Les services web avec **Jersey**

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en Programmation par contrainte (IA)
Ingénieur en Génie logiciel

elmouelhi.achref@gmail.com

Plan

Introduction

Service web (WS pour Web Service en anglais), c'est quoi ?

- Un programme (ensemble de fonctionnalités exposées en temps réel et sans intervention humaine)
- Accessible via internet, ou intranet
- Indépendant de tout système d'exploitation
- Indépendant de tout langage de programmation
- Utilisant un système standard d'échange (XML ou JSON), ces messages sont généralement transportés par des protocoles internet connus HTTP (ou autres comme FTP, SMTP...)
- Pouvant communiquer avec d'autres WS

Introduction

Les WS peuvent utiliser les technologies web suivantes :

- `HTTP` (Hypertext Transfer Protocol) : le protocole, connu, utilisé par le World Wide Web et inventé par Roy Fiedling.
- `REST` (Representational State Transfer) : une architecture de services Web, créée aussi par Roy Fielding en 2000 dans sa thèse de doctorat.
- `SOAP` (Simple object Access Protocol) : un protocole, défini par Microsoft et IBM ensuite standarisé par W3C, permettant la transmission de messages entre objets distants (physiquement distribués).
- `WSDL` (Web Services Description Language) : est un langage de description de service web utilisant le format XML (standarisé par le W3C depuis 2007).
- `UDDI` (Universal Description, Discovery and Integration) : un annuaire de WS.

Introduction

RESTful, REST et HTTP, c'est quoi le lien ?

- Le Web et l'API REST sont basés sur le protocole `HTTP` (architecture client/serveur)
- RESTful est juste un adjectif qui désigne une API REST.
- L'API REST utilise donc des méthodes comme `get`, `post`, `delete`... pour l'échange de données entre client et serveur (Un client envoie une requête, et le serveur retourne une réponse)

Introduction

Et Jersey, c'est quoi ? (ʒɛʁzɛ, ʒɛʁzi)

- Framework implémenté en langage Java
- Open-source
- Permettant le développement des WS
- Respectant l'architecture `REST` et les spécifications de `JAX-RS`

Introduction

Et Jersey, c'est quoi ? (~~ville~~, ~~île~~)

- Framework implémenté en langage Java
- Open-source
- Permettant le développement des WS
- Respectant l'architecture `REST` et les spécifications de `JAX-RS`

Et JAX-RS, c'est quoi ?

- `Java API for RESTful Web Services`
- API Java permettant de créer des WS selon l'architecture REST.

Télécharger les .jar pour Jersey

Étapes

- Aller sur `https://jersey.github.io/download.html`
- Cliquer sur `Jersey JAX-RS 2.1 RI bundle` pour télécharger
- Décompresser l'archive et vérifier qu'il contient 3 répertoires contenant chacun un (ou plusieurs) fichier(s) `.jar`
- Sous Eclipse, créer un nouveau projet JEE (`Dynamic Web Project`) et placer tous les `.jar` téléchargés précédemment dans le répertoire `lib` situé dans `WEB-INF`

Premier projet Jersey (sans maven)

Le point d'entrée (composant JAX-RS) : une classe qui étend `Application`

```
package org.eclipse.classes;

import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;

@ApplicationPath("/ws")
public class MyRestWs extends Application{

}
```

`@ApplicationPath` permet de déclarer la base de l'URI pour toutes les ressources

Premier projet Jersey (sans maven)

Notre ressource : une classe Java aussi

```
package org.eclipse.resources;

import javax.ws.rs.Produces;
import javax.ws.rs.GET;
import javax.ws.rs.Path;

@Path("/hello")
public class HelloResource {

        @GET
        @Produces("text/plain")
        public String sayHello() {
                return "Hello Jax-Rs Community";
        }
}
```

Explication

- `@Path` : précise l'URI permettant d'accéder à la ressource
- `@Produces` : indique le type de réponse
- `@GET` : indique la méthode HTTP permettant d'accéder à la resource
- Par convention, le nom de la classe ressource est suffixée par `Resource`

Premier projet Jersey (sans maven)

Pour tester

- Exécuter le projet
- Aller sur `http://localhost:8080/ProjectName/ws/hello`

Premier projet Jersey (avec maven)

