J2EE
TP 3

BRIZAI Olivier
THORAVAL Maxime

1 Utilisation de Spring

Dans la première partie du TP, nous avons appris à utiliser les bases de Spring. Ce dernier va nous permettre de dissocier, du code Java, les classes de service et les DAO. Ceci aura pour effet une plus grande facilité à changer de méthode récupération des données.
Par exemple : Nous pourrons décider de passer de l'utilisation d'une base de données à celle de fichiers juste en modifiant un fichier XML (en supposant que les classes nécessaires aient été créées au préalable).

1.1 DAO

Avant d'utiliser Spring, la récupération d'une classe DAO se faisait de cette manière :

```
1         DAOImpl dao = new DAOImpl();
2         dao.init();
```

Ici, nous avons instancié un objet de type *DAOImpl* et l'avons initialisé.
Le principal problème est lié au fait que nous utilisons une classe définie. Si l'on décide de changer le nom du DAO , il faudra modifier le code en conséquence. L'utilisation de Spring va ainsi eviter plusieurs recompilations en cas de changement du DAO. La couche web et le couche service ne seront en effet pas à recompiler.

Voici le nouveau code lorsque l'on utilise Spring :

```
1         IDAO dao =(IDAO) (new XmlBeanFactory(new
             ClassPathResource("spring-config.xml"))).
             getBean("dao");
```

L'identifiant "dao" fait ici appel au bean définie dans le fichier spring-config.xml :

```
1 <bean id="dao" class="ensicaen.tb.mvc.eleves.dao.DAOImpl"
      destroy-method="destroy" init-method="init">
2 </bean>
```

Le XML va permettre d'instancier le bean, tandis que c'est le java qui l'utilisera. On remarque que l'on ignore dans le code java de quelle implémentation du IDAO

il s'agit. Ce choix est fait dans le XML et permet de diminuer les dépendances entre les couches.

On vient de mettre en place Spring pour la couche DAO et on fait de même pour la couche service grâce à un bean "service qui représentera une instance de notre classe de service.

Dans le fichier Application.java, on appellera alors l'instance de service ainsi :

```
1 service =(IService) (new XmlBeanFactory(new
     ClassPathResource("spring-config.xml"))).getBean("
     service");
```

On remarque que l'on fait appelle à un IService et non plus à une ServiceImpl, ce qui limite les dépendances (comme pour le DAO). Notre implémentation d'IService est également défninie dans le fichier spring-config.xml :

```
1 <!-- La classe Service -->
2         <bean id="service" class="ensicaen.tb.mvc.eleves.
             service.IServiceImpl">
3               <property name="dao" ref="dao" />
4         </bean>
```

On constate qu'elle a comme attribut un "dao", qui fait bien entendu référence au "dao" définit dans le même fichier en XML. Ainsi lorsque l'on instancie la classe qui gère les service, la classe du DAO sera automatiquement intanciée grâce au XML.

