Spring MVC par l'exemple
- Partie 5 -

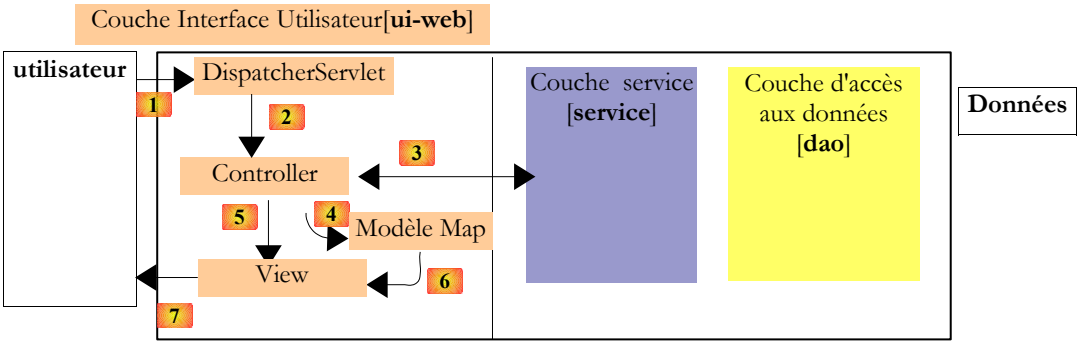
serge.tahe@istia.univ-angers.fr, avril 2006

1 Rappels

Nous poursuivons dans cet article le travail fait dans les précédents :

- Spring MVC par l'exemple – partie 1 : [http://tahe.developpez.com/java/springmvc-part1]
- Spring MVC par l'exemple – partie 2 : [http://tahe.developpez.com/java/springmvc-part2]
- Spring MVC par l'exemple – partie 3 : [http://tahe.developpez.com/java/springmvc-part3]
- Spring MVC par l'exemple – partie 4 : [http://tahe.developpez.com/java/springmvc-part4]

Dans le dernier article, nous avons mis en oeuvre Spring MVC dans une architecture 3tier [web, metier, dao] sur un exemple basique de gestion d'une liste de personnes. Cela nous a permis d'utiliser les concepts qui avaient été présentés dans les précédents articles.



Dans la version 1, la liste des personnes était maintenue en mémoire et disparaissait au déchargement de l'application web. Dans la version étudiée ici, la liste des personnes est maintenue dans une table de base de données. Nous ne développerons que les couches [service] et [dao]. La couche [ui-web] reste celle qui a été développée pour la version 1.

Nous développerons l'exemple tout d'abord avec une base de données Firebird. Puis ensuite, nous présenterons des versions pour les SGBD Postgres, MySQL et SQL Server Express. Le tableau suivant indique au lecteur où se procurer librement ces SGBD et des clients pour les administrer :

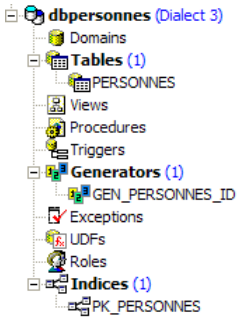
SGBD	site(s)	client(s) d'administration
Firebird	http://www.firebirdsql.org/	IBExpert : http://www.ibexpert.com/ SQL Manager Lite pour Firebird : http://www.sqlmanager.net/fr/products/ibfb/manager
Postgres	http://www.postgresql.org/	SQL Manager Lite pour Postgres : http://www.sqlmanager.net/fr/products/postgresql/manager
MySQL	http://www.mysql.com/ http://www.easyphp.org/	SQL Manager Lite pour MySQL : http://www.sqlmanager.net/fr/products/mysql/manager
SQL Server Express	http://msdn.microsoft.com/vstudio/express/sql/	SQL Server Management Studio Express : http://msdn.microsoft.com/vstudio/express/sql/download/ SQL Manager Lite pour SQL Server Express: http://www.sqlmanager.net/fr/products/mssql/manager

2 Spring MVC dans une architecture 3tier – Exemple 2 - Firebird

2.1 La base de données Firebird

Dans cette version, nous allons installer la liste des personnes dans une table de base de données Firebird. On trouvera dans le document [http://tahe.developpez.com/divers/sql-firebird/] des informations pour installer et gérer ce SGBD. Dans ce qui suit, les copies d'écran proviennent d' IBExpert, un client d'administration des SGBD Interbase et Firebird.

La base de données s'appelle [dbpersonnes.gdb]. Elle contient une table [PERSONNES] :

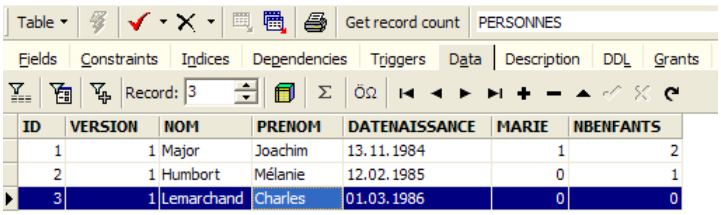


La table [PERSONNES] contiendra la liste des personnes gérée par l'application web. Elle a été construite avec les ordres SQL suivants :

```
1.
2.  CREATE TABLE PERSONNES (
3.      ID              INTEGER NOT NULL,
4.      "VERSION"       INTEGER NOT NULL,
5.      NOM             VARCHAR(30) NOT NULL,
6.      PRENOM          VARCHAR(30) NOT NULL,
7.      DATENAISSANCE   DATE NOT NULL,
8.      MARIE           SMALLINT NOT NULL,
9.      NBENFANTS       SMALLINT NOT NULL
10. );
11.
12.
13. ALTER TABLE PERSONNES ADD CONSTRAINT CHK_PRENOM_PERSONNES check (PRENOM<>'');
14. ALTER TABLE PERSONNES ADD CONSTRAINT CHK_MARIE_PERSONNES check (MARIE=0 OR MARIE=1);
15. ALTER TABLE PERSONNES ADD CONSTRAINT CHK_NOM_PERSONNES check (NOM<>'');
16. ALTER TABLE PERSONNES ADD CONSTRAINT CHK_ENFANTS_PERSONNES check (NBENFANTS>=0);
17.
18.
19. ALTER TABLE PERSONNES ADD CONSTRAINT PK_PERSONNES PRIMARY KEY (ID);
```

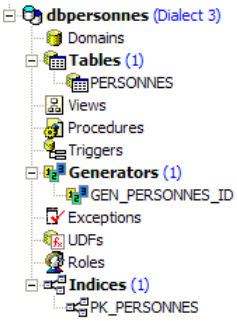
- lignes 2-10 : la structure de la table [PERSONNES], destinée à sauvegarder des objets de type [Personne], reflète la structure de cet objet. Le type booléen n'existant pas dans Firebird, le champ [MARIE] (ligne 8) a été déclaré de type [SMALLINT], un entier. Sa valeur sera 0 (pas marié) ou 1 (marié).
- lignes 13-16 : des contraintes d'intégrité qui reflètent celles du validateur de données [ValidatePersonne].
- ligne 19 : le champ ID est clé primaire de la table [PERSONNES]

La table [PERSONNES] pourrait avoir le contenu suivant :

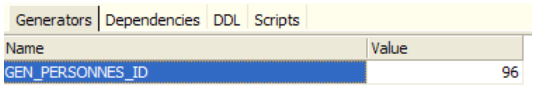


ID	VERSION	NOM	PRENOM	DATENAISSANCE	MARIE	NBENFANTS
1	1	Major	Joachim	13.11.1984	1	2
2	1	Humbort	Mélanie	12.02.1985	0	1
3	1	Lemarchand	Charles	01.03.1986	0	0

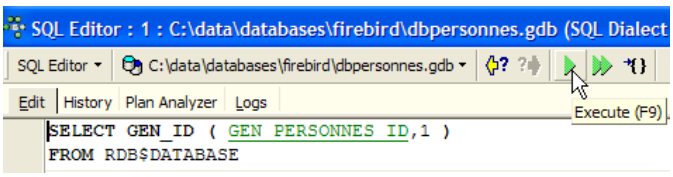
La base [dbpersonnes.gdb] a, outre la table [PERSONNES], un objet appelé générateur et nommé [GEN_PERSONNES_ID]. Ce générateur délivre des nombres entiers successifs que nous utiliserons pour donner sa valeur, à la clé primaire [ID] de la classe [PERSONNES]. Prenons un exemple pour illustrer son fonctionnement :



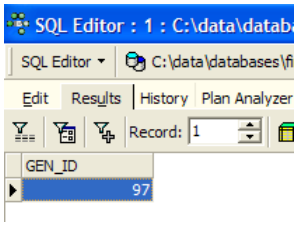
- double-cliquer sur [GEN_PERSONNES_ID]



- le générateur a actuellement la valeur 96

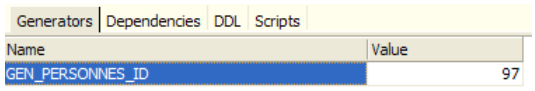


- émettons l'ordre SQL ci-dessus (F12) ->



- la valeur obtenue est l'ancienne valeur du générateur +1

On peut constater que la valeur du générateur [GEN_PERSONNES_ID] a changé (double-clic dessus + F5 pour rafraîchir) :



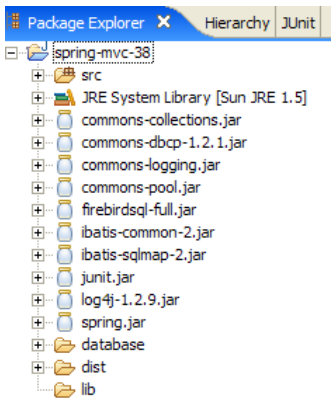
L'ordre SQL

```
SELECT GEN_ID ( GEN_PERSONNES_ID,1 ) FROM RDB$DATABASE
```

permet donc d'avoir la valeur suivante du générateur [GEN_PERSONNES_ID]. GEN_ID est une fonction interne de Firebird et [RDB$DATABASE], une table système de ce SGBD.