Étapes

- Créer un projet Maven `New > Maven Project > Next`
- Dans la liste `Catalog`, sélectionner `All Catalogs` puis chercher `jersey` et Choisir `jersey-quickstart-webapp` (s'il n'existe pas, aller voir le slide suivant)
- Cliquer sur `Next` et remplir les champs `Group Id` (par `org.eclipse` ou `com.example` par exemple) et `Artifact Id` (par `FirstJersey` par exemple)
  ensuite valider

Premier projet Jersey (avec maven)

Si `jersey-quickstart-webapp` n'existe pas

- Cliquer sur `Add Archetype`
- Saisir `org.glassfish.jersey.archetypes` dans `Archetype Group Id`
- Saisir `jersey-quickstart-webapp` dans `Archetype Artifact Id`
- Saisir `2.27` dans `Archetype Version`
- Valider en cliquant sur OK
- Si `jersey-quickstart-webapp` n'apparaît toujours pas, redémarrer Eclipse

Premier projet Jersey (avec maven)

Si le projet est signalé en rouge

- Aller `Project > Properties > Targeted Runtimes`
- Sélectionner un serveur de la liste ou ajouter un nouveau en cliquant sur `New`
- Ensuite valider en cliquant sur `Apply and Close`

Premier projet Jersey (avec maven)

Si le projet est signalé en rouge

- Aller `Project > Properties > Targeted Runtimes`
- Sélectionner un serveur de la liste ou ajouter un nouveau en cliquant sur `New`
- Ensuite valider en cliquant sur `Apply and Close`

Exécuter

- Lancer le serveur puis aller sur `http://localhost:8080/FirstJersey/`

Premier projet Jersey (avec maven)

Accédons maintenant à notre ressource via l'URL précisé dans la classe `MyResource`

```
package org.eclipse.FirstJersey;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

/**
 * Root resource (exposed at "myresource" path)
 */
@Path("myresource")
public class MyResource {

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String getIt() {
        return "Got it!";
    }
}
```

En allant sur lURL `http://localhost:8080/FirstJersey/myresource` une erreur 404

Premier projet Jersey (avec maven)

Allons voir le `web.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee␣http://java.
   sun.com/xml/ns/javaee/web-app_2_5.xsd">
  <servlet>
    <servlet-name>Jersey Web Application</servlet-name>
    <servlet-class>org.glassfish.jersey.servlet.ServletContainer</
       servlet-class>
    <init-param>
      <param-name>jersey.config.server.provider.packages</param-name>
      <param-value>org.eclipse.FirstJersey</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>Jersey Web Application</servlet-name>
    <url-pattern>/webapi/*</url-pattern>
  </servlet-mapping>
</web-app>
```

Premier projet Jersey (avec maven)

Que contient le `web.xml` ?

- L'URL doit donc comporter
  `http://localhost:8080/FirstJersey/webapi/myresource`
- Toutes les ressources (`ClassResource`) doivent être déclarées dans le package `org.eclipse.FirstJersey`
- `<load-on-startup>` indique l'ordre de chargement de la servlet, la servlet avec la plus grande valeur sera chargé en premier.

Premier projet Jersey (avec maven)

Créons une classe `Personne`

```
package org.eclipse.FirstJersey;

import javax.xml.bind.annotation.
   XmlRootElement;

@XmlRootElement
public class Personne {
  private String nom;
  private String prenom;
  private int age;
  public String getNom() {
    return nom;
  }
  public void setNom(String nom) {
    this.nom = nom;
  }
  public String getPrenom() {
    return prenom;
  }
  public void setPrenom(String
     prenom) {
    this.prenom = prenom;
  }
  public int getAge() {
    return age;
  }
  public void setAge(int age) {
    this.age = age;
  }
}
```

L'annotation `@XmlRootElement`

permet de générer un document XML à partir d'une instance de cette classe.

Premier projet Jersey (avec maven)

Créons maintenant une classe `PersonneResource`

```
package org.eclipse.FirstJersey;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

@Path("personnes")
public class PersonneResource {
        @GET
        @Produces(MediaType.APPLICATION_XML)
        public Personne getPersonne(){
                Personne personne = new Personne();
                personne.setNom("Wick");
                personne.setPrenom("John");
                personne.setAge(45);
                return personne;
        }
}
```

Premier projet Jersey (avec maven)

Testons ça

- Lancer le serveur puis aller sur `http://localhost:8080/FirstJersey/webapi/personnes`

Le résultat est :

