Web services SOAP avec Java (JAX-WS)

Hyacinthe MENIET

3 août 2019

Les Web services permettent par exemple à des programmes Java d'appeler des procédures .Net à distance ou d'échanger des messages asynchrones avec ces derniers. Les messages sont généralement du XML et ils transitent via le protocole HTTP. De manière générale les Web services peuvent être considérés comme un moyen du Web sémantique. C'est à dire qu'ils appartiennent aux technologies utilisables via HTTP et qui proposent du contenu compréhensible par des programmes.
Pour votre information, sachez que l'implémentation des Web services dans Java 8 (e.g. Metro) est conforme au WS-I Basic Profile en version 2.0. Dans ce document je vais expliquer comment déployer un Web service SOAP+WSDL sur Tomcat 8. Puis j'indiquerai comment consommer ce Web service depuis un programme Java.

1. Pré-requis

- Vous êtes familier de Java 8 et de sa syntaxe.
- Vous êtes familier des Web services et notamment de SOAP et WSDL.
- Vous disposez du JDK 8 minimum.
- Vous êtes familier de Tomcat et vous avez installé et configuré sa version 8 minimum.
- Vous êtes familier de Maven et disposez de sa version 3.3 minimum

2. Vue d'ensemble

2.1 Présentation rapide des Web services SOAP

SOAP permet de construire des Web services orientés action. C'est-à-dire qu'avec SOAP vous vous concentrez sur les actions que vous pourriez effectuer plutôt que sur les ressources sur lesquelles elles agissent. Un exemple simple d'un service orienté action serait une transaction bancaire dans laquelle un client transfère des fonds d'un compte vers un autre. Dans ce cas de figure, le client ne souhaite pas manipuler directement les ressources (l'argent et les comptes bancaires), il veut simplement passer un ordre et entend que la banque fasse ce qu'il faut pour qu'il soit satisfait. Parce que les Web services SOAP sont orientés action, les services qu'ils proposent (ici les actions) sont fortement liés à l'activité. C'est ainsi qu'un Web service bancaire ne proposera pas les mêmes services qu'un Web service bibliothécaire.
Ci-dessous un exemple de requête SOAP :

```
POST /ws/soap.php HTTP/1.1
Host: www.dotmyself.net
Content-Type: text/xml; charset=utf-8
```

```
Content-Length: 19
SOAPAction: "http://file.dotmyself.net/source/5/soap/HelloWorld"
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Body>
<HelloWorld xmlns="http://www.dotmyself.net/" />
</soap:Body>
</soap:Envelope>
```

C'est la requête envoyée par le client au serveur. Dans cette requête le client invoque la méthode **HelloWorld**. La réponse associée :

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: 14
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Body>
<HelloWorldResponse xmlns="http://www.dotmyself.net/">
<HelloWorldResult>Hello World</HelloWorldResult>
</HelloWorldResponse>
</soap:Body>
</soap:Envelope>
```

En guise de réponse, le serveur lui retourne la message *« Hello World »*

2.2 Présentation de l'article

Dans ce document je propose une vision centrée sur Java des Web services SOAP. C'est-à-dire que je vais expliquer comment faire du SOAP sans se préoccuper ni du XML généré, ni de la sérialisation, ni de la désérialisation des objets Java en XML (et vice versa). Consultez la littérature dédiée à SOAP pour obtenir plus d'informations oreintées SOAP.
Dans cette optique, cet article se base sur 4 éléments indispensables au bon fonctionnement de la chaîne Web service SOAP :

- Un programme Java déployé sur un serveur d'application et exposé comme Web service.
- Un fichier WSDL (*Web Services Description Language*) qui décrit comment communiquer avec le Web service.
- Un protocole d'échange de données XML ici SOAP (*Simple Object Access Protocol*).
- Un programme Java qui va consommer le Web service.

Pour rendre ce document digeste, je vais parcourir les capacités de Metro. Pour rappel, Metro est un morceau de GlassFish. En particulier, c'est Metro qui fourni l'implémentation de référence de JAX-WS. Je vais donc explorer les capacités de JAX-WS (inclus dans Metro) à travers l'exemple

d'un site Internet qui donne pour chaque département français le nombre d'habitants, la superficie et une indication sur le niveau d'urbanisation. Les résultats seront aléatoires ceci pour rester indépendant des sources externes de données.
Vous pouvez télécharger le code source des projets Maven de web services SOAP avec Java (JAX-WS) (i.e. serveur et client).

3. Le Web service côté serveur

Créez un nouveau projet de type webapp avec Maven (commande à taper à la racine de votre workspace) :