2.2 Le projet Eclipse des couches [dao] et [service]

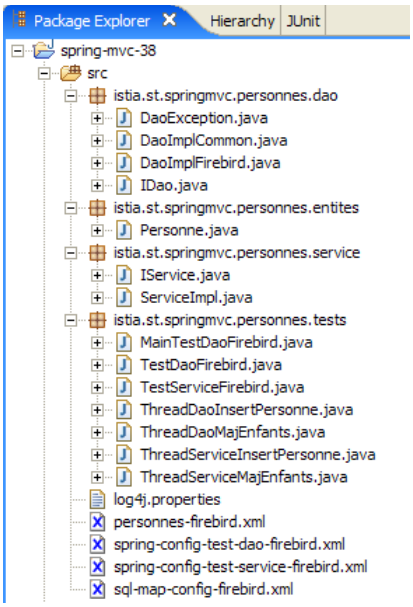
Pour développer les couches [dao] et [service] de notre application avec base de données, nous utiliserons le projet Eclipse [spring-mvc-38] suivant :



Le projet est un simple projet Java, pas un projet web Tomcat. Rappelons que la version 2 de notre application va utiliser la couche [web] de la version 1. Cette couche n'a donc pas à être écrite.

Dossier [src]

Ce dossier contient les codes source des couches [dao] et [service] :



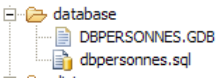
On y trouve différents paquetages :

- [istia.st.springmvc.personnes.dao] : contient la couche [dao]
- [istia.st.springmvc.personnes.entites] : contient la classe [Personne]
- [istia.st.springmvc.personnes.service] : contient la classe [service]
- [istia.st.springmvc.personnes.tests] : contient les tests JUnit des couches [dao] et [service]

ainsi que des fichiers de configuration qui doivent être dans le *ClassPath* de l'application.

Dossier [database]

Ce dossier contient la base de données Firebird des personnes :



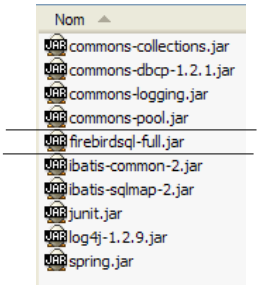
- [dbpersonnes.gdb] est la base de données.
- [dbpersonnes.sql] est le script SQL de génération de la base :

```
1.  /******************************************************************************/
2.  /***          Generated by IBExpert 2006.03.07 27/04/2006 10:27:11          ***/
3.  /******************************************************************************/
4.
5.  SET SQL DIALECT 3;
6.
7.  SET NAMES NONE;
8.
9.  CREATE DATABASE 'C:\data\2005-2006\webjava\dvp-spring-mvc\mvc-38\database\DBPERSONNES.GDB'
10. USER 'SYSDBA' PASSWORD 'masterkey'
11. PAGE_SIZE 16384
12. DEFAULT CHARACTER SET NONE;
13.
14.
15.
16. /******************************************************************************/
17. /***                              Generators                                ***/
18. /******************************************************************************/
19.
20. CREATE GENERATOR GEN_PERSONNES_ID;
21. SET GENERATOR GEN_PERSONNES_ID TO 787;
22.
23.
24.
25. /******************************************************************************/
26. /***                                Tables                                  ***/
27. /******************************************************************************/
28.
29.
```

```
30.
31. CREATE TABLE PERSONNES (
32.     ID              INTEGER NOT NULL,
33.     "VERSION"       INTEGER NOT NULL,
34.     NOM             VARCHAR(30) NOT NULL,
35.     PRENOM          VARCHAR(30) NOT NULL,
36.     DATENAISSANCE   DATE NOT NULL,
37.     MARIE           SMALLINT NOT NULL,
38.     NBENFANTS       SMALLINT NOT NULL
39. );
40.
41. INSERT INTO PERSONNES (ID, "VERSION", NOM, PRENOM, DATENAISSANCE, MARIE, NBENFANTS) VALUES (1, 1,
    'Major', 'Joachim', '1984-11-13', 1, 2);
42. INSERT INTO PERSONNES (ID, "VERSION", NOM, PRENOM, DATENAISSANCE, MARIE, NBENFANTS) VALUES (2, 1,
    'Humbort', 'Mélanie', '1985-02-12', 0, 1);
43. INSERT INTO PERSONNES (ID, "VERSION", NOM, PRENOM, DATENAISSANCE, MARIE, NBENFANTS) VALUES (3, 1,
    'Lemarchand', 'Charles', '1986-03-01', 0, 0);
44.
45. COMMIT WORK;
46.
47.
48.
49. /* Check constraints definition */
50.
51. ALTER TABLE PERSONNES ADD CONSTRAINT CHK_PRENOM_PERSONNES check (PRENOM<>'');
52. ALTER TABLE PERSONNES ADD CONSTRAINT CHK_NOM_PERSONNES check (NOM<>'');
53. ALTER TABLE PERSONNES ADD CONSTRAINT CHK_MARIE_PERSONNES check (MARIE=0 OR MARIE=1);
54. ALTER TABLE PERSONNES ADD CONSTRAINT CHK_ENFANTS_PERSONNES check (NBENFANTS>=0);
55.
56.
57. /******************************************************************************/
58. /***                              Primary Keys                              ***/
59. /******************************************************************************/
60.
61. ALTER TABLE PERSONNES ADD CONSTRAINT PK_PERSONNES PRIMARY KEY (ID);
```

Dossier [lib]

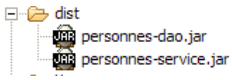
Ce dossier contient les archives nécessaires à l'application :



On notera la présence du pilote JDBC [firebirdsql-full.jar] du SGBD Firebird. Toutes ces archives font partie du *Classpath* du projet Eclipse.

Dossier [dist]

Ce dossier contiendra les archives issues de la compilation des classes de l'application :

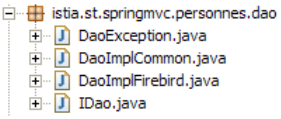


- [personnes-dao.jar] : archive de la couche [dao]
- [personnes-service.jar] : archive de la couche [service]

2.3 La couche [dao]

2.3.1 Les composantes de la couche [dao]

La couche [dao] est constituée des classes et interfaces suivantes :



- [IDao] est l'interface présentée par la couche [dao]
- [DaoImplCommon] est une implémentation de celle-ci où le groupe de personnes se trouve dans une table de base de données. [DaoImplCommon] regroupe des fonctionnalités indépendantes du SGBD.
- [DaoImplFirebird] est une classe dérivée de [DaoImplCommon] pour gérer spécifiquement une base Firebird.
- [DaoException] est le type des exceptions non contrôlées, lancées par la couche [dao]. Cette classe est celle de la version 1.

L'interface [IDao] est la suivante :

```
1.  package istia.st.springmvc.personnes.dao;
2.
3.  import istia.st.springmvc.personnes.entites.Personne;
4.
5.  import java.util.Collection;
6.
7.  public interface IDao {
8.    // liste de toutes les personnes
9.    Collection getAll();
10.   // obtenir une personne particulière
11.   Personne getOne(int id);
12.   // ajouter/modifier une personne
13.   void saveOne(Personne personne);
14.   // supprimer une personne
15.   void deleteOne(int id);
16. }
```

- l'interface a les mêmes quatre méthodes que dans la version précédente.

La classe [DaoImplCommon] implémentant cette interface sera la suivante :

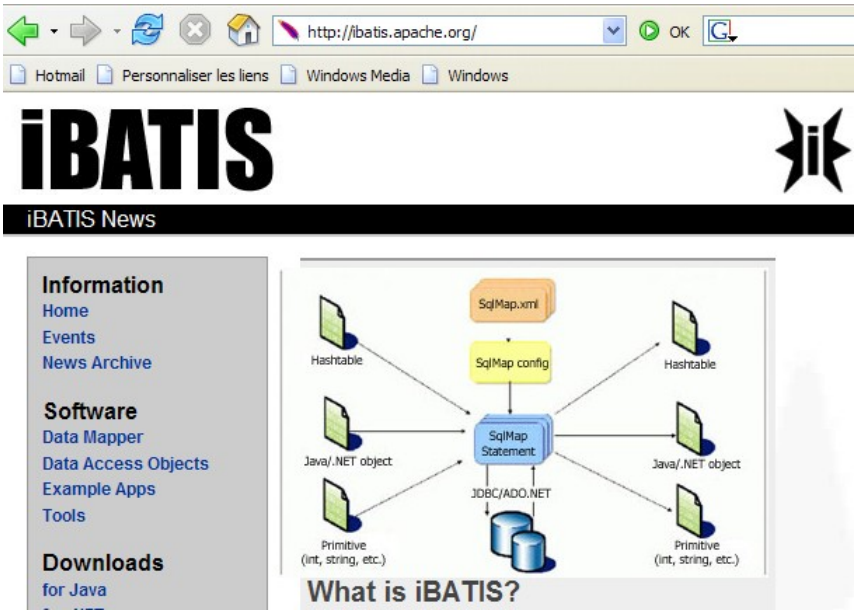
```
1.  package istia.st.springmvc.personnes.dao;
2.
3.  import istia.st.springmvc.personnes.entites.Personne;
4.  import org.springframework.orm.ibatis.support.SqlMapClientDaoSupport;
5.
6.  import java.util.Collection;
7.
8.  public class DaoImplCommon extends SqlMapClientDaoSupport implements
9.      IDao {
10.
11.   // liste des personnes
12.   public Collection getAll() {
13. ...
14.   }
15.
16.   // obtenir une personne en particulier
17.   public Personne getOne(int id) {
18. ...
19.   }
20.
21.   // suppression d'une personne
22.   public void deleteOne(int id) {
23. ...
24.   }
25.
26.   // ajouter ou modifier une personne
27.   public void saveOne(Personne personne) {
28.     // le paramètre personne est-il valide ?
29.     check(personne);
30.     // ajout ou modification ?
31.     if (personne.getId() == -1) {
32.       // ajout
33.       insertPersonne(personne);
34.     } else {
35.       updatePersonne(personne);
36.     }
37.   }
38.
39.   // ajouter une personne
40.   protected void insertPersonne(Personne personne) {
41. ...
42.   }
43.
44.   // modifier une personne
45.   protected void updatePersonne(Personne personne) {
```

```
46. ...
47.  }
48.
49.  // vérification validité d'une personne
50.  private void check(Personne p) {
51. ...
52.  }
53.
54. ...
55. }
```

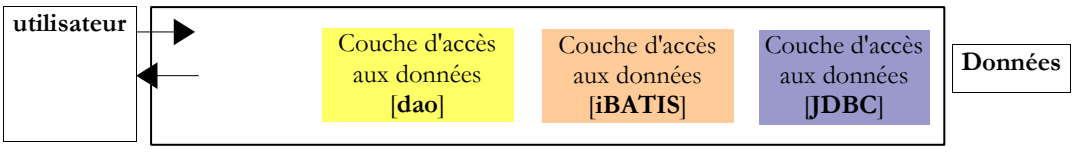
- lignes 8-9 : la classe [DaoImpl] implémente l'interface [IDao] et donc les quatre méthodes [getAll, getOne, saveOne, deleteOne].
- lignes 27-37 : la méthode [saveOne] utilise deux méthodes internes [insertPersonne] et [updatePersonne] selon qu'on doit faire un ajout ou une modification de personne.
- ligne 50 : la méthode privée [check] est celle de la version précédente. Nous ne reviendrons pas dessus.
- ligne 8 : pour implémenter l'interface [IDao], la classe [DaoImpl] dérive de la classe Spring [**SqlMapClientDaoSupport**].