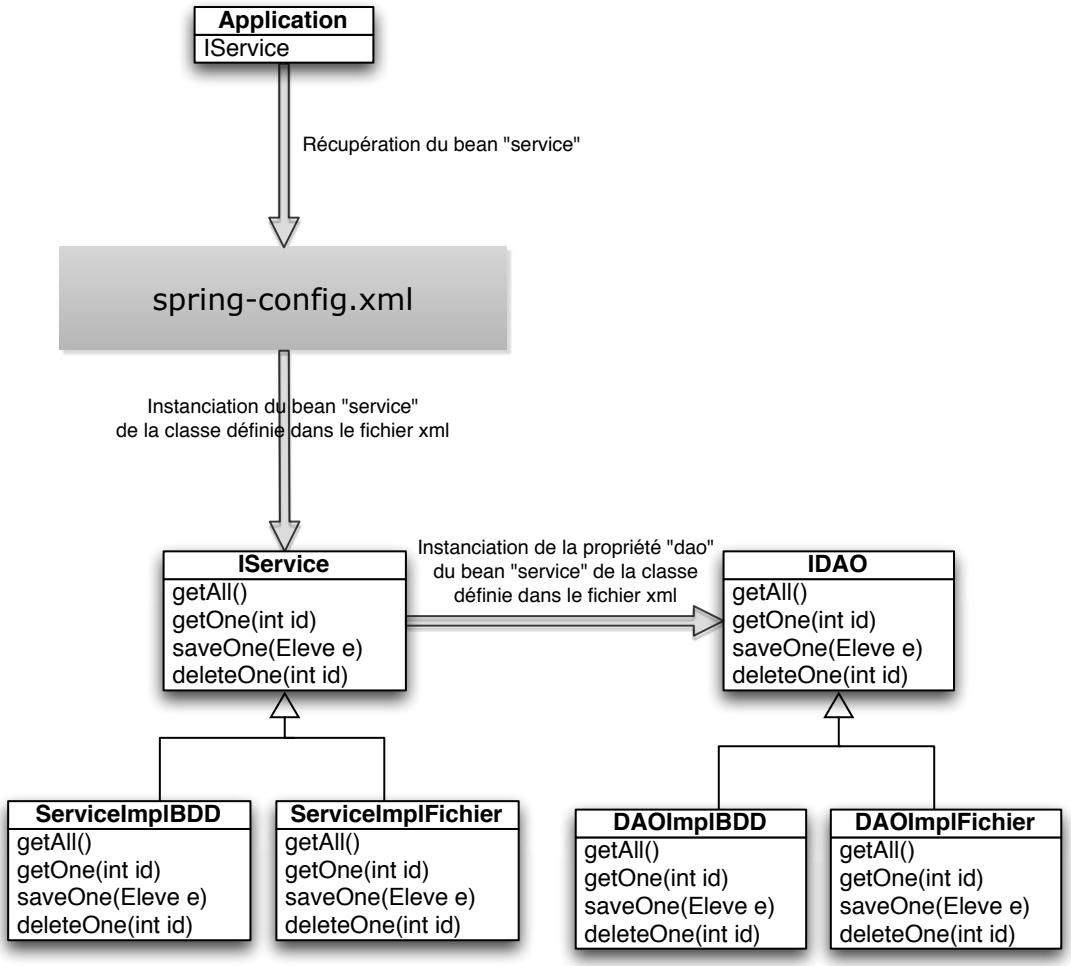

FIGURE 1 – Architecture d'utilisation de spring

2 Utilisation d'iBatis

iBatis est une nouvelle couche qui vient s'insérer entre le DAO et la couche d'accès aux données (JDBC). Dans le fichier spring-config.xml, on va définir de nouveaux Bean. Ces beans vont nous permettre de déporter tous les appels à des fonctionnalitées de JDBC du code Java vers le XML.

Si l'on part du coté de la couche d'accès aux données et qu'on remonte vers le DAO, voici les étapes a effectuer dans l'ordre :

Etape 1 : La connection au SGBD est faite dans le bean *dataSource* :

```
1 <!-- la source de donn es utilisant jdbc -->
2         <bean id="dataSource"
3                 class="org.springframework.jdbc.
                    datasource.SingleConnectionDataSource"
4                 destroy-method="destroy">
5                 <property name="driverClassName" value="
                    org.postgresql.Driver" />
6                 <property name="url" value="jdbc:
                    postgresql://postgres/clinique" />
7                 <property name="username" value="thoraval
                    " />
8                 <property name="password" value="canari"
                    />
9         </bean>
```

Etape 2 : On va maintenant définir le fichier XML contenant les informations sur l'architecture de la base de données :

```
1 <!-- SqlMapClient -->
2         <bean id="sqlMapClient" class="org.
             springframework.orm.ibatis.
             SqlMapClientFactoryBean">
3                 <property name="dataSource">
4                         <ref local="dataSource" />
5                 </property>
6                 <property name="configLocation" value="
                    classpath:sql-map-config-postgres.xml"
                     />
7         </bean>
```

On remarque que ce bean est lié au Bean précédent *dataSource*, en effet l'accès aux données requiert une connexion au SGDB.

