: nosniff
Strict-Transport-Security: max-age=31536000; includeSubDomains
X-Frame-Options: DENY
X-XSS-Protection: 1; mode=block
```

```
@EnableWebSecurity
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.headers()
            .contentSecurityPolicy("script-src 'self' " +
                "https://trustedscripts.example.com; " +
                "object-src https://trustedplugins.example.com; " +
                "report-uri /csp-report-endpoint/");
    }
}
```

Pour tester : <https://securityheaders.com>

8.4.3. Utiliser OIDC

Surcouche à OAuth 2 permettant de récupérer un user ID token transformé automatiquement par Spring Sec en Principal injectable !



8.4.4. OIDC config

```
spring:
  security:
    oauth2:
      client:
        registration:
          okta:
            client-id: {clientId}
            client-secret: {clientSecret}
        provider:
          okta:
            issuer-uri: https://{yourOktaDomain}/oauth2/default
```

8.4.5. Définir un PasswordEncoder

```
// config
@Bean
public PasswordEncoder passwordEncoder() {
    return new SCryptPasswordEncoder();
}

// injection
@Autowired
private PasswordEncoder passwordEncoder;

public String hashPassword(String password) {
    return passwordEncoder.encode(password);
}
```

8.4.6. Password

- JAMAIS en String [immutable]
- toujours char[]

8.4.7. Tests

Utiliser ZAP pour tester les vulnérabilités de votre site

<https://github.com/zaproxy/zaproxy>

8.5. Petite conclusions ?

8.5.1. Gestion des états dans REST

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

8.5.2. Sécurité

- 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...)