2.3.2 La couche d'accès aux données [iBATIS]

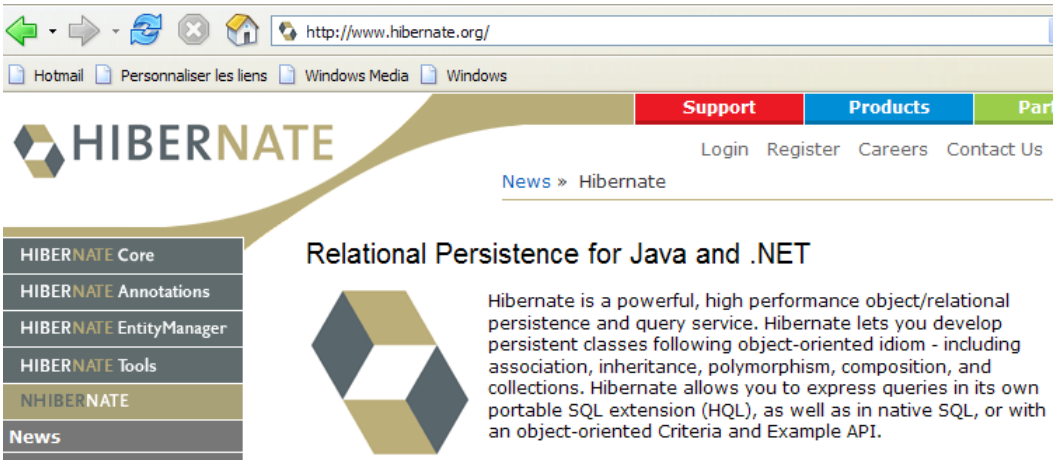
La classe Spring [SqlMapClientDaoSupport] utilise un framework tierce [Ibatis SqlMap] disponible à l'url [http://ibatis.apache.org/] :

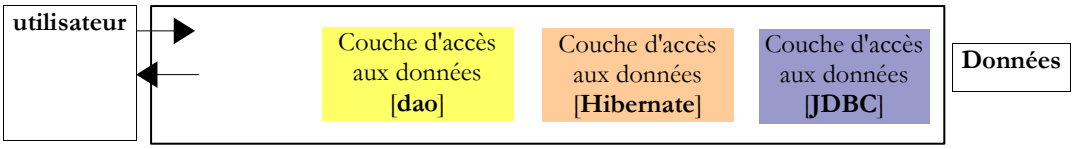


[iBATIS] est un projet Apache qui facilite la construction de couches [dao] s'appuyant sur des bases de données. Avec [iBATIS], l'architecture de la couche d'accès aux données est la suivante :



[iBATIS] s'insère entre la couche [dao] de l'application et le pilote JDBC de la base de données. Il existe des alternatives à [iBATIS] telle, par exemple, l'alternative [Hibernate] :



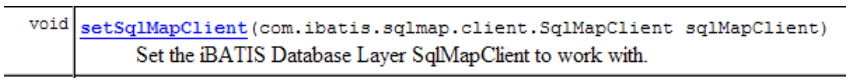


L'utilisation du framework [iBATIS] nécessite deux archives [ibatis-common, ibatis-sqlmap] qui ont été toutes deux placées dans le dossier [lib] du projet :

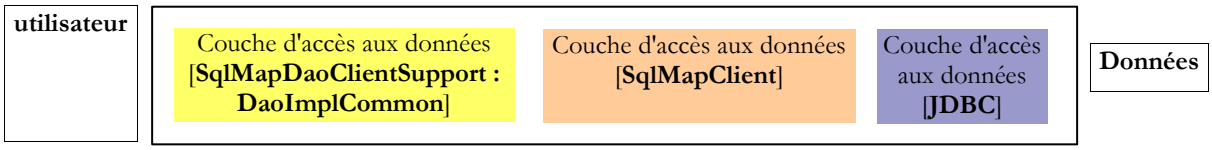
La classe [SqlMapClientDaoSupport] encapsule la partie générique de l'utilisation du framework [iBATIS], c.a.d. des parties de code qu'on retrouve dans toutes les couche [dao] utilisant l'outil [iBATIS]. Pour écrire la partie non générique du code, c'est à dire ce qui est spécifique à la couche [dao] que l'on écrit, il suffit de dériver la classe [SqlMapClientDaoSupport]. C'est ce que nous faisons ici.

La classe [SqlMapClientDaoSupport] est définie comme suit :

Parmi les méthodes de cette classe, l'une d'elles permet de configurer le client [iBATIS] avec lequel on va exploiter la base de données :



L'objet [SqlMapClient sqlMapClient] est l'objet [IBATIS] utilisé pour accéder à une base de données. A lui tout seul, il implémente la couche [iBATIS] de notre architecture :



Une séquence typique d'actions avec cet objet est la suivante :

1. demander une connexion à un pool de connexions
2. ouvrir une transaction
3. exécuter une série d'ordres SQL mémorisée dans un fichier de configuration
4. fermer la transaction
5. rendre la connexion au pool

Si notre implémentation [DaoImplCommon] travaillait directement avec [iBATIS], elle devrait faire cette séquence de façon répétée. Seule l'opération 3 est spécifique à une couche [dao], les autres opérations étant génériques. La classe Spring [SqlMapClientDaoSupport] assurera elle-même les opérations 1, 2, 4 et 5, déléguant l'opération 3 à sa classe dérivée, ici la classe [DaoImplCommon].

Pour pouvoir fonctionner, la classe [SqlMapClientDaoSupport] a besoin d'une référence sur l'objet iBATIS [SqlMapClient sqlMapClient] qui va assurer le dialogue avec la base de données. Cet objet a besoin de deux choses pour fonctionner :

- un objet [DataSource] connecté à la base de données auprès duquel il va demander des connexions
- un (ou des) fichier de configuration où sont externalisés les ordres SQL à exécuter. En effet, ceux-ci ne sont pas dans le code Java. Ils sont identifiés par un code dans un fichier de configuration et l'objet [SqlMapClient sqlMapClient] utilise ce code pour faire exécuter un ordre SQL particulier.

Un embryon de configuration de notre couche [dao] qui refléterait l'architecture ci-dessus serait le suivant :

```
1.  <!-- la classes d'accè à la couche [dao] -->
2.  <bean id="dao" class="istia.st.springmvc.personnes.dao.DaoImplCommon">
3.    <property name="sqlMapClient">
4.      <ref local="sqlMapClient"/>
5.    </property>
6.</bean>
```

Ici la propriété [sqlMapClient] (ligne 3) de la classe [DaoImplCommon] (ligne 2) est initialisée. Elle l'est par la méthode [setSqlMapClient] de la classe [DaoImpl]. Cette classe n'a pas cette méthode. C'est sa classe parent [SqlMapClientDaoSupport] qui l'a. C'est donc elle qui est en réalité initialisée ici.

Maintenant ligne 4, on fait référence à un objet nommé " sqlMapClient " qui reste à construire. Celui-ci, on l'a dit, est de type [SqlMapClient], un type [iBATIS] :

com.ibatis.sqlmap.client

Interface SqlMapClient

[SqlMapClient] est une interface. Spring offre la classe [SqlMapClientFactoryBean] pour obtenir un objet implémentant cette interface :

org.springframework.orm.ibatis

Class SqlMapClientFactoryBean

java.lang.Object
  └ org.springframework.orm.ibatis.SqlMapClientFactoryBean

All Implemented Interfaces:
FactoryBean, InitializingBean

Rappelons que nous cherchons à instancier un objet implémentant l'interface [SqlMapClient]. Ce n'est apparemment pas le cas de la classe [SqlMapClientFactoryBean]. Celle-ci implémente l'interface [FactoryBean] (cf ci-dessus). Celle-ci a la méthode [getObject()] suivante :

Object	getObject() Return an instance (possibly shared or independent) of the object managed by this factory.

Lorsqu'on demande à Spring une instance d'un objet implémentant l'interface [FactoryBean], il :

- crée une instance [I] de la classe - ici il crée une instance de type [SqlMapClientFactoryBean].
- rend à la méthode appelante, le résultat de la méthode [I].getObject() - la méthode [SqlMapClientFactoryBean].getObject() va rendre ici un objet implémentant l'interface [SqlMapClient].

Pour pouvoir rendre un objet implémentant l'interface [SqlMapClient], la classe [SqlMapClientFactoryBean] a besoin de deux informations nécessaires à cet objet :

- un objet [DataSource] connecté à la base de données auprès duquel il va demander des connexions
- un (ou des) fichier de configuration où sont externalisés les ordres SQL à exécuter

La classe [SqlMapClientFactoryBean] possède les méthodes **set** pour initialiser ces deux propriétés :

void	**setConfigLocation**(Resource configLocation) Set the location of the iBATIS SqlMapClient config file.
void	**setDataSource**(DataSource dataSource) Set the DataSource to be used by iBATIS SQL Maps.

Nous progressons... Notre fichier de configuration se précise et devient :

```
1.<!-- SqlMapCllient -->
2.  <bean id="sqlMapClient"
3.    class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
4.    <property name="dataSource">
5.      <ref local="dataSource"/>
6.    </property>
7.    <property name="configLocation">
8.      <value>classpath:sql-map-config-firebird.xml</value>
9.    </property>
10. </bean>
11. <!-- la classes d'accè à la couche [dao] -->
12. <bean id="dao" class="istia.st.springmvc.personnes.dao.DaoImplCommon">
13.   <property name="sqlMapClient">
14.     <ref local="sqlMapClient"/>
15.   </property>
16. </bean>
```

- lignes 2-3 : le bean " sqlMapClient " est de type [SqlMapClientFactoryBean]. De ce qui vient d'être expliqué, nous savons que lorsque nous demandons à Spring une instance de ce bean, nous obtenons un objet implémentant l'interface iBATIS [SqlMapClient]. C'est ce dernier objet qui sera donc obtenu en ligne 14.
- lignes 7-9 : nous indiquons que le fichier de configuration nécessaire à l'objet iBATIS [SqlMapClient] s'appelle " sql-map-config-firebird.xml " et qu'il doit être cherché dans le *ClassPath* de l'application. La méthode [SqlMapClientFactoryBean].setConfigLocation est ici utilisée.
- lignes 4-6 : nous initialisons la propriété [dataSource] de [SqlMapClientFactoryBean] avec sa méthode [setDataSource].