```
$ mvn archetype:generate -DgroupId=net.dotmyself.ws -DartifactId=
    department -Dversion=1.0 -Dpackage=net.dotmyself.ws -
    DarchetypeArtifactId=maven-archetype-webapp -DinteractiveMode=
    false
```

Ecrasez le pom.xml généré, dans le dossier department, par ce pom.xml.

3.1 Le bean associé à un département

Créez le Java bean qui contiendra les données relatives à un département (Department.java) :

```
package net.dotmyself.ws;
import javax.xml.bind.annotation.XmlElement;
/**
* Contains the data relating to a Department
*
* @author Hyacinthe MENIET
*/
public class Department {
private int code;
private int population;
private float surface;
private String urbanization;
/**
* @return the code
*/
@XmlElement(name = "code")
public int getCode() {
return code;
}
/**
* @param code the code to set
*/
public void setCode(int code) {
this.code = code;
```

```
}
/**
* @return the population
*/
@XmlElement(name = "population")
public int getPopulation() {
return population;
}
/**
* @param population the population to set
*/
public void setPopulation(int population) {
this.population = population;
}
/**
* @return the surface
*/
@XmlElement(name = "surface")
public float getSurface() {
return surface;
}
/**
* @param surface the surface to set
*/
public void setSurface(float surface) {
this.surface = surface;
}
/**
* @return the urbanization
*/
@XmlElement(name = "urbanization")
public String getUrbanization() {
return urbanization;
}
/**
* @param urbanization the urbanization to set
*/
public void setUrbanization(String urbanization) {
this.urbanization = urbanization;
}
}
```

Chaque assesseur (**getXXX**) est annoté grâce au tag `@XmlElement`. C'est un tag JAXB qui permet d'indiquer que le l'attribut correspondant doit apparaître dans le flux SOAP produit et le nom de la balise XML associée.

3.2 Le Web service

Créez la classe qui sera exposée comme Web service (DepartmentInformation.java) :

```
package net.dotmyself.ws;
import java.util.Random;
import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.WebService;
/**
* Randomly generates useful data about the given Department.
*
* @author Hyacinthe MENIET
*/
@WebService(name="DepartmentService")
public class DepartmentInformation {
private final Random random;
public DepartmentInformation() {
this.random = new Random();
}
/**
* Retrieves random data from the given department's code.
*
* @param departmentCode the department's code.
* @return a {@link Department}
*/
@WebMethod
public @WebResult(name = "department")
Department getDepartment(
@WebParam(name = "departmentcode") int departmentCode) {
// we set the seed so that the sequence can be
// reproduced for the same department
random.setSeed(departmentCode);
// fills the department
String urbanization = "campagnard";
Department department = new Department();
department.setCode(departmentCode);
department.setPopulation(random.nextInt(10000000));
department.setSurface(random.nextFloat() * 10);
if (random.nextBoolean()) {
urbanization = "citadin";
}
department.setUrbanization(urbanization);
return department;
}
}
```

Le code est massivement annoté cette fois par des annotations JAX-WS :

— `@WebService` : Déclare et nomme le Web service.
— `@WebMethod` : Publie la méthode correspondante dans le Web service.
— `@WebResult` : Indique et nomme le résultat de la méthode.
— `@WebParam` : Indique et nomme le paramètre de la méthode.

3.3 L'enveloppe SOAP

Compilez votre projet maven puis exécutez `wsgen` (**livré avec le JDK 8**) pour générer l'enveloppe SOAP. La commande ci-dessous doit être exécutée dans le dossier qui contient les .class de votre projet maven serveur (chez moi c'est **departement/target/classes/**) :

```
> wsgen -cp . net.dotmyself.ws.DepartmentInformation -s ../../src
  /main/java/
```

— L'option `-cp` permet spécifier un classpath
— L'option `-s` permet d'indiquer à wsgen où déposer les sources java.

Dans le dossier `src/main/java/` de votre projet, la commande ci-dessus a généré les classes suivantes :

