3.11.3 Accéder au service exposé

Relu

Relecture RB

Sommaire :

- Introduction
- Partage de l'interface JAVA
- Génération de code java via maven
- Le pom.xml
  - Le plugin
  - Les dépendances
- Le code métier
- Testons...

Introduction

Deux cas se présentent généralement. L'un où l'on maîtrise le client et le serveur. L'autre où le serveur est développé par un tiers.

Dans le premier cas on a accès au code JAVA de l'interface définissant la couche métier mais aussi au code des objets manipulés. On aura alors intérêt à partager l'interface JAVA entre le client et le serveur.

Dans le deuxième cas, le Web Service n'est pas forcément développé en JAVA et on n'a pas accès au code des objets métiers. On aura alors intérêt à générer automatiquement du code java client via un plugin CXF pour maven.

Partage de l'interface JAVA

Dans ce cas, l'application doit avoir accès aux mêmes modules maven (ou aux jar correspondants) **domain-service** et **domain-bean** de l'application serveur exposant le Web Service. L'interface étant dans le module **domain-service** et les potentiels objets métiers manipulés dans le module **domain-bean**.

Afin d'accéder à la couche service d'un Web Service il y a quelques éléments de configuration CXF à mettre dans la configuration Spring de la couche métier (**properties/domain/domain.xml**) :

```
<import resource="classpath:META-INF/cxf/cxf.xml" />
        <import resource="classpath:META-INF/cxf/cxf-extension-soap.xml" />
        <import resource="classpath:META-INF/cxf/cxf-servlet.xml" />

        <jaxws:client id="domainService"
                serviceClass="org.esupportail.example.domain.DomainService"
                address="http://localhost:8085/cxf/DomainService" />
```

Explications :

- Les imports spécifiques à CXF. A noter l'utilisation du préfixe **classpath:** qui permet de rechercher les fichier XML dans les jar du projet (celui de CXF ici)
- L'espace de nom **jaxws:** est défini dans la balise racine du fichier *Spring* :

  ```
  xmlns:jaxws="http://cxf.apache.org/jaxws"
  xsi:schemaLocation=".../...
   http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd"
  ```

- La balise **jaxws:client** définit un bean, au sens *Spring*, qui va donner accès aux services du Web Service comme si ces services étaient locaux à l'application cliente. Elle a comme propriétés :
  - **serviceClass** qui pointe vers l'interface qui est implémentée par le Web service distant.
  - **address** correspond à l'url qui permettra d'interagir avec le Web Service.

Génération de code java via maven

La méthode qui est présentée ici consiste à :

- Générer automatiquement du code java d'accès au service distant via un plugin CXF pour maven (Ici on utilise, à titre d'exemple, un service de création d'entrée dans le WorkFlow de ORI-OAI cf. ori-oai.org)

- Utiliser ce code java générée dans la couche métier de l'application cliente

Au final on accède au service avec très peu de code. Tout le travaille est fait par le plugin CXF pour maven qui va complètement masquer la complexité du code JAX-WS en le générant automatiquement.

Le pom.xml

Le plugin

Pour que le plugin CXF pour maven génère automatiquement le code, il faut ajouter ces lignes dans le pom.xml :

```
<plugin>
 <groupId>org.apache.cxf</groupId>
 <artifactId>cxf-codegen-plugin</artifactId>
 <version>2.4.3</version>
 <executions>
  <execution>
   <id>generate-sources</id>
   <phase>generate-sources</phase>
   <configuration>
    <wsdlOptions>
     <wsdlOption>
      <wsdl>http://srv.univ.fr/workflow/xfire/OriWorkflowService?wsdl</wsdl>
     </wsdlOption>
    </wsdlOptions>
   </configuration>
   <goals>
    <goal>wsdl2java</goal>
   </goals>
  </execution>
 </executions>
</plugin>
```

configuration/wsdlOptions/wsdlOption/wsdl est le WSDL qui va être utilisé par la génération du code.

⚠ Ce WSDL sera redéfini, par la configuration de l'application, lors de l'exécution. En effet, on ne peut pas préjuger, lors de l'écriture du code, de la localisation du Web Service à utiliser.

⚠ Le code sera automatiquement généré lors d'appels maven classiques comme **mvn compile** ou autres. Il sera aussi automatiquement ajouté au classpath Eclipse via la commande **mvn eclipse:eclipse**

Les dépendances

Pour pouvoir utiliser CXF dans un développement, le plus simple est d'ajouter la dépendance suivante :

```
<dependency>
 <groupId>org.esupportail</groupId>
 <artifactId>esup-commons2-ws-cxf</artifactId>
 <version>${esup-commons.version}</version>
</dependency>
```

⚠ Le fait d'utiliser cette dépendance *esup-commons* permet de rationaliser les différents développements en garantissant l'utilisation, pour une version d'*esup-commons* donnée, des mêmes librairies.

Avec la version 0.2 de *esup-commons* on a constaté qu'il fallait aussi préciser d'utiliser cette dépense explicitement dans le pom.xml de l'application :

```
<dependency>
 <groupId>xerces</groupId>
 <artifactId>xercesImpl</artifactId>
 <version>2.8.1</version>
</dependency>
```

Cette dépendance sera ajoutée dans les futures versions de *esup-commons2-ws-cxf* afin d'éviter au développeur de s'en préoccuper et de perdre du temps à comprendre pourquoi son développement ne fonctionne pas.

Le code métier

On prend ici l'exemple d'un appel au Web Service du moteur de WorkFlow de ORI-OAI.

```
public Long createWorkflowInstance(String XML, String uid) {
 IOriWorkflowService oriWorkflowService;
 try {
  oriWorkflowService = new IOriWorkflowService(new URL(wsdl));
 }
 catch (MalformedURLException e) {
  throw new RuntimeException(e);
 }
 IOriWorkflowServicePortType client = oriWorkflowService.getIOriWorkflowServiceHttpPort();
 return client.newWorkflowInstance(XML, metadataTypeId, uid);
}
```

IOriWorkflowService est un objet automatiquement généré par le plugin CXF pour maven.

Il est ensuite instancié en prenant en paramètre l'URL du WSDL du Web Service à utiliser. Typiquement ici, il s'agit d'un attribut de la classe domainServiceImpl qui est injecté par Spring et correspond à un paramétrage de l'exploitant de l'application.

Ensuite, on crée un objet de type IOriWorkflowServicePortType (lui aussi généré automatiquement) qui est le point d'entrée vers les méthodes exposées par le Web Service.

On a ainsi directement accès aux méthodes métier du WorkFlow ORI comme *newWorkflowInstance*.

Testons...

Là aussi c'est simple !

Dans **src/test/java**, on ajoute une classe de test :

```
@ContextConfiguration(locations="/properties/applicationContext.xml")
@RunWith(SpringJUnit4ClassRunner.class)
public class OriWorkflowWsTest {
.../...
```

Les annotations permettent ici d'automatiquement charger le contexte Spring lors de l'exécution du test.

On injecte ensuite le **domainService** par annotation :

```
@Autowired
DomainService domainService;
```

Voici le code de test :

```
@Test
public void testOriWorkflowWs() {
 InputStream inputStream = getClass().getResourceAsStream("/tef-simple-auteur-test.xml");
 String XML = getString(inputStream);
 domainService.createWorkflowInstance(XML, "bourges");
}
```

⚠ **tef-simple-auteur-test.xml** est un fichier XML d'exemple présent dans le dossier **src/test/resources** du projet