Ligne 5, nous faisons référence à un bean appelé " dataSource " qui reste à construire. Si on regarde le paramètre attendu par la méthode [setDataSource] de [SqlMapClientFactoryBean], on voit qu'il est de type [DataSource] :

javax.sql
Interface DataSource

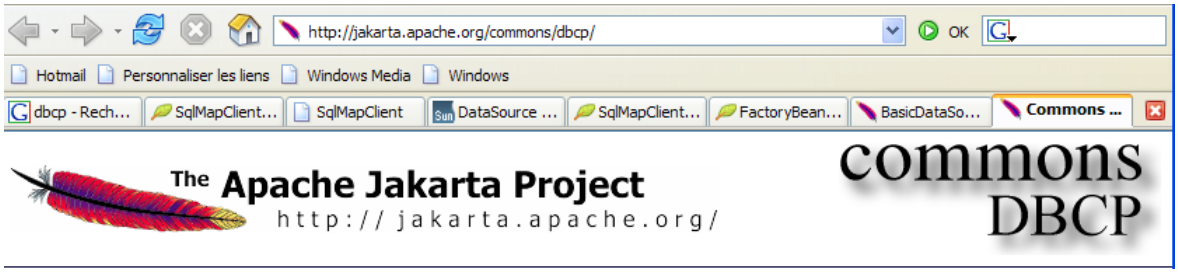
On a de nouveau affaire à une interface dont il nous faut trouver une classe d'implémentation. Le rôle d'une telle classe est de fournir à une application, de façon efficace, des connexions à une base de données particulière. Un SGBD ne peut maintenir ouvertes simultanément un grand nombre de connexions. Pour diminuer le nombre de connexions ouvertes à un moment donné, on est amenés, pour chaque échange avec la base, à :

- ouvrir une connexion
- commencer une transaction
- émettre des ordres SQL
- fermer la transaction
- fermer la connexion

Ouvrir et fermer des connexions de façon répétée est coûteux en temps. Pour résoudre ces deux problèmes (limiter à la fois le nombre de connexions ouvertes à un moment donné, et limiter le coût d'ouverture / fermeture de celles-ci, les classes implémentant l'interface [DataSource] procèdent souvent de la façon suivante :

- elles ouvrent dès leur instanciation, N connexions avec la base de données visée. N a en général une valeur par défaut et peut le plus souvent être défini dans un fichier de configuration. Ces N connexions vont rester tout le temps ouvertes et forment un pool de connexions disponibles pour les threads de l'application.
- lorsqu'un thread de l'application demande une ouverture de connexion, l'objet [DataSource] lui donne l'une des N connexions ouvertes au démarrage, s'il en reste de disponibles. Lorsque l'application ferme la connexion, cette dernière n'est en réalité pas fermée mais simplement remise dans le pool des connexions disponibles.

Il existe diverses implémentations de l'interface [DataSource] disponibles librement. Nous allons utiliser ici l'implémentation [commons DBCP] disponible à l'url [http://jakarta.apache.org/commons/dbcp/] :



L'utilisation de l'outil [commons DBCP] nécessite deux archives [commons-dbcp, commons-pool] qui ont été toutes deux placées dans le dossier [lib] du projet :

La classe [BasicDataSource] de [commons DBCP] fournit l'implémentation [DataSource] dont nous avons besoin :

Cette classe va nous fournir un pool de connexions pour accéder à la base Firebird [dbpersonnes.gdb] de notre application. Pour cela, il faut lui donner les informations dont elle a besoin pour créer les connexions du pool :

1. le nom du pilote JDBC à utiliser – initialisé avec [setDriverClassName]
2. le nom de l'url de la base de données à exploiter - initialisé avec [setUrl]
3. l'identifiant de l'utilisateur propriétaire de la connexion – initialisé avec [setUser**n**ame] (et non pas setUser**N**ame comme on aurait pu s'y attendre)
4. son mot de passe - initialisé avec [setPassword]

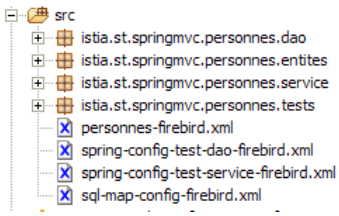
Le fichier de configuration de notre couche [dao] pourra être le suivant :

```
1.  <?xml version="1.0" encoding="ISO_8859-1"?>
2.  <!DOCTYPE beans SYSTEM "http://www.springframework.org/dtd/spring-beans.dtd">
3.  <beans>
4.    <!-- la source de donnéees DBCP -->
5.    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
6.      destroy-method="close">
7.      <property name="driverClassName">
8.        <value>org.firebirdsql.jdbc.FBDriver</value>
9.      </property>
10.     <!-- attention : ne pas laisser d'espaces entre les deux balises <value> de l'url -->
11.     <property name="url">
12.       <value>jdbc:firebirdsql:localhost/3050:C:/data/2005-2006/webjava/dvp-spring-mvc/mvc-
    38/database/dbpersonnes.gdb</value>
13.     </property>
14.     <property name="username">
15.       <value>sysdba</value>
```

```
16.     </property>
17.     <property name="password">
18.       <value>masterkey</value>
19.     </property>
20.   </bean>
21.   <!-- SqlMapCllient -->
22.   <bean id="sqlMapClient"
23.     class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
24.     <property name="dataSource">
25.       <ref local="dataSource"/>
26.     </property>
27.     <property name="configLocation">
28.       <value>classpath:sql-map-config-firebird.xml</value>
29.     </property>
30.   </bean>
31.   <!-- la classes d'accè à la couche [dao] -->
32.   <bean id="dao" class="istia.st.springmvc.personnes.dao.DaoImplCommon">
33.     <property name="sqlMapClient">
34.       <ref local="sqlMapClient"/>
35.     </property>
36.   </bean>
37. </beans>
```

- lignes 7-9 : le nom du pilote JDBC du SGBD Firebird
- lignes 11-13 : l'url de la base Firebird [dbpersonnes.gdb]. On fera particulièrement attention à l'écriture de celle-ci. Il ne doit y avoir aucun espace entre les balises <value> et l'url.
- lignes 14-16 : le propriétaire de la connexion – ici, [sysdba] qui est l'administrateur par défaut des distributions Firebird
- lignes 17-19 : son mot de passe [masterkey] – également la valeur par défaut

On a beaucoup progressé mais il reste toujours des points de configuration à élucider : la ligne 28 référence le fichier [sql-map-config-firebird.xml] qui doit configurer le client [SqlMapClient] d'iBATIS. Avant d'étudier son contenu, montrons l'emplacement de ces fichiers de configuration dans notre projet Eclipse :



- [spring-config-test-dao-firebird.xml] est le fichier de configuration de la couche [dao] que nous venons d'étudier
- [sql-map-config-firebird.xml] est référencé par [spring-config-test-dao-firebird.xml]. Nous allons l'étudier.
- [personnes-firebird.xml] est référencé par [sql-map-config-firebird.xml]. Nous allons l'étudier.

Les trois fichiers précédents sont dans le dossier [src]. Sous Eclipse, cela signifie qu'à l'exécution ils seront présents dans le dossier [bin] du projet (non représenté ci-dessus). Ce dossier fait partie du *ClassPath* de l'application. Au final, les trois fichiers précédents seront donc bien présents dans le *ClassPath* de l'application. C'est nécessaire.

Le fichier [sql-map-config-firebird.xml] est le suivant :

```
1.  <?xml version="1.0" encoding="UTF-8" ?>
2.  <!DOCTYPE sqlMapConfig
3.      PUBLIC "-//iBATIS.com//DTD SQL Map Config 2.0//EN"
4.      "http://www.ibatis.com/dtd/sql-map-config-2.dtd">
5.
6.  <sqlMapConfig>
7.    <sqlMap resource="personnes-firebird.xml"/>
8.  </sqlMapConfig>
```

- ce fichier doit avoir <sqlMapConfig> comme balise racine (lignes 6 et 8)
- ligne 7 : la balise <sqlMap> sert à désigner les fichiers qui contiennent les ordres SQL à exécuter. Il y a souvent, mais ce n'est pas obligatoire, un fichier par table. Cela permet de rassembler les ordres SQL sur une table donnée dans un même fichier. Mais on trouve fréquemment des ordres SQL impliquant plusieurs tables. Dans ce cas, la décomposition précédente ne tient pas. Il faut simplement se rappeler que l'ensemble des fichiers désignés par les balises <sqlMap> seront fusionnés. Ces fichiers sont cherchés dans le *ClassPath* de l'application.