```
net/dotmyself/ws/jaxws/GetDepartment.java
net/dotmyself/ws/jaxws/GetDepartmentResponse.java
```

3.4 Fichiers JSP et XML

A ce stade vous avez un bean (e.g. `Department`), une classe qui retourne des informations sur les départements français (e.g. `DepartmentInformation`) et un groupe de classes qui permettent d'interroger DepartmentInformation comme un Web service (e.g. `GetDepartment` et `GetDepartmentResponse`).
Vous allez maintenant déployer votre groupe de classes (e.g. `Department, DepartmentInformation, GetDepartment, GetDepartmentResponse`) sur Tomcat, pour cela il vous manque :
index.jsp : La page d'accueil de votre application web.
web.xml : Le descripteur de déploiement de l'application Web :

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
version="3.1"
metadata-complete="true">
<display-name>French departments Web service application</display
  -name>
<description>
Randomly generates useful data about the given French department.
</description>
<listener>
<listener-class>
com.sun.xml.ws.transport.http.servlet.WSServletContextListener
```

```
</listener-class>
</listener>
<servlet>
<servlet-name>jaxservlet</servlet-name>
<servlet-class>com.sun.xml.ws.transport.http.servlet.WSServlet</
    servlet-class>
<load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
<servlet-name>jaxservlet</servlet-name>
<url-pattern>/department</url-pattern>
</servlet-mapping>
</web-app>
```

sun-jaxws.xml : le descripteur de déploiement de JAX-WS :

```
<?xml version="1.0" encoding="UTF-8"?>
<endpoints xmlns="http://java.sun.com/xml/ns/jax-ws/ri/runtime"
    version="2.0">
<endpoint
name="department"
implementation="net.dotmyself.ws.DepartmentInformation"
url-pattern="/department"/>
</endpoints>
```

4. Déploiement du Web service

4.1 Mise à jour de Tomcat 8

La version de JAX-WS livrée avec le JDK 8 ne contient pas toutes les classes nécessaires au fonctionnement d'un Web service dans un conteneur de Servlet comme Tomcat. Il est en revanche suffisant pour consommer un Web service côté client ou pour déployer votre Web service via le serveur web interne au JDK 8.
Pour compléter votre Tomcat 8, téléchargez la dernières version de Metro. Décompressez-la puis placez ses jars **webservices-api-xxx.jar** et **webservices-rt-xxx.jar** dans **$CATALINA_HOME/lib/**

4.2 Préparation et déploiement du war

Générez le war de votre projet maven, l'arborescence devrait être la suivante :

```
index.jsp
WEB-INF/sun-jaxws.xml
WEB-INF/web.xml
WEB-INF/classes/net/dotmyself/ws/Department.class
WEB-INF/classes/net/dotmyself/ws/DepartmentInformation.class
WEB-INF/classes/net/dotmyself/ws/jaxws/GetDepartment.class
```

```
WEB-INF/classes/net/dotmyself/ws/jaxws/GetDepartmentResponse.
    class
```

Quand vous avez terminé poussez simplement votre war dans **$CATALINA_HOME/webapps/** et redémarrez Tomcat. Vous pouvez tester l'application web en vous connectant à l'adresse http://localhost:8080/depar

5. Le client Web service

Il y a trivialement deux méthodes pour consommer un Web service :

- **La méthode synchrone :** la transaction n'a pas d'état et est encadré par une durée limite au bout de laquelle le client stoppe la transaction si elle n'est pas arrivée à son terme. C'est plus rigide mais en général plus économe en code et en mémoire.
- **La méthode asynchrone :** le client dispose d'informations sur l'état d'avancement de l'opération et peut réagir plus finement en cas de difficulté du serveur. C'est plus gourmand mais plus souple.

Dans la suite j'indique comment créer les deux.
Créez un nouveau projet de type jar avec Maven (commande à taper à la racine de votre workspace) :

```
$ mvn archetype:generate -DgroupId=net.dotmyself.wsclient -
    DartifactId=department-client -Dversion=1.0 -Dpackage=net.
    dotmyself.wsclient -DarchetypeArtifactId=maven-archetype-
    quickstart -DinteractiveMode=false
```

Ecrasez le pom.xml généré, dans le dossier department-client, par ce pom.xml.