```
<?xml version="1.0" encoding="UTF-8" standalone="
   true"?>
-<personne>
   <nom>Wick</nom>
   <prenom>John</prenom>
   <age>45</age>
</personne>
```

Si on modifie l'annotation `@XmlRootElement(name="person")`, la balise `personne` sera remplacée par `person` dans le xml.

Premier projet Jersey (avec maven)

L'API JAXB (Java Architecture for XML Binding)

est une API Java qui permet de faire correspondre un document XML à un ensemble de classes et inversement en utilisant des annotations comme (`@XmlRootElement`)

Premier projet Jersey (avec maven)

Autres annotations de l'API JAXB (Java Architecture for XML Binding)

- `@XmlType(propOrder={"nom", "age", "prenom"})` : permet de modifier l'ordre des attributs (ici, on aura le nom, puis l'age et enfin le prenom

```
<?xml version="1.0" encoding="UTF-8" standalone=
   "true"?>
-<personne>
   <nom>Wick</nom>
   <age>45</age>
   <prenom>John</prenom>
</personne>
```

Premier projet Jersey (avec maven)

Autres annotations de l'API JAXB (Java Architecture for XML Binding)

- Annoter le getter `getPrenom()` par `@XmlAttribute(name="firstName")` permet de considérer le prénom comme un attribut de balise dans le xml.

```
<?xml version="1.0" encoding="UTF-8" standalone=
   "true"?>
-<person firstName="John">
  <age>45</age>
  <nom>Wick</nom>
</person>
```

Premier projet Jersey (avec maven)

Découvrons l'annotation `@XmlValue`

```
package org.eclipse.FirstJersey;

import javax.xml.bind.annotation.
   XmlRootElement;

@XmlRootElement(name="person")
public class Personne {
  private String nom;
  private String prenom;
  private int age;
  @XmlAttribute(name="lastName")
  public String getNom() {
    return nom;
  }
  public void setNom(String nom) {
    this.nom = nom;
  }
  @XmlAttribute(name="firstName")
  public String getPrenom() {
    return prenom;
  }
  public void setPrenom(String
     prenom) {
    this.prenom = prenom;
  }
  @XmlValue
  public int getAge() {
    return age;
  }
  public void setAge(int age) {
    this.age = age;
  }
}
```

Premier projet Jersey (avec maven)

Découvrons l'annotation `@XmlValue`

```
package org.eclipse.FirstJersey;

import javax.xml.bind.annotation.
   XmlRootElement;

@XmlRootElement(name="person")
public class Personne {
  private String nom;
  private String prenom;
  private int age;
  @XmlAttribute(name="lastName")
  public String getNom() {
    return nom;
  }
  public void setNom(String nom) {
    this.nom = nom;
  }
  @XmlAttribute(name="firstName")
  public String getPrenom() {
    return prenom;
  }
  public void setPrenom(String
     prenom) {
    this.prenom = prenom;
  }
  @XmlValue
  public int getAge() {
    return age;
  }
  public void setAge(int age) {
    this.age = age;
  }
}
```

Le résultat est : (`@XmlValue` est à utiliser une seule fois par classe)

```
<?xml version="1.0" encoding="UTF-8" standalone="true"?>
<person firstName="John" lastName="Wick">45</person>
```

Premier projet Jersey (avec maven)

Et si on veut retourner plusieurs objets ?

Il faut juste mettre à jour la classe `PersonneResource`, Rien à modifier dans la classe `Personne`

Pour la suite

Supprimer (ou commenter) toutes les annotations de la classe `Personne` et garder seulement `@XmlRootElement`

Pour la suite

Supprimer (ou commenter) toutes les annotations de la classe `Personne` et garder seulement `@XmlRootElement`

```
@XmlRootElement
public class Personne {
  private String nom;
  private String prenom;
  private int age;
  public String getNom() {
    return nom;
  }
  public void setNom(String
      nom) {
    this.nom = nom;
  }
  public String getPrenom()
      {
    return prenom;
  }
  public void setPrenom(
      String prenom) {
    this.prenom = prenom;
  }
  public int getAge() {
    return age;
  }
  public void setAge(int
      age) {
    this.age = age;
  }
}
```

Premier projet Jersey (avec maven)

Modifions la classe `PersonneResource`

```
package org.eclipse.
   FirstJersey;

import java.util.Arrays;
import java.util.List;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.
   MediaType;