Le fichier [personnes-firebird.xml] décrit les ordres SQL qui vont être émis sur la table [PERSONNES] de la base de données Firebird [dbpersonnes.gdb]. Son contenu est le suivant :

```
1.  <?xml version="1.0" encoding="UTF-8" ?>
2.
```

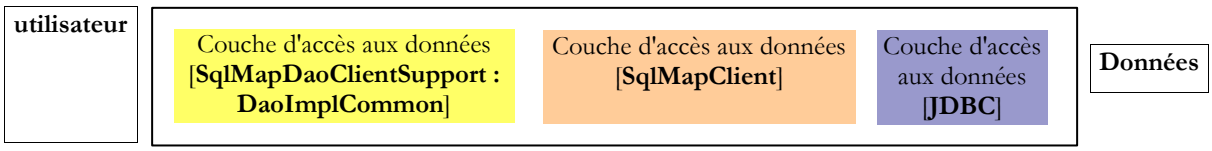
```
3.  <!DOCTYPE sqlMap
4.      PUBLIC "-//iBATIS.com//DTD SQL Map 2.0//EN"
5.      "http://www.ibatis.com/dtd/sql-map-2.dtd">
6.
7.  <sqlMap>
8.    <!-- alias classe [Personne] -->
9.    <typeAlias alias="Personne.classe"
10.     type="istia.st.springmvc.personnes.entites.Personne"/>
11.   <!-- mapping table [PERSONNES] - objet [Personne] -->
12.   <resultMap id="Personne.map"
13.     class="Personne.classe">
14.     <result property="id" column="ID" />
15.     <result property="version" column="VERSION" />
16.     <result property="nom" column="NOM"/>
17.     <result property="prenom" column="PRENOM"/>
18.     <result property="dateNaissance" column="DATENAISSANCE"/>
19.     <result property="marie" column="MARIE"/>
20.     <result property="nbEnfants" column="NBENFANTS"/>
21.   </resultMap>
22.   <!-- liste de toutes les personnes -->
23.   <select id="Personne.getAll" resultMap="Personne.map" > select ID, VERSION, NOM,
24.     PRENOM, DATENAISSANCE, MARIE, NBENFANTS FROM PERSONNES</select>
25.   <!-- obtenir une personne en particulier -->
26.     <select id="Personne.getOne" resultMap="Personne.map" >select ID, VERSION, NOM,
27.     PRENOM, DATENAISSANCE, MARIE, NBENFANTS FROM PERSONNES WHERE ID=#value#</select>
28.   <!-- ajouter une personne -->
29.   <insert id="Personne.insertOne" parameterClass="Personne.classe">
30.     <selectKey keyProperty="id">
31.       SELECT GEN_ID(GEN_PERSONNES_ID,1) as "value" FROM RDB$$DATABASE
32.     </selectKey>
33.     insert into
34.     PERSONNES(ID, VERSION, NOM, PRENOM, DATENAISSANCE, MARIE, NBENFANTS)
35.     VALUES(#id#, #version#, #nom#, #prenom#, #dateNaissance#, #marie#,
36.     #nbEnfants#) </insert>
37.   <!-- mettre à jour une personne -->
38.   <update id="Personne.updateOne" parameterClass="Personne.classe"> update
39.     PERSONNES set VERSION=#version#+1, NOM=#nom#, PRENOM=#prenom#, DATENAISSANCE=#dateNaissance#,
40.     MARIE=#marie#, NBENFANTS=#nbEnfants# WHERE ID=#id# and
41.     VERSION=#version#</update>
42.   <!-- supprimer une personne -->
43.   <delete id="Personne.deleteOne" parameterClass="int"> delete FROM PERSONNES WHERE
44.     ID=#value# </delete>
45. </sqlMap>
```

- le fichier doit avoir <sqlMap> comme balise racine (lignes 7 et 45)
- lignes 9-10 : pour faciliter l'écriture du fichier on donne l'alias (synonyme) [Personne.classe] à la classe [istia.st.springmvc.personnes.entites.Personne].
- lignes 12-21 : fixe les correspondances entre colonnes de la table [PERSONNES] et champs de l'objet [Personne].
- lignes 23-24 : l'ordre SQL [select] pour obtenir toutes les personnes de la table [PERSONNES]
- lignes 26-27 : l'ordre SQL [select] pour obtenir une personne particulière de la table [PERSONNES]
- lignes 29-36 : l'ordre SQL [insert] qui insère une personne dans la table [PERSONNES]
- lignes 38-41 : l'ordre SQL [update] qui met à jour une personne de la table [PERSONNES]
- lignes 42-44 : l'ordre SQL [delete] qui supprime une personne de la table [PERSONNES]

Le rôle et la signification du contenu du fichier [personnes-firebird.xml] vont être expliqués via l'étude de la classe [DaoImplCommon] qui implémente la couche [dao]

2.3.3 La classe [DaoImplCommon]

Revenons sur l'architecture d'accès aux données :



La classe [DaoImplCommon] est la suivante :

```
1.  package istia.st.springmvc.personnes.dao;
2.
3.  import istia.st.springmvc.personnes.entites.Personne;
4.  import org.springframework.orm.ibatis.support.SqlMapClientDaoSupport;
5.
6.  import java.util.Collection;
```

```
7.
8.  public class DaoImplCommon extends SqlMapClientDaoSupport implements
9.      IDao {
10.
11.   // liste des personnes
12.   public Collection getAll() {
13. ...
14.   }
15.
16.   // obtenir une personne en particulier
17.   public Personne getOne(int id) {
18. ...
19.   }
20.
21.   // suppression d'une personne
22.   public void deleteOne(int id) {
23. ...
24.   }
25.
26.   // ajouter ou modifier une personne
27.   public void saveOne(Personne personne) {
28.     // le paramètre personne est-il valide ?
29.     check(personne);
30.     // ajout ou modification ?
31.     if (personne.getId() == -1) {
32.       // ajout
33.       insertPersonne(personne);
34.     } else {
35.       updatePersonne(personne);
36.     }
37.   }
38.
39.   // ajouter une personne
40.   protected void insertPersonne(Personne personne) {
41. ...
42.   }
43.
44.   // modifier une personne
45.   protected void updatePersonne(Personne personne) {
46. ...
47.   }
48.
49.   // vérification validité d'une personne
50.   private void check(Personne p) {
51. ...
52.   }
53.
54. ...
55. }
```

Nous allons étudier les méthodes les unes après les autres.

getAll

Cette méthode permet d'obtenir toutes les personnes de la liste. Son code est le suivant :

```
1.  // liste des personnes
2.  public Collection getAll() {
3.    return getSqlMapClientTemplate().queryForList("Personne.getAll", null);
4.}
```

Rappelons-nous tout d'abord que la classe [DaoImplCommon] dérive de la classe Spring [SqlMapClientDaoSupport]. C'est cette classe qui a la méthode [getSqlMapClientTemplate()] utilisée ligne 3 ci-dessus. Cette méthode a la signature suivante :

SqlMapClientTemplate	getSqlMapClientTemplate() Return the SqlMapClientTemplate for this DAO, pre-initialized with the SqlMapClient or set explicitly.

Le type [SqlMapClientTemplate] encapsule l'objet [SqlMapClient] de la couche [iBATIS]. C'est par lui qu'on aura accès à la base de données. Le type [iBATIS] SqlMapClient pourrait être directement utilisé puisque la classe [SqlMapClientDaoSupport] y a accès :

com.ibatis.sqlmap.client.SqlMapClient	getSqlMapClient() Return the iBATIS Database Layer SqlMapClient that this template works with.

L'inconvénient de la classe [iBATIS] SqlMapClient est qu'elle lance des exceptions de type [SQLException], un type d'exception contrôlée, c.a.d. qui doit être gérée par un try / catch ou déclarée dans la signature des méthodes qui la lance. Or souvenons-nous que la couche [dao] implémente une interface [IDao] dont les méthodes ne comportent pas d'exceptions dans leurs signatures. Les

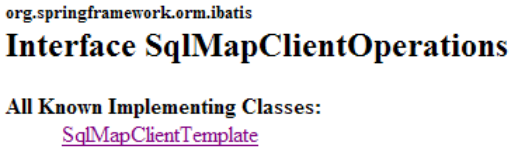
méthodes des classes d'implémentation de l'interface [IDao] ne peuvent donc, elles non plus, avoir d'exceptions dans leurs signatures. Il nous faut donc intercepter chaque exception [SQLException] lancée par la couche [iBATIS] et l'encapsuler dans une exception non contrôlée. Le type [DaoException] de notre projet ferait l'affaire pour cette encapsulation.

Plutôt que de gérer nous-mêmes ces exceptions, nous allons les confier au type Spring [SqlMapClientTemplate] qui encapsule l'objet [SqlMapClient] de la couche [iBATIS]. En effet [SqlMapClientTemplate] a été construit pour intercepter les exceptions [SQLException] lancées par la couche [SqlMapClient] et les encapsuler dans un type [DataAccessException] non contrôlé. Ce comportement nous convient. On se souviendra simplement que la couche [dao] est désormais susceptible de lancer deux types d'exceptions non contrôlées :

- notre type propriétaire [DaoException]
- le type Spring [DataAccessException]

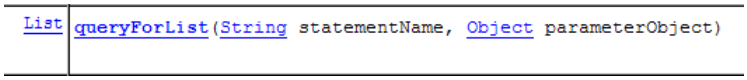
Le type [SqlMapClientTemplate] est défini comme suit :

Il implémente l'interface [SqlMapClientOperations] suivante :



Cette interface définit des méthodes capables d'exploiter le contenu du fichier [personnes-firebird.xml] :

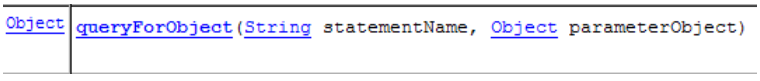
[**queryForList**]



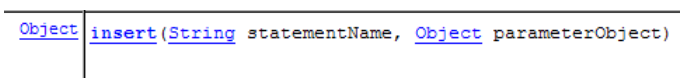
Cette méthode permet d'émettre un ordre [SELECT] et d'en récupérer le résultat sous forme d'une liste d'objets :

- [statementName] : l'identifiant (id) de l'ordre [select] dans le fichier de configuration
- [parameterObject] : l'objet " paramètre " pour un [select] paramétré. L'objet " paramètre " peut prendre deux formes :
  - un objet respectant la norme **Javabean** : les paramètres de l'ordre [select] sont alors les noms des champs du Javabean. A l'exécution de l'ordre [select], ils sont remplacés par les valeurs de ces champs.
  - un **dictionnaire** : les paramètres de l'ordre [select] sont alors les clés du dictionnaire. A l'exécution de l'ordre [select], celles-ci sont remplacées par leurs valeurs associées dans le dictionnaire.
- si le [SELECT] ne ramène aucune ligne, le résultat [List] est un objet vide d'éléments mais pas *null* (**à vérifier**).

[**queryForObject**]



Cette méthode est identique dans son esprit à la précédente mais elle ne ramène qu'un unique objet. Si le [SELECT] ne ramène aucune ligne, le résultat est le pointeur *null*.



[**insert**]

Cette méthode permet d'exécuter un ordre SQL [insert] paramétré par le second paramètre. L'objet rendu est la clé primaire de la ligne qui a été insérée. Il n'y a pas d'obligation à utiliser ce résultat.

[**update**]

```
int update(String statementName, Object parameterObject)
```

Cette méthode permet d'exécuter un ordre SQL [update] paramétré par le second paramètre. Le résultat est le nombre de lignes modifiées par l'ordre SQL [update].