5.1 Génération des classes dérivées (stubs)

Avant de créer les clients synchrones et asynchrones vous allez générer, grâce à wsimport, les classes dérivées du Web service. Par défaut wsimport ne génère pas les classes qui supportent les appels asynchrones, vous allez l'y obliger à l'aide du fichier bindings.xml :

```
<?xml version="1.0" encoding="UTF-8"?>
<bindings
wsdlLocation="http://localhost:8080/department/department?wsdl"
xmlns="http://java.sun.com/xml/ns/jaxws">
<enableAsyncMapping>true</enableAsyncMapping>
</bindings>
```

Copiez bindings.xml et appelez `wsimport` (**livré avec le JDK 8**) depuis le dossier qui contient les .class de votre projet maven serveur (chez moi c'est **departement/target/classes/**) :

```
> wsimport -s ../../../department-client/src/main/java -b
    bindings.xml http://localhost:8080/department/department?wsdl
```

Cette commande va générer dans votre projet maven client, les stubs en se basant sur les .class de votre projet maven serveur ainsi que le wsdl du web service déployé dans Tomcat.

5.2 Le client synchrone

Le client synchrone est une simple classe Java pourvue d'un main (SynchDepartmentWSClient.java) :

```
package net.dotmyself.wsclient;
import net.dotmyself.ws.Department;
import net.dotmyself.ws.DepartmentInformationService;
import net.dotmyself.ws.DepartmentService;
/**
* Synchronous Client for Department's Web service
*
* @author Hyacinthe MENIET
*/
public class SynchDepartmentWSClient {
public static void main(String[] args) {
if (args == null || args.length < 1) {
throw new IllegalArgumentException("You must indicate a
    department code");
}
int code;
code = Integer.parseInt(args[0]);
// Synchronous Invocation
DepartmentInformationService departInfoService;
departInfoService = new DepartmentInformationService();
DepartmentService departService = departInfoService.
    getDepartmentServicePort();
Department dept = departService.getDepartment(code);
System.out.println("Urbanization="
+ "Population =" + dept.getPopulation() + " habs, "
+ "Surface=" + dept.getSurface() + " km2, " + dept.
    getUrbanization());
}
}
```

Après génération du jar client, le résultat de l'exécution :

```
> java -cp department-client-1.0.jar net.dotmyself.wsclient.
    SynchDepartmentWSClient 38
Urbanization=Population =3200628 habs, Surface=9.785364 km2,
    campagnard
```

5.3 Le client asynchrone

Le client asynchrone reprend le code ci-dessus et le complète par un appel asynchrone (ASynch-DepartmentWSClient.java) :

```
package net.dotmyself.wsclient;
import java.util.concurrent.ExecutionException;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.xml.ws.AsyncHandler;
import javax.xml.ws.Response;
import net.dotmyself.ws.Department;
import net.dotmyself.ws.DepartmentInformationService;
import net.dotmyself.ws.DepartmentService;
import net.dotmyself.ws.GetDepartmentResponse;
/**
* Asynchronous Client for Department's Web service
*
* @author Hyacinthe MENIET
*/
public class AsynchDepartmentWSClient {
public static void main(String[] args) {
try {
if (args == null || args.length < 1) {
throw new IllegalArgumentException("You must indicate a
    department code");
}
int code = Integer.parseInt(args[0]);
DepartmentInformationService departInfoService = new
    DepartmentInformationService();
DepartmentService departService = departInfoService.
    getDepartmentServicePort();
departService.getDepartmentAsync(code, new AsyncHandler<
    GetDepartmentResponse>() {
@Override
public void handleResponse(Response<GetDepartmentResponse> res) {
// Asynchronous Invocation
if (!res.isCancelled() && res.isDone()) {
try {
GetDepartmentResponse message = res.get();
Department dept = message.getDepartment();
System.out.println("Population =" + dept.getPopulation() + " habs
    , "
+ "Surface=" + dept.getSurface() + " km2, "
+ "Urbanization=" + dept.getUrbanization());
} catch (InterruptedException | ExecutionException ex) {
Logger.getLogger(AsynchDepartmentWSClient.class.getName()).log(
    Level.SEVERE, null, ex);
}
}
}
});
// give 10 secondes to asynchronous call to complete
```

```
Thread.sleep(10000);
} catch (InterruptedException ex) {
Logger.getLogger(AsynchDepartmentWSClient.class.getName()).log(
    Level.SEVERE, null, ex);
}
}
}
```

Après génération du jar client, le résultat de l'exécution :

```
> java -cp department-client-1.0.jar net.dotmyself.wsclient.
    AsynchDepartmentWSClient 38
Population =3200628 habs, Surface=9.785364 km2, Urbanization=
    campagnard
```