TP 1 : Création de service web SOAP

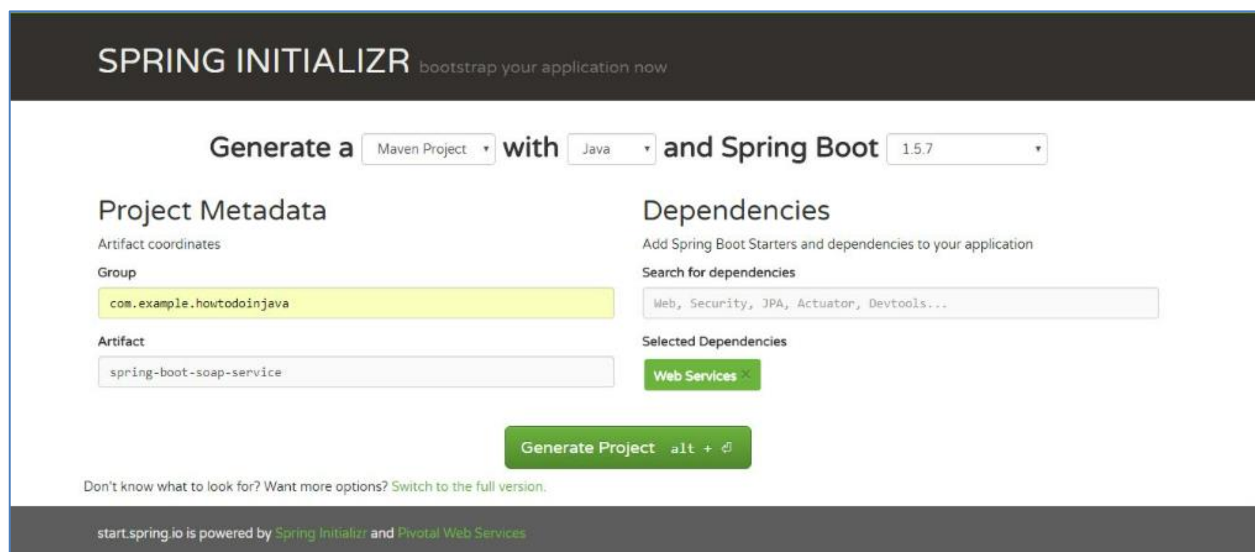
Objectifs : Dans ce TP, vous initiez à la création d'un service web SOAP. Pour ce faire, le TP comporte 2 parties :

1. Création d'un service web SOAP en java, le déployer ensuite le tester.
2. Créer un service web client qui pourra interagir avec le premier WS que vous avez créée.

Environnement de travail : IDE eclipse, Netbeans ou intellige, JDK 7 ou 8

Création d'un service web SOAP en java

1. Créer un nouveau projet Spring boot : Créez un projet Spring à partir du site SPRING INITIALIZR tout en ajoutant les dépendances SOAP. Après avoir sélectionné les dépendances et donné les coordonnées Maven appropriées, téléchargez le projet au format compressé. Décompressez puis importez le projet dans eclipse en tant que projet maven.



2. Ajouter la dépendance Wsdl4j : Editez le fichier **Pom.xml** et ajoutez la dépendance comme suit :

```xml
<dependency>
  <groupId>wsdl4j</groupId>
  <artifactId>wsdl4j</artifactId>
</dependency>
```

3. Créer un modèle de domaine SOAP et générer du code Java : Comme nous suivons l'approche du contrat d'abord pour développer le service, nous devons d'abord créer le domaine (méthodes et paramètres) pour notre service. Pour plus de simplicité, nous avons conservé à la fois la demande et la réponse dans le même XSD, mais dans le cas d'utilisation réel d'entreprise, nous aurons plusieurs XSD qui s'importeront pour former la définition finale.

Student.xsd

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:tns="http://www.exemple.org/student"
targetNamespace="http://www.exemple.org/student" elementFormDefault="qualified">

    <xs:element name="StudentDetailsRequest">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="name" type="xs:string"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <xs:element name="StudentDetailsResponse">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="Student" type="tns:Student"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <xs:complexType name="Student">
        <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="standard" type="xs:int"/>
            <xs:element name="address" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>

</xs:schema>
```

4. Ajouter le plug-in JAXB maven pour XSD afin d'avoir une génération automatique d'objets Java : Nous utiliserons **jaxb2-maven-plugin** pour générer efficacement les classes de domaine. Nous devons maintenant ajouter le plug-in Maven ci-dessous à la section plugin du fichier **pom.xml** du projet.

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>jaxb2-maven-plugin</artifactId>
  <version>1.6</version>
  <executions>
    <execution>
      <id>xjc</id>
      <goals>
        <goal>xjc</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <schemaDirectory>${project.basedir}/src/main/resources/</schemaDirectory>
    <outputDirectory>${project.basedir}/src/main/java</outputDirectory>
    <clearOutputDir>false</clearOutputDir>
  </configuration>
</plugin>
```

- ➩ ***Le plugin utilise l'outil XJC comme moteur de génération de code. XJC compile un fichier de schéma XML en classes Java entièrement annotées.***

- ➩ ***Maintenant, exécutez le plugin maven ci-dessus pour générer du code Java à partir de XSD***