[**delete**]

```
int delete(String statementName, Object parameterObject)
```

Cette méthode permet d'exécuter un ordre SQL [delete] paramétré par le second paramètre. Le résultat est le nombre de lignes supprimées par l'ordre SQL [delete].

Revenons à la méthode [getAll] de la classe [DaoImplCommon] :

```
1.  // liste des personnes
2.  public Collection getAll() {
3.    return getSqlMapClientTemplate().queryForList("Personne.getAll", null);
4.}
```

- ligne 4 : l'ordre [select] nommé " Personne.getAll " est exécuté. Il n'est pas paramétré et donc l'objet " paramètre " est *null*.

Dans [personnes-firebird.xml], l'ordre [select] nommé " Personne.getAll " est le suivant :

```
1.  <?xml version="1.0" encoding="UTF-8" ?>
2.
3.  <!DOCTYPE sqlMap
4.      PUBLIC "-//iBATIS.com//DTD SQL Map 2.0//EN"
5.      "http://www.ibatis.com/dtd/sql-map-2.dtd">
6.
7.  <sqlMap>
8.    <!-- alias classe [Personne] -->
9.    <typeAlias alias="Personne.classe"
10.     type="istia.st.springmvc.personnes.entites.Personne"/>
11.   <!-- mapping table [PERSONNES] - objet [Personne] -->
12.   <resultMap id="Personne.map"
13.     class="Personne.classe">
14.     <result property="id" column="ID" />
15.     <result property="version" column="VERSION" />
16.     <result property="nom" column="NOM"/>
17.     <result property="prenom" column="PRENOM"/>
18.     <result property="dateNaissance" column="DATENAISSANCE"/>
19.     <result property="marie" column="MARIE"/>
20.     <result property="nbEnfants" column="NBENFANTS"/>
21.   </resultMap>
22.   <!-- liste de toutes les personnes -->
23.   <select id="Personne.getAll" resultMap="Personne.map" > select ID, VERSION, NOM,
24.     PRENOM, DATENAISSANCE, MARIE, NBENFANTS FROM PERSONNES</select>
25. ...
26. </sqlMap>
```

- ligne 23 : l'ordre SQL " Personne.getAll " est non paramétré (absence de paramètres dans le texte de la requête).
- la ligne 3 de la méthode [getAll] demande l'exécution de la requête [select] appelée " Personne.getAll ". Celle-ci va être exécutée. [iBATIS] s'appuie sur JDBC. On sait alors que le résultat de la requête va être obtenu sous la forme d'un objet [ResultSet]. Ligne 23, l'attribut [resultMap] de la balise <select> indique à [iBATIS] quel " resultMap " il doit utiliser pour transformer chaque ligne du [ResultSet] obtenu en objet. C'est le " resultMap " [Personne.map] défini lignes 12-21 qui indique comment passer d'une ligne de la table [PERSONNES] à un objet de type [Personne]. [iBATIS] va utiliser ces correspondances pour fournir une liste d'objets [Personne] à partir des lignes de l'objet [ResultSet].
- la ligne 3 de la méthode [getAll] renvoie alors une collection d'objets [Personne]
- la méthode [queryForList] peut lancer une exception Spring [DataAccessException]. Nous la laissons remonter.

Nous expliquons les autres méthodes de la classe [AbstractDaoImpl] plus rapidement, l'essentiel sur l'utilisation d'[iBATIS] ayant été dit dans l'étude de la méthode [getAll].

---

getOne

Cette méthode permet d'obtenir une personne identifiée par son [id]. Son code est le suivant :

```
1.    // obtenir une personne en particulier
2.  public Personne getOne(int id) {
3.    // on la récupère dans la BD
4.    Personne personne = (Personne) getSqlMapClientTemplate()
5.        .queryForObject("Personne.getOne", new Integer(id));
6.    // a-t-on récupéré qq chose ?
7.    if (personne == null) {
8.      // on lance une exception
9.      throw new DaoException(
10.         "La personne d'id [" + id + "] n'existe pas", 2);
11.   }
12.   // on rend la personne
13.   return personne;
14. }
```

- ligne 4 : demande l'exécution de l'ordre [select] nommé " Personne.getOne ". Celui-ci est le suivant dans le fichier [personnes-firebird.xml] :

```
1.  <!-- obtenir une personne en particulier -->
2.      <select id="Personne.getOne" resultMap="Personne.map" parameterClass="int">
3.        select ID, VERSION, NOM, PRENOM, DATENAISSANCE, MARIE, NBENFANTS FROM
4.        PERSONNES WHERE ID=#value#</select>
```

L'ordre SQL est paramétré par le paramètre #value# (ligne 4). L'attribut #value# désigne la valeur du paramètre passé à l'ordre SQL, lorsque ce paramètre est de type simple : Integer, Double, String, ... Dans les attributs de la balise <select>, l'attribut [parameterClass] indique que le paramètre est de type entier (ligne 2). Ligne 5 de [getOne], on voit que ce paramètre est l'identifiant de la personne cherchée sous la forme d'un objet *Integer*. Ce changement de type est obligatoire puisque le second paramètre de [queryForList] doit être de type [Object].

Le résultat de la requête [select] sera à transformer en objet via l'attribut [resultMap="Personne.map"] (ligne 2). On obtiendra donc un type [Personne].

- lignes 7-11 : si la requête [select] n'a ramené aucune ligne, on récupère alors le pointeur *null* en ligne 4. Cela signifie qu'on n'a pas trouvé la personne cherchée. Dans ce cas, on lance une [DaoException] de code 2 (lignes 9-10).
- ligne 13 : s'il n'y a pas eu d'exception, alors on rend l'objet [Personne] demandé.

---

deleteOne

Cette méthode permet de supprimer une personne identifiée par son [id]. Son code est le suivant :

```
1.  // suppression d'une personne
2.  public void deleteOne(int id) {
3.    // on supprime la personne
4.    int n = getSqlMapClientTemplate().delete("Personne.deleteOne",
5.        new Integer(id));
6.    // a-t-on réussi
7.    if (n == 0) {
8.      throw new DaoException("Personne d'id [" + id + "] inconnue", 2);
9.    }
10. }
```

- lignes 4-5 : demande l'exécution de l'ordre [delete] nommé " Personne.deleteOne ". Celui-ci est le suivant dans le fichier [personnes-firebird.xml] :

```
1.<!-- supprimer une personne -->
2.  <delete id="Personne.deleteOne" parameterClass="int"> delete FROM PERSONNES WHERE
3.    ID=#value# </delete>
```

L'ordre SQL est paramétré par le paramètre #value# (ligne 3) de type [parameterClass="int"] (ligne 2). Ce sera l'identifiant de la personne cherchée (ligne 5 de *deleteOne*)

- ligne 4 : le résultat de la méthode *[SqlMapClientTemplate].delete* est le nombre de lignes détruites.
- lignes 7-8 : si la requête [delete] n'a détruit aucune ligne, cela signifie que la personne n'existe pas. On lance une [DaoException] de code 2 (ligne 8).

---

saveOne

Cette méthode permet d'ajouter une nouvelle personne ou de modifier une personne existante. Son code est le suivant :

```
1.    // ajouter ou modifier une personne
2.  public void saveOne(Personne personne) {
3.    // le paramètre personne est-il valide ?
```

```
4.    check(personne);
5.    // ajout ou modification ?
6.    if (personne.getId() == -1) {
7.      // ajout
8.      insertPersonne(personne);
9.    } else {
10.     updatePersonne(personne);
11.   }
12. }
13....
```

- ligne 4 : on vérifie la validité de la personne avec la méthode [check]. Cette méthode existait déjà dans la version précédente et avait été alors commentée. Elle lance une [DaoException] si la personne est invalide. On laisse remonter celle-ci.
- ligne 6 : si on arrive là, c'est qu'il n'y a pas eu d'exception. La personne est donc valide.
- lignes 6-11 : selon l'id de la personne, on a affaire à un ajout (id= -1) ou à une mise à jour (id<> -1). Dans les deux cas, on fait appel à deux méthodes internes à la classe :
  - **insertPersonne** : pour l'ajout
  - **updatePersonne** : pour la mise à jour

---

insertPersonne

Cette méthode permet d'ajouter une nouvelle personne. Son code est le suivant :

```
1.// ajouter une personne
2.  protected void insertPersonne(Personne personne) {
3.    // 1ère version
4.    personne.setVersion(1);
5.    // on attend 10 ms - pour les tests mettre true au lieu de false
6.    if (true)
7.      wait(10);
8.    // on insère la nouvelle personne dans la table de la BD
9.    getSqlMapClientTemplate().insert("Personne.insertOne", personne);
10. }
```

- ligne 4 : on met à 1 le n° de version de la personne que l'on est en train de créer
- ligne 9 : on fait l'insertion via la requête nommée " Personne.insertOne " qui est la suivante :

```
1.      <insert id="Personne.insertOne" parameterClass="Personne.classe">
2.        <selectKey keyProperty="id">
3.          SELECT GEN_ID(GEN_PERSONNES_ID,1) as "value" FROM RDB$$DATABASE
4.        </selectKey>
5.      insert into
6.      PERSONNES(ID, VERSION, NOM, PRENOM, DATENAISSANCE, MARIE, NBENFANTS)
7.      VALUES(#id#, #version#, #nom#, #prenom#, #dateNaissance#, #marie#,
8.       #nbEnfants#) </insert>
```

C'est une requête paramétrée et le paramètre est de type [Personne] (parameterClass="Personne.classe", ligne 1). Les champs de l'objet [Personne] passés en paramètre (ligne 9 de *insertPersonne*) sont utilisés pour remplir les colonnes de la ligne qui va être insérée dans la table [PERSONNES] (lignes 5-8). On a un problème à résoudre. Lors d'une insertion, l'objet [Personne] à insérer a son id égal à -1. Il faut remplacer cette valeur par une clé primaire valide. On utilise pour cela les lignes 2-4 de la balise <**selectKey**> ci-dessus. Elles indiquent :

- la requête SQL à exécuter pour obtenir une valeur de clé primaire. Celle indiquée ici est celle que nous avons présentée au paragraphe 2.1 - page 4. Deux points sont à noter :
  - **as " value "** est obligatoire. On peut aussi écrire *as value* mais *value* est un mot clé de Firebird qui a du être protégé par des guillemets.
  - la table Firebird s'appelle en réalité [RDB$DATABASE]. Mais le caractère $ est interprété par [iBATIS]. Il a été protégé en le dédoublant.
- le champ de l'objet [Personne] qu'il faut initialiser avec la valeur récupérée par l'ordre [SELECT], ici le champ [id]. C'est l'attribut [keyProperty] de la ligne 2 qui indique ce champ.