Etape 3 : L'étape située en amont de l'accès aux données, c'est la définition du bean du DAO. Ce bean est lié au bean *sqlMapClient* précédent à travers lequel il va effectuer ses requêtes sur la base de données.

```
1 <!-- La classe DAO -->
2         <bean id="dao" class="ensicaen.tb.mvc.eleves.dao.
             DAOImplCommon">
3                 <property name="sqlMapClient" ref="
                    sqlMapClient" />
```

```
4           </bean>
```

Comme on le voit ici, on a définit pour l'occasion un nouveau DAO en java : *DAOImplCommon*

```
1 public class DAOImplCommon extends SqlMapClientDaoSupport
      implements IDAO {
```

Il remplace la précédente classe DAOImpl. Il implémente naturellement l'interface IDAO, mais surtout il hérite de SqlMapClientSupport qui est une classe de la bibliothèque iBatis.

Le fichier *sql-map-config-postgres.xml* permet de "mapper" nos classes métiers à des tables de la base de données. *sqlMapClient*.

```
1 <sqlMapConfig>
2         <sqlMap resource="eleve-postgres.xml" />
3 </sqlMapConfig>
```

Dans *sql-map-config-postgres.xml* on a renseigné le fichier *eleve-postgres.xml*. Dans ce fichier est défini le mapping de classe Eleve avec une table associée dans la base de donnée. Ce mapping va permettre à iBatis d'instancier un bean de type Eleve à partir des informations de la base de données, tout cela sans que le développeur n'est à intervenir. De plus, une insertion dans la base d'un Eleve va automatiquement mettre l'objet avec l'id généré. On a ainsi déporté toute la partie JDBC au sein de fichier XML.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <!DOCTYPE sqlMap PUBLIC "-//ibatis.apache.org//DTD SQL
       Map 2.0//EN"
4          "http://ibatis.apache.org/dtd/sql-map-2.dtd">
5
6  <sqlMap>
7          <!--  alias classe [Eleve] -->
8          <typeAlias alias="Eleve.classe" type = "ensicaen.
             tb.mvc.eleves.entities.Eleve"/>
9
10         <!--  mapping table [ELEVES] -objet [Eleve] -->
11         <resultMap id="Eleve.map" class="Eleve.classe">
12                 <result property="id" column="id"/>
13                 <result property="version" column="
                     version"/>
14                 <result property="nom" column="nom"/>
15                 <result property="prenom" column="prenom"
                     />
16                 <result property="dateNaissance" column="
                     datenaissance"/>
17                 <result property="redoublant" column="
                     redoublant"/>
18                 <result property="annee" column="annee"/>
19                 <result property="filiere" column="
                     filiere"/>
```

```
20          </resultMap>
21
22          <!--  liste de tous les eleves -->
23          <select id="Eleve.getAll" resultMap="Eleve.map">
24                  SELECT * FROM ELEVES
25          </select>
26
27          <!--  obtenir un eleve en particulier  -->
28          <select id="Eleve.getOne" parameterClass="int"
              resultMap="Eleve.map">
29                  SELECT * FROM ELEVES WHERE id=#value#
30          </select>
31
32          <select id="Eleve.nbEleve" resultClass="int">
33                  SELECT count(*) FROM ELEVES
34          </select>
35
36          <!-- ajouter un eleve -->
37          <insert id="Eleve.insertionOne" parameterClass="
              Eleve.classe">
38                  <selectKey keyProperty="id">
39                          SELECT nextval('SEQ_ELEVES') as
                              value
40                  </selectKey>
41                  INSERT INTO ELEVES values (#id#, 1, #nom#
                        , #prenom#, #dateNaissance#, #
                      redoublant#, #annee#, #filiere#)
42          </insert>
43
44          <!-- mettre    jour un   lve   -->
45          <update id="Eleve.updateOne" parameterClass="
              Eleve.classe">
46                  UPDATE ELEVES SET version = #version#,
                      nom = #nom#, prenom = #prenom#,
                      dateNaissance = #dateNaissance#,
47                   annee = #annee#, redoublant = #
                        redoublant#, filiere = #filiere#
                        WHERE id = #id#
48          </update>
49
50          <!-- supprimer un   lve   -->
51          <delete id="Eleve.deleteOne" parameterClass="int"
              >
52                  DELETE FROM ELEVES WHERE id = #id#
53          </delete>
54
55  </sqlMap>
```

L'appel aux fonctions de cette classe Eleve, se feront comme précédemment dans le DAO, en l'occurance ici dans la classe *DaoImplCommon*. En revanche l'appel est

ici transparent puisque l'accès à la BDD est géré dans le XML : Voici un exemple pour la suppression d'un élève :