5. Créer un Endpoint SOAP : La classe StudentEndpoint gérera toutes les requêtes entrantes pour le service et déléguera l'appel à la méthode finder du datarepository.

```
package com.example.howtodoinjava.springbootsoapservice;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.ws.server.endpoint.annotation.Endpoint;
import org.springframework.ws.server.endpoint.annotation.PayloadRoot;
import org.springframework.ws.server.endpoint.annotation.RequestPayload;
import org.springframework.ws.server.endpoint.annotation.ResponsePayload;
import com.howtodoinjava.xml.school.StudentDetailsRequest;
import com.howtodoinjava.xml.school.StudentDetailsResponse;

@Endpoint
public class StudentEndpoint
{
  private static final String NAMESPACE_URI = "http://www.exemple.org/student";

  private StudentRepository StudentRepository;

  @Autowired
  public StudentEndpoint(StudentRepository StudentRepository) {
    this.StudentRepository = StudentRepository;
  }

  @PayloadRoot(namespace = NAMESPACE_URI, localPart = "StudentDetailsRequest")
  @ResponsePayload
  public StudentDetailsResponse getStudent(@RequestPayload StudentDetailsRequest request) {
    StudentDetailsResponse response = new StudentDetailsResponse();
    response.setStudent(StudentRepository.findStudent(request.getName()));

    return response;
  }
}
```

Voici quelques détails sur les annotations :

- @Endpoint enregistre la classe auprès de Spring WS en tant que candidat potentiel pour le traitement des messages SOAP entrants.
- @PayloadRoot est ensuite utilisé par Spring WS pour choisir la méthode handler en fonction du namespace et du localPart.
- @RequestPayload indique que le message entrant sera mappé au paramètre de requête de la méthode.
- L'annotation @ResponsePayload permet à Spring WS de mapper la valeur renvoyée au paramètres de la réponse.

6. **Créer un référentiel de données :** Comme mentionné, nous utiliserons les données codées en dur comme backend pour ce TP, ajoutons une classe appelée StudentRepository.java avec l'annotation Spring @Repository. Il contiendra simplement des données dans HashMap et donnera également une méthode de recherche appelée findStudent().

```
package com.example.howtodoinjava.springbootsoapservice;

import java.util.HashMap;
import java.util.Map;
import javax.annotation.PostConstruct;
import org.springframework.stereotype.Component;
import org.springframework.util.Assert;
import com.howtodoinjava.xml.school.Student;

@Component
public class StudentRepository {
  private static final Map<String, Student> students = new HashMap<>();

  @PostConstruct
  public void initData() {

    Student student = new Student();
    student.setName("Sajal");
    student.setStandard(5);
    student.setAddress("Pune");
    students.put(student.getName(), student);

    student = new Student();
    student.setName("Kajal");
    student.setStandard(5);
    student.setAddress("Chicago");
    students.put(student.getName(), student);

    student = new Student();
    student.setName("Lokesh");
    student.setStandard(6);
    student.setAddress("Delhi");
    students.put(student.getName(), student);

    student = new Student();
    student.setName("Sukesh");
    student.setStandard(7);
    student.setAddress("Noida");
    students.put(student.getName(), student);
  }

  public Student findStudent(String name) {
    Assert.notNull(name, "The Student's name must not be null");
    return students.get(name);
  }
}
```

7. **Ajouter un bean de configuration du WS SOAP :** Créez une classe avec l'annotation @Configuration pour contenir les définitions de bean :

```java
import org.springframework.boot.web.servlet.ServletRegistrationBean;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.io.ClassPathResource;
import org.springframework.ws.config.annotation.EnableWs;
import org.springframework.ws.config.annotation.WsConfigurerAdapter;
import org.springframework.ws.transport.http.MessageDispatcherServlet;
import org.springframework.ws.wsdl.wsdl11.DefaultWsdl11Definition;
import org.springframework.xml.xsd.SimpleXsdSchema;
import org.springframework.xml.xsd.XsdSchema;

@EnableWs
@Configuration
public class Config extends WsConfigurerAdapter
{
  @Bean
  public ServletRegistrationBean messageDispatcherServlet(ApplicationContext applicationContext)
  {
    MessageDispatcherServlet servlet = new MessageDispatcherServlet();
    servlet.setApplicationContext(applicationContext);
    servlet.setTransformWsdlLocations(true);
    return new ServletRegistrationBean(servlet, "/service/*");
  }

  @Bean(name = "studentDetailsWsdl")
  public DefaultWsdl11Definition defaultWsdl11Definition(XsdSchema countriesSchema)
  {
    DefaultWsdl11Definition wsdl11Definition = new DefaultWsdl11Definition();
    wsdl11Definition.setPortTypeName("StudentDetailsPort");
    wsdl11Definition.setLocationUri("/service/student-details");
    wsdl11Definition.setTargetNamespace("http://www.exemple.org/student");
    wsdl11Definition.setSchema(countriesSchema);
    return wsdl11Definition;
  }

  @Bean
  public XsdSchema countriesSchema()
  {
    return new SimpleXsdSchema(new ClassPathResource("school.xsd"));
  }
}
```

- MessageDispatcherServlet - Spring-WS l'utilise pour gérer les requêtes SOAP. Nous devons injecter ApplicationContext dans cette servlet pour que Spring-WS puisse reconnaitre les autres beans. Il déclare également le mappage d'URL pour les requêtes.
- DefaultWsdl11Definition expose un WSDL 1.1 standard utilisant XsdSchema. Le nom du bean studentDetailsWsdl sera le nom wsdl qui sera exposé. Il sera disponible sous http://localhost:8080/service/studentDetailsWsdl.wsdl.

8. Test et manipulation du WS SOAP :
    - Faites maven build à l'aide de « **mvn clean install** » et démarrez l'application à l'aide de la commande « **java -jar target\spring-boot-soap-service-0.0.1-SNAPSHOT.jar.** ». Cela fera apparaître un serveur Tomcat dans le port par défaut 8080 et l'application y sera déployée.
    - Allez maintenant sur http://localhost:8080/service/studentDetailsWsdl.wsdl pour voir si le WSDL.
    - Une fois que nous avons généré le WSDL, nous pouvons utiliser ce WSDL pour tester l'application avec SOAPUI ou Postman.