- lignes 6-7 : pour le besoin des tests, nous serons amenés à attendre 10 ms avant de faire l'insertion, ceci pour voir s'il y a des conflits entre threads qui voudraient faire en même temps des ajouts.

---

updatePersonne

Cette méthode permet de modifier une personne existant déjà dans la table [PERSONNES]. Son code est le suivant :

```
1.// modifier une personne
2.  protected void updatePersonne(Personne personne) {
3.    // on attend 10 ms - pour les tests mettre true au lieu de false
4.    if (true)
```

```
5.      wait(10);
6.    // modification
7.    int n = getSqlMapClientTemplate()
8.        .update("Personne.updateOne", personne);
9.    if (n == 0)
10.     throw new DaoException("La personne d'Id [" + personne.getId()
11.         + "] n'existe pas ou bien a été modifiée", 2);
12. }
```

- une mise à jour peut échouer pour au moins deux raisons :
  1. la personne à mettre à jour n'existe pas
  2. la personne à mettre à jour existe mais le thread qui veut la modifier n'a pas la bonne version

- lignes 7-8 : la requête SQL [update] nommée " Personne.updateOne " est exécutée. C'est la suivante :

```
1.    <!-- mettre à jour une personne -->
2.    <update id="Personne.updateOne" parameterClass="Personne.classe"> update
3.      PERSONNES set VERSION=#version#+1, NOM=#nom#, PRENOM=#prenom#,
    DATENAISSANCE=#dateNaissance#,
4.      MARIE=#marie#, NBENFANTS=#nbEnfants# WHERE ID=#id# and
5.  VERSION=#version#</update>
```

  - ligne 2 : la requête est paramétrée et admet pour paramètre un type [Personne] (parameterClass="Personne.classe"). Celui-ci est la personne à modifier (ligne 8 – updatePersonne).
  - on ne veut modifier que la personne de la table [PERSONNES] ayant le même n° [id] et la même version [version] que le paramètre. C'est pourquoi, on a la contrainte [WHERE ID=#id# and VERSION=#version#]. Si cette personne est trouvée, elle est mise à jour avec la personne paramètre et sa version est augmentée de 1 (ligne 3 ci-dessus).

- ligne 9 : on récupère le nombre de lignes mises à jour.
- lignes 10-11 : si ce nombre est nul, on lance une [DaoException] de code 2, indiquant que, soit la personne à mettre à jour n'existe pas, soit elle a changé de version entre-temps.

2.4 Tests de la couche [dao]

2.4.1 Tests de l'implémentation [DaoImplCommon]

Maintenant que nous avons écrit la couche [dao], nous nous proposons de la tester avec des tests JUnit :

Avant de faire des tests intensifs, nous pouvons commencer par un simple programme de type [main] qui va afficher le contenu de la table [PERSONNES]. C'est la classe [MainTestDaoFirebird] :

```
1.  package istia.st.springmvc.personnes.tests;
2.
3.  import istia.st.springmvc.personnes.dao.IDao;
4.
5.  import java.util.Collection;
6.  import java.util.Iterator;
7.
8.  import org.springframework.beans.factory.xml.XmlBeanFactory;
9.  import org.springframework.core.io.ClassPathResource;
10.
11. public class MainTestDaoFirebird {
12.   public static void main(String[] args) {
13.     IDao dao = (IDao) (new XmlBeanFactory(new ClassPathResource(
14.         "spring-config-test-dao-firebird.xml"))).getBean("dao");
15.     // liste actuelle
16.     Collection personnes = dao.getAll();
17.     // affichage console
18.     Iterator iter = personnes.iterator();
19.     while (iter.hasNext()) {
20.       System.out.println(iter.next());
21.     }
```

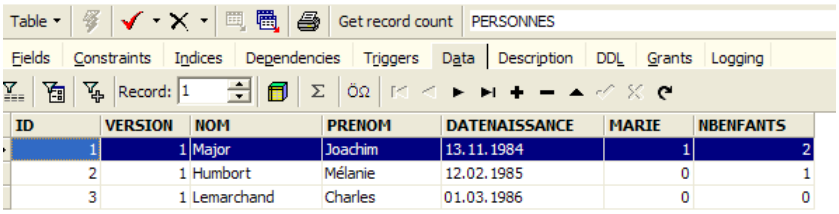
```
22.   }
23. }
```

Le fichier de configuration [spring-config-test-dao-firebird.xml] de la couche [dao], utilisé lignes 13-14, est le suivant :

```
1.  <?xml version="1.0" encoding="ISO_8859-1"?>
2.  <!DOCTYPE beans SYSTEM "http://www.springframework.org/dtd/spring-beans.dtd">
3.  <beans>
4.    <!-- la source de donnéees DBCP -->
5.    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
6.      destroy-method="close">
7.      <property name="driverClassName">
8.        <value>org.firebirdsql.jdbc.FBDriver</value>
9.      </property>
10.     <!-- attention : ne pas laisser d'espaces entre les deux balises <value> -->
11.     <property name="url">
12.       <value>jdbc:firebirdsql:localhost/3050:C:/data/2005-2006/webjava/dvp-spring-mvc/mvc-
    38/database/dbpersonnes.gdb</value>
13.     </property>
14.     <property name="username">
15.       <value>sysdba</value>
16.     </property>
17.     <property name="password">
18.       <value>masterkey</value>
19.     </property>
20.   </bean>
21.   <!-- SqlMapCllient -->
22.   <bean id="sqlMapClient"
23.     class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
24.     <property name="dataSource">
25.       <ref local="dataSource"/>
26.     </property>
27.     <property name="configLocation">
28.       <value>classpath:sql-map-config-firebird.xml</value>
29.     </property>
30.   </bean>
31.   <!-- la classes d'accè à la couche [dao] -->
32.   <bean id="dao" class="istia.st.springmvc.personnes.dao.DaoImplCommon">
33.     <property name="sqlMapClient">
34.       <ref local="sqlMapClient"/>
35.     </property>
36.   </bean>
37. </beans>
```

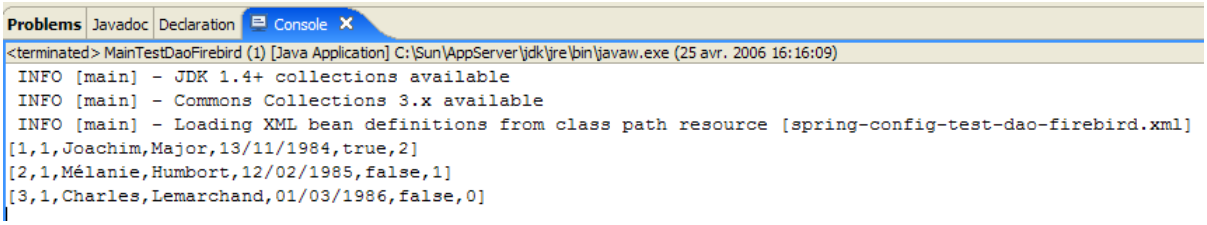
Ce fichier est celui étudié au paragraphe 2.3.2, page 12.

Pour le test, le SGBD Firebird est lancé. Le contenu de la table [PERSONNES] est le suivant :



ID	VERSION	NOM	PRENOM	DATENAISSANCE	MARIE	NBENFANTS
1	1	Major	Joachim	13.11.1984	1	2
2	1	Humbort	Mélanie	12.02.1985	0	1
3	1	Lemarchand	Charles	01.03.1986	0	0

L'exécution du programme [MainTestDaoFirebird] donne les résultats écran suivants :



On a bien obtenu la liste des personnes. On peut passer au test JUnit.

Le test JUnit [TestDaoFirebird] est le suivant :

```
1.  package istia.st.springmvc.personnes.tests;
2.
3.  import java.text.ParseException;
4.  import java.text.SimpleDateFormat;
5.  import java.util.Collection;
```

```
6.  import java.util.Iterator;
7.  import org.springframework.beans.factory.xml.XmlBeanFactory;
8.  import org.springframework.core.io.ClassPathResource;
9.
10. import istia.st.springmvc.personnes.dao.DaoException;
11. import istia.st.springmvc.personnes.dao.IDao;
12. import istia.st.springmvc.personnes.entites.Personne;
13. import junit.framework.TestCase;
14.
15. public class TestDaoFirebird extends TestCase {
16.
17.  // couche [dao]
18.  private IDao dao;
19.
20.  public IDao getDao() {
21.    return dao;
22.  }
23.
24.  public void setDao(IDao dao) {
25.    this.dao = dao;
26.  }
27.
28.  // constructeur
29.  public void setUp() {
30.    dao = (IDao) (new XmlBeanFactory(new ClassPathResource(
31.        "spring-config-test-dao-firebird.xml"))).getBean("dao");
32.  }
33.
34.  // liste des personnes
35.  private void doListe(Collection personnes) {
36. ...
37.  }
38.
39.  // test1
40.  public void test1() throws ParseException {
41. ...
42.  }
43.
44.  // modification-suppression d'un élément inexistant
45.  public void test2() throws ParseException {
46. ..
47.  }
48.
49.  // gestion des versions de personne
50.  public void test3() throws ParseException, InterruptedException {
51. ...
52.  }
53.
54.  // optimistic locking - accès multi-threads
55.  public void test4() throws Exception {
56. ...
57.  }
58.
59.  // tests de validité de saveOne
60.  public void test5() throws ParseException {
61. ....
62.  }
63.
64.  // insertions multi-threads
65.  public void test6() throws ParseException, InterruptedException{
66. ...
67. }
```

- les tests [test1] à [test5] sont les mêmes que dans la version 1, sauf [test4] qui a légèrement évolué. Le test [test6] est lui nouveau. Nous ne commentons que ces deux tests.

[test4]

[test4] a pour objectif de tester la méthode [updatePersonne - DaoImplCommon]. On rappelle le code de celle-ci :

```
1.// modifier une personne
2.  protected void updatePersonne(Personne personne) {
3.    // on attend 10 ms - pour les tests mettre true au lieu de false
4.    if (true)
5.      wait(10);
6.    // modification
7.    int n = getSqlMapClientTemplate()
8.        .update("Personne.updateOne", personne);
9.    if (n == 0)
10.     throw new DaoException("La personne d'Id [" + personne.getId()
11.         + "] n'existe pas ou bien a été modifiée", 2);
12. }
```

- lignes 4-5 : on attend 10 ms. On force ainsi le thread qui exécute [updatePersonne] à perdre le processeur, ce qui peut augmenter nos chances de voir des conflits d'accès entre threads concurrents.