```
1  public void deleteOne(int id) {
2                  try {
3                          getSqlMapClientTemplate().delete(
                              "Eleve.deleteOne",id);
4                  } catch (Exception ex) {
5                          throw new DAOException("Delete
                              impossible \n" + ex.getMessage
                              (), 50);
6                  }
7  }
```

Comme précédemment, pour diminuer le couplage, l'implémentation de la couche service est définie dans le fichier *spring-config.xml* :

```
1  <bean id="service"
2                  class="org.springframework.transaction.
                      interceptor.
                      TransactionProxyFactoryBean">
3                  <property name="transactionManager" ref="
                      transactionManager" />
4                  <property name="target">
5                          <bean class="ensicaen.tb.mvc.
                              eleves.service.IServiceImpl">
6                                  <property name="dao" ref=
                                      "dao" />
7                          </bean>
8                  </property>
9                  <property name="transactionAttributes">
10                         <props>
11                                 <prop key="saveMany">
                                       PROPAGATION_REQUIRED</
                                       prop>
12                                 <prop key="deleteMany">
                                       PROPAGATION_REQUIRED</
                                       prop>
13                         </props>
14                 </property>
15         </bean>
```

Ici aussi l'instanciation de la classe de service provoque l'instanciation de la classe du DAO. (rajouter un comentaire sur les transactions)

Voici un schéma qui résume un peu plus clairement ce qui a été dit :

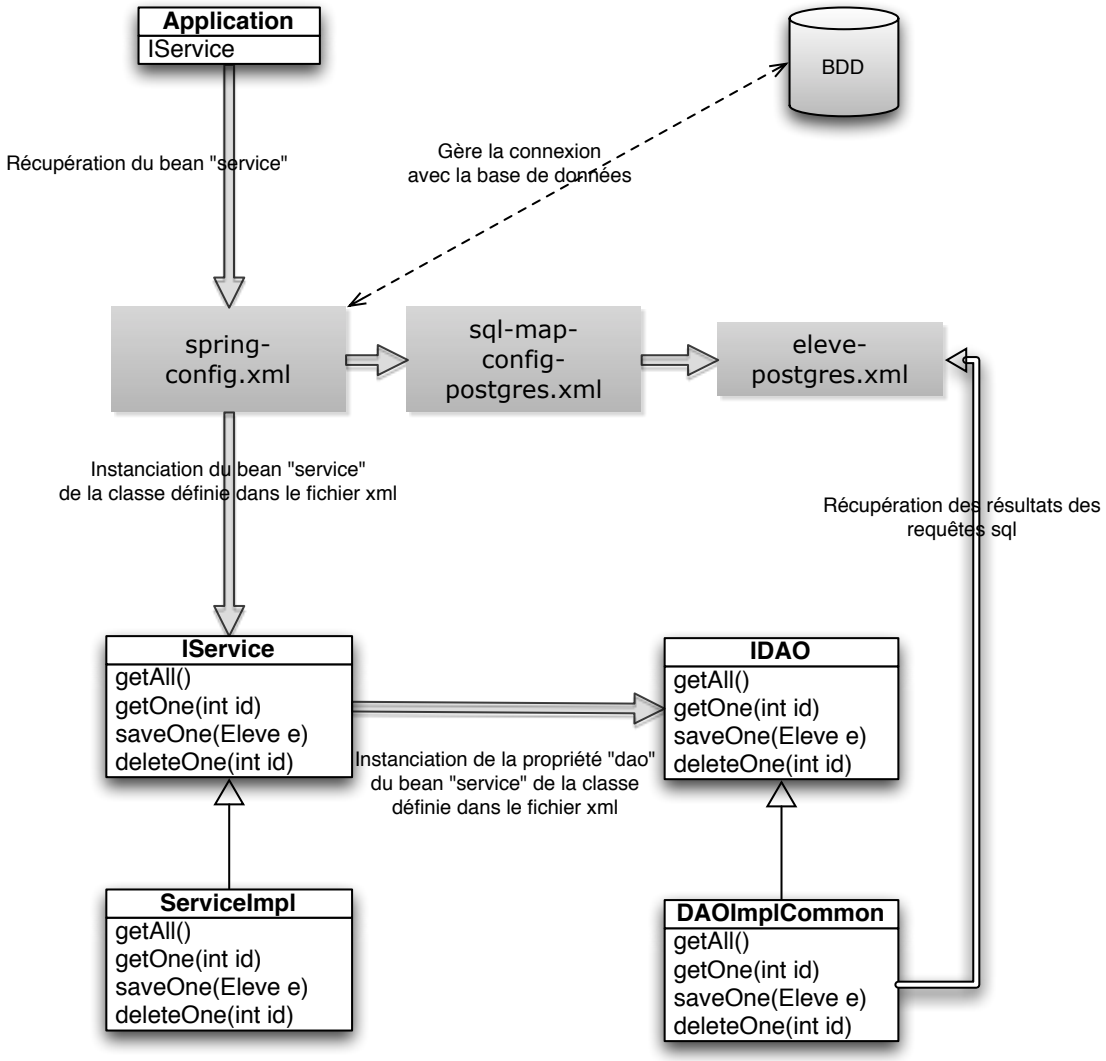

FIGURE 2 – Architecture d'utilisation de spring avec ibatis

3 Utilisation des transaction

Lorsque l'on fait plusieurs ajouts à la suite au sein de la base de données, nous pouvons décider que si un ne fonctionne pas alors les autres ne doivent pas être pris en compte. C'est ce que nous avons fait dans la suite du TP.

Dans un premier temps, nous rajoutons les fonctions "saveMany" et "deleteMany" dans la couche service. Chacune d'entre elle faisant appel à des fonctions déjà existantes au sein des DAO.