@Path("personnes")
public class PersonneResource
   {
  @GET
  @Produces(MediaType.
     APPLICATION_XML)
  public List <Personne>
     getPersonnes()
   {
     Personne personne = new
        Personne();
     personne.setNom("Wick");
     personne.setPrenom("John")
        ;
     personne.setAge(45);
     Personne personne2 = new
        Personne();
     personne2.setNom("Bob");
     personne2.setPrenom("Mike"
        );
     personne2.setAge(35);
     List <Personne> personnes
        = Arrays.asList(personne
        ,personne2);
     return personnes;
   }
}
```

Premier projet Jersey (avec maven)

Le résultat est :

```
<?xml version="1.0" encoding="UTF-8" standalone="
   true"?>
-<personnes>
  -<person>
    <age>45</age>
    <nom>Wick</nom>
    <prenom>John</prenom>
  </person>
  -<person>
    <age>35</age>
    <nom>Bob</nom>
    <prenom>Mike</prenom>
  </person>
</personnes>
```

Premier projet Jersey (avec maven)

Restructurons notre code

- ajoutant un attribut `id` dans la classe `Personne`
- aménageant toute la partie données dans une classe `DAO` ou `Repository` (comme le nomme Spring)

Premier projet Jersey (avec maven)

La classe `PersonneRepository`

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class
   PersonneRepository {
  List <Personne> personnes;
  public PersonneRepository()
     {
    personnes = new ArrayList<
       Personne>();
    Personne personne = new
       Personne();
    personne.setNom("Wick");
    personne.setPrenom("John")
       ;
    personne.setAge(45);
    personne.setId(0);
    Personne personne2 = new
       Personne();
    personne2.setNom("Bob");
    personne2.setPrenom("Mike"
       );
    personne2.setAge(35);
    personne2.setId(1);
    personnes.add(personne);
    personnes.add(personne2);
  }
  public List <Personne>
     getPersonnes(){
    return personnes;
  }
}
```

Premier projet Jersey (avec maven)

Modifions la classe `PersonneResource`

```
import java.util.List;
import javax.validation.ReportAsSingleViolation;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

@Path("personnes")
public class PersonneResource {
  PersonneRepository repo = new PersonneRepository();
  @GET
  @Produces(MediaType.APPLICATION_XML)
  public List <Personne> getPersonnes()  {
    System.out.println("resource personne called");
    return repo.getPersonnes();
  }
}
```

Premier projet Jersey (avec maven)

Modifions la classe `PersonneRepository`

```
public void addPersonne(Personne p) {
  personnes.add(p);
}
```

Et maintenant la classe `PersonneResource`

```
@POST
@Path("personne")
public Personne createPerson(Personne p) {
  System.out.println(p);
  repo.addPersonne(p);
  return p;
}
```

Premier projet Jersey (avec maven)

Utilisation de Postman : simulateur d'un client REST

- Aller sur le navigateur Chrome
- Chercher et installer l'application Postman
- S'authentifier

Il existe plusieurs autres tel que ARC (pour Advanced REST Client)...

Premier projet Jersey (avec maven)

Testons le post de notre web service

- Dans la liste déroulante, choisir POST puis saisir l'url vers notre web service
  http://localhost:8080/FirstJersey/webapi/personnes/personne
- Dans le Headers, saisir Content-Type comme Key et application/xml comme Value
- Ensuite cliquer sur Body, cocher raw, choisir XML (application/xml) et saisir des données sous format xml correspondant à l'objet personne à ajouter

```
<personne>
  <id>7</id>
  <nom>elmou</nom>
  <prenom>ach</prenom>
  <age>32</age>
</personne>
```

- Cliquer sur Send

Premier projet Jersey (avec maven)

Modifions la classe `PersonneRepository`

```
public Personne getPersonne(int id){
  for(Personne personne : personnes) {
    if (personne.getId()==id) {
      return personne;
    }
  }
  return null;
}
```

Premier projet Jersey (avec maven)

Et maintenant la classe `PersonneResource`

```
@GET
@Path("{id}")
@Produces(MediaType.APPLICATION_XML)
public Personne getPersonne(@PathParam("id") int id)
   {
  System.out.println("resource personne called");
  return repo.getPersonne(id);
}
```

Premier projet Jersey (avec maven)

Et maintenant la classe `PersonneResource`

```
@GET
@Path("{id}")
@Produces(MediaType.APPLICATION_XML)
public Personne getPersonne(@PathParam("id") int id)
    {
   System.out.println("resource personne called");
   return repo.getPersonne(id);
}
```