```
1 void saveMany(Eleve[] eleves);
2 void deleteMany(int[] ids);
```

Nous indiquons maintenant au sein du fichier de configuration de Spring que pour ces deux fonctions, nous souhaitons qu'il y ait ouverture d'une transaction. Pour cela, créons un bean *transactionManager* que va se charger de gérer les transactions.

```
1 <!-- D finition du gestionnaire de transaction -->
2 <bean id="transactionManager"
3         class="org.springframework.jdbc.datasource.
             DataSourceTransactionManager">
4         <property name="dataSource" ref="dataSource" />
5 </bean>
```

Nous pouvons remarquer que ce dernier va utiliser la base de données (propriété dataSource présente). En effet, les transactions sont des mécanismes implémentés au sein de la base. Lorsqu'une transaction est ouverte, il y aura un ROLLBACK de toutes le modifications si une requête échoue.

Dans un deuxième temps, nous devons modifier la définition du bean "service" pour qu'il soit de type *TransactionProxyFactoryBean*. Nous le lions à *transactionManager* et y définissons comme propriétés la classe d'implémentation de la couche service (*IServiceImpl*). Nous indiquons enfin les méthodes qui déclencheront une transaction (*saveMany, deleteMany*). Pour chacune, nous indiquons une condition quand à l'ouverture d'une transaction (toujours une nouvelle ; si une présente, on l'utilise ; ...). Ici, nous avons mis *PROPAGATION_REQUIRED* qui indique qu'elles doivent être utilisé dans une transaction, si une existe, elle est utilisée, sinon on en crée une.

```
1  <bean id="service"
2          class="org.springframework.transaction.
              interceptor.TransactionProxyFactoryBean">
3          <property name="transactionManager" ref="
              transactionManager" />
4          <property name="target">
5                  <bean class="ensicaen.tb.mvc.eleves.
                      service.IServiceImpl">
6                          <property name="dao" ref="dao" />
7                  </bean>
8          </property>
9          <property name="transactionAttributes">
10                 <props>
11                         <prop key="saveMany">
                               PROPAGATION_REQUIRED</prop>
```

```
12                          <prop key="deleteMany">
                               PROPAGATION_REQUIRED</prop>
13                  </props>
14          </property>
15  </bean>
```

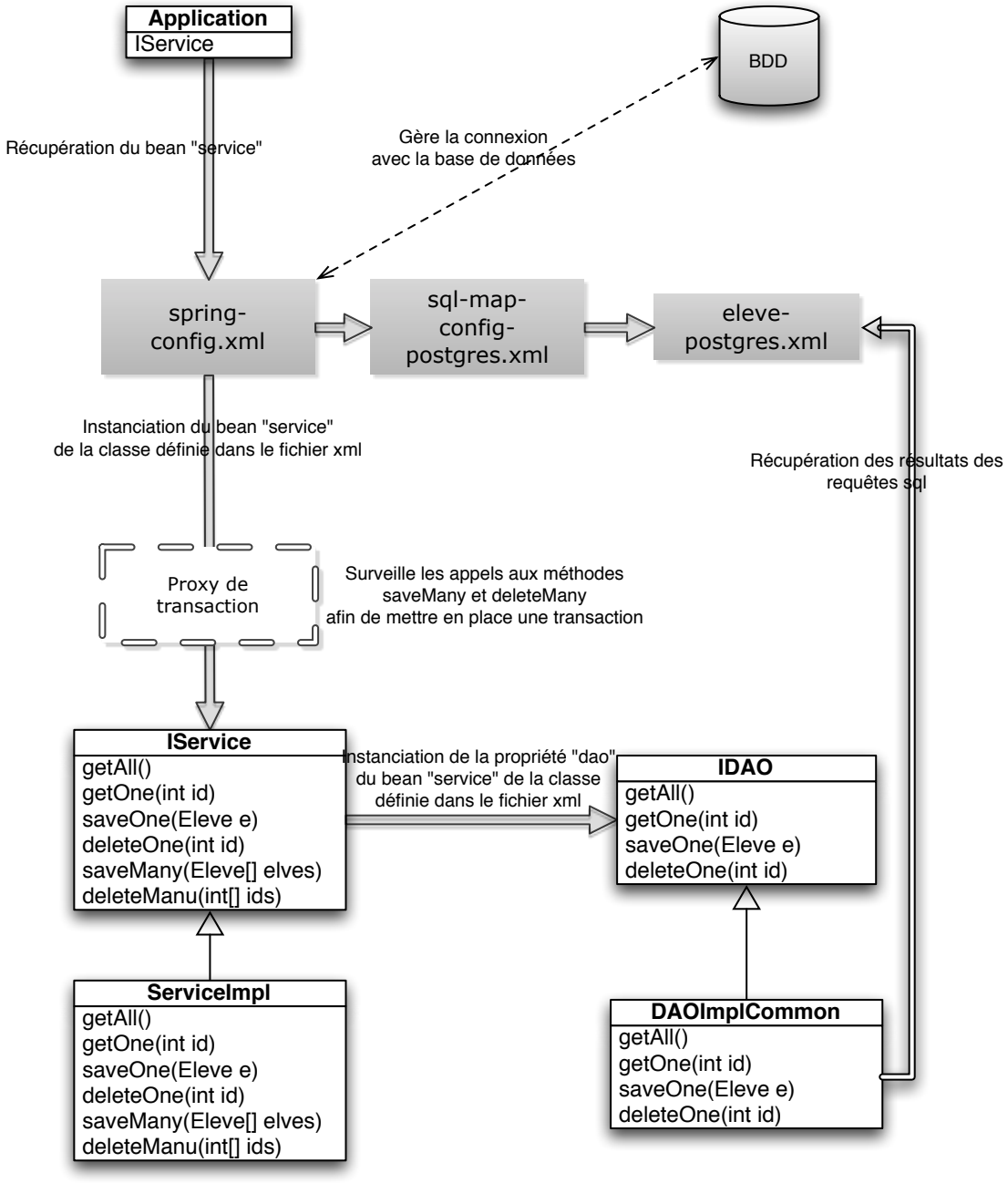

FIGURE 3 – Architecture d'utilisation de spring avec les transactions