`@Path("id")` : pour indiquer que `id` est un paramètre de la requête

Premier projet Jersey (avec maven)

Et maintenant la classe `PersonneResource`

```
@GET
@Path("{id}")
@Produces(MediaType.APPLICATION_XML)
public Personne getPersonne(@PathParam("id") int id)
    {
   System.out.println("resource personne called");
   return repo.getPersonne(id);
}
```

`@Path("id")` : pour indiquer que `id` est un paramètre de la requête
`@PathParam("id")` : pour récupérer ce paramètre dans l'argument `id`

Premier projet Jersey (avec maven)

Et maintenant la classe `PersonneResource`

```
@GET
@Path("{id}")
@Produces(MediaType.APPLICATION_XML)
public Personne getPersonne(@PathParam("id") int id)
    {
   System.out.println("resource personne called");
   return repo.getPersonne(id);
}
```

`@Path("id")` : pour indiquer que `id` est un paramètre de la requête
`@PathParam("id")` : pour récupérer ce paramètre dans l'argument `id`
Pour tester, on peut aller sur l'url
`http://localhost:8080/FirstJersey/webapi/personnes/1`

Premier projet Jersey (avec maven)

Étapes

- Dans `PersonneResource`, remplacer `@Produces(MediaType.APPLICATION_XML)` par `@Produces(MediaType.APPLICATION_JSON)`
- Dans `pom.xml`, décommenter la section suivante :

```
<!-- uncomment this to get JSON support
<dependency>
  <groupId>org.glassfish.jersey.media</groupId>
  <artifactId>jersey-media-json-binding</
    artifactId>
</dependency>
-->
```

Premier projet Jersey (avec maven)

Pour tester avec `Postman`

- Dans `Headers`, remplacer `application/xml` par `application/json`
- Dans le `Headers`, saisir `Accept` comme `Key` et `application/json` comme `Value`
- Aller sur l'url suivante `http://localhost:8080/FirstJersey/webapi/personnes/1`

Premier projet Jersey (avec maven)

Pour tester avec `Postman`

- Dans `Headers`, remplacer `application/xml` par `application/json`
- Dans le `Headers`, saisir `Accept` comme `Key` et `application/json` comme `Value`
- Aller sur l'url suivante `http://localhost:8080/FirstJersey/webapi/personnes/1`

Le résultat est :

```
{
    "age": 45,
    "id": 1,
    "nom": "Wick",
    "prenom": "John"
}
```

Premier projet Jersey (avec maven)

Pour accepter les deux formats XML et JSON

- Dans `PersonneResource`, remplacer `@Produces(MediaType.APPLICATION_JSON)` par `@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})`
- Maintenant, on peut retourner les deux formats, il suffit de le préciser.

Premier projet Jersey (avec maven)

Et si on veut ajouter des données au format JSON avec `POST`

- Dans `PersonneResource`, ajouter `@Consumes({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})`

Exercice : Utilisation des WS par Angular

Créer un projet Jersey supportant JPA

- Créer un nouveau projet Maven (en Allant dans `File`, ensuite `New` et chercher `Maven Project`)
- Choisir un projet `jersey-quickstart-webapp`
- Remplir les champs `Artifact Id` et `Group Id` puis valider
- Faire un clic droit sur le nom du projet et aller dans `Properties`
- Chercher `Project Facets`
- Cocher la case `JPA` puis cliquer sur `Apply and Close`

Exercice : Utilisation des WS par Angular

Exercice 1

- Ajouter la dépendance suivante (driver MySQL) dans `pom.xml` et enregistrer

```
<!-- https://mvnrepository.com/artifact/mysql/
   mysql-connector-java -->
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>5.1.46</version>
</dependency>
```

- Créer un web service puis le lier à la base de données **formation** et implémenter les méthodes HTTP principales : `GET`, `POST`, `DELETE` et `PUT`.

Exercice : Utilisation des WS par Angular

Exercice 2

Appeler ces méthodes depuis Angular en utilisant les méthodes HTTP dans les services.

Pour pouvoir tester

installer le plugin

- (Allow-Control-Allow-Origin : *) sous chrome
  `https://chrome.google.com/webstore/detail/allow-control-allow-origi/nlfbmbojpeacfghkpbjhddihlkkiljbi`
- (CORS Everywhere) sous firefox
  `https://addons.mozilla.org/en-US/firefox/addon/cors-everywhere/`