[test4] lance N=100 threads chargés d'incrémenter, en même temps, de 1 le nombre d'enfants de la même personne. On veut voir comment les conflits de version et les conflits d'accès sont gérés.

```
1.   public void test4() throws Exception {
2.     // ajout d'une personne
3.     Personne p1 = new Personne(-1, "X", "X", new SimpleDateFormat(
4.         "dd/MM/yyyy").parse("01/02/2006"), true, 0);
5.     dao.saveOne(p1);
6.     int id1 = p1.getId();
7.     // création de N threads de mise à jour du nombre d'enfants
8.     final int N = 100;
9.     Thread[] taches = new Thread[N];
10.    for (int i = 0; i < taches.length; i++) {
11.      taches[i] = new ThreadDaoMajEnfants("thread n° " + i, dao, id1);
12.      taches[i].start();
13.    }
14.    // on attend la fin des threads
15.    for (int i = 0; i < taches.length; i++) {
16.      taches[i].join();
17.    }
18.    // on récupère la personne
19.    p1 = dao.getOne(id1);
20.    // elle doit avoir N enfants
21.    assertEquals(N, p1.getNbEnfants());
22.    // suppression personne p1
23.    dao.deleteOne(p1.getId());
24.    // vérification
25.    boolean erreur = false;
26.    int codeErreur = 0;
27.    try {
28.      p1 = dao.getOne(p1.getId());
29.    } catch (DaoException ex) {
30.      erreur = true;
31.      codeErreur = ex.getCode();
32.    }
33.    // on doit avoir une erreur de code 2
34.    assertTrue(erreur);
35.    assertEquals(2, codeErreur);
36.  }
```

Les threads sont créés lignes 8-13. Chacun va augmenter de 1 le nombre d'enfants de la personne créée lignes 3-5. Les threads [ThreadDaoMajEnfants] de mise à jour sont les suivants :

```
1.  package istia.st.springmvc.personnes.tests;
2.
3.  import java.util.Date;
4.
5.  import istia.st.springmvc.personnes.dao.DaoException;
6.  import istia.st.springmvc.personnes.dao.IDao;
7.  import istia.st.springmvc.personnes.entites.Personne;
8.
9.  public class ThreadDaoMajEnfants extends Thread {
10.  // nom du thread
11.  private String name;
12.
13.  // référence sur la couche [dao]
14.  private IDao dao;
15.
16.  // l'id de la personne sur qui on va travailler
17.  private int idPersonne;
18.
19.  // constructeur
20.  public ThreadDaoMajEnfants(String name, IDao dao, int idPersonne) {
21.    this.name = name;
22.    this.dao = dao;
23.    this.idPersonne = idPersonne;
24.  }
25.
26.  // coeur du thread
27.  public void run() {
28.    // suivi
29.    suivi("lancé");
30.    // on boucle tant qu'on n'a pas réussi à incrémenter de 1
31.    // le nbre d'enfants de la personne idPersonne
32.    boolean fini = false;
33.    int nbEnfants = 0;
34.    while (!fini) {
35.      // on récupère une copie de la personne d'idPersonne
36.      Personne personne = dao.getOne(idPersonne);
37.      nbEnfants = personne.getNbEnfants();
```

```
38.      // suivi
39.      suivi("" + nbEnfants + " -> " + (nbEnfants + 1)
40.          + " pour la version " + personne.getVersion());
41.      // attente de 10 ms pour abandonner le processeur
42.      try {
43.        // suivi
44.        suivi("début attente");
45.        // on s'interrompt pour laisser le processeur
46.        Thread.sleep(10);
47.        // suivi
48.        suivi("fin attente");
49.      } catch (Exception ex) {
50.        throw new RuntimeException(ex.toString());
51.      }
52.      // attente terminée - on essaie de valider la copie
53.      // entre-temps d'autres threads ont pu modifier l'original
54.      int codeErreur = 0;
55.      try {
56.        // incrémente de 1 le nbre d'enfants de cette copie
57.        personne.setNbEnfants(nbEnfants + 1);
58.        // on essaie de modifier l'original
59.        dao.saveOne(personne);
60.        // on est passé - l'original a été modifié
61.        fini = true;
62.      } catch (DaoException ex) {
63.        // on récupère le code erreur
64.        codeErreur = ex.getCode();
65.        // si une erreur d'ID ou de version de code erreur 2, on réessaie la mise à jour
66.        switch (codeErreur) {
67.        case 2:
68.          suivi("version corrompue ou personne inexistante");
69.          break;
70.        default:
71.          // exception non gérée - on laisse remonter
72.          throw ex;
73.        }
74.      }
75.    }
76.    // suivi
77.    suivi("a terminé et passé le nombre d'enfants à " + (nbEnfants + 1));
78.  }
79.
80.  // suivi
81.  private void suivi(String message) {
82.    System.out.println(name + " [" + new Date().getTime() + "] : "
83.        + message);
84.  }
85. }
```

Une mise à jour de personne peut échouer parce que la personne qu'on veut modifier n'existe pas ou qu'elle a été mise à jour auparavant par un autre thread. Ces deux cas sont ici gérés lignes 67-69. Dans ces deux cas en effet, la méthode [updatePersonne] lance une [DaoException] de code 2. Le thread sera alors ramené à recommencer la procédure de mise à jour depuis son début (boucle while, ligne 34).

[test6]

[test6] a pour objectif de tester la méthode [insertPersonne - DaoImplCommon]. On rappelle le code de celle-ci :

```
1.// ajouter une personne
2.  protected void insertPersonne(Personne personne) {
3.    // 1ère version
4.    personne.setVersion(1);
5.    // on attend 10 ms - pour les tests mettre true au lieu de false
6.    if (true)
7.      wait(10);
8.    // on insère la nouvelle personne dans la table de la BD
9.    getSqlMapClientTemplate().insert("Personne.insertOne", personne);
10. }
```

- lignes 6-7 : on attend 10 ms pour forcer le thread qui exécute [insertPersonne] à perdre le processeur et augmenter ainsi nos chances de voir apparaître des conflits dus à des threads qui font des insertions en même temps.

Le code de [test6] est le suivant :

```
1.  // insertions multi-threads
2.  public void test6() throws ParseException, InterruptedException{
3.    // création d'une personne
4.    Personne p = new Personne(-1, "X", "X", new SimpleDateFormat(
5.        "dd/MM/yyyy").parse("01/02/2006"), true, 0);
6.    // qu'on duplique N fois dans un tableau
7.    final int N = 100;
```

```
8.    Personne[] personnes=new Personne[N];
9.    for(int i=0;i<personnes.length;i++){
10.     personnes[i]=new Personne(p);
11.   }
12.   // création de N threads d'insertion - chaque thread insère 1 personne
13.   Thread[] taches = new Thread[N];
14.   for (int i = 0; i < taches.length; i++) {
15.     taches[i] = new ThreadDaoInsertPersonne("thread n° " + i, dao, personnes[i]);
16.     taches[i].start();
17.   }
18.   // on attend la fin des threads
19.   for (int i = 0; i < taches.length; i++) {
20.     // thread n° i
21.     taches[i].join();
22.     // supression personne
23.     dao.deleteOne(personnes[i].getId());
24.   }
25.}
```

On crée 100 threads qui vont insérer en même temps 100 personnes différentes. Ces 100 threads vont tous obtenir une clé primaire pour la personne qu'ils doivent insérer puis être interrompus pendant 10 ms (ligne 10 – insertPersonne) avant de pouvoir faire leur insertion. On veut vérifier que les choses se passent bien et que notamment ils obtiennent bien des valeurs de clé primaire différentes.

- lignes 7-11 : un tableau de 100 personnes est créé. Ces personnes sont toutes des copies de la personne p créée lignes 4-5.
- lignes 14-17 : les 100 threads d'insertion sont lancés. Chacun d'eux est chargé d'insérer l'une des 100 personnes créée précédemment..
- lignes 19-23 : [test6] attend la fin de chacun des 100 threads qu'il a lancés. Lorsqu'il a détecté la fin du thread n° i, il supprime la personne que ce thread vient d'insérer.

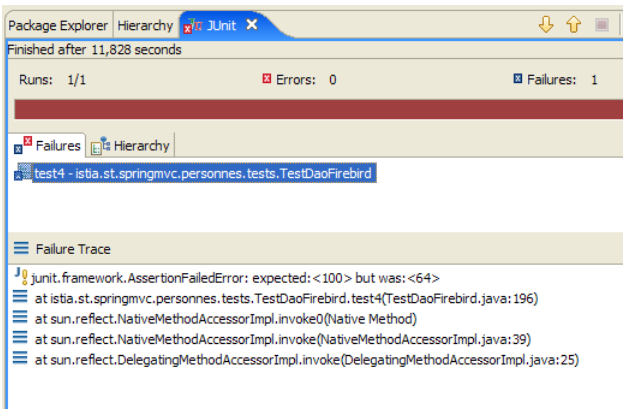
Le thread d'insertion [ThreadDaoInsertPersonne] est le suivant :

```
1.  package istia.st.springmvc.personnes.tests;
2.
3.  import java.util.Date;
4.
5.  import istia.st.springmvc.personnes.dao.IDao;
6.  import istia.st.springmvc.personnes.entites.Personne;
7.
8.  public class ThreadDaoInsertPersonne extends Thread {
9.    // nom du thread
10.   private String name;
11.
12.   // référence sur la couche [dao]
13.   private IDao dao;
14.
15.   // l'id de la personne sur qui on va travailler
16.   private Personne personne;
17.
18.   // constructeur
19.   public ThreadDaoInsertPersonne(String name, IDao dao, Personne personne) {
20.     this.name = name;
21.     this.dao = dao;
22.     this.personne = personne;
23.   }
24.
25.   // coeur du thread
26.   public void run() {
27.     // suivi
28.     suivi("lancé");
29.     // insertion
30.     dao.saveOne(personne);
31.     // suivi
32.     suivi("a terminé");
33.   }
34.
35.   // suivi
36.   private void suivi(String message) {
37.     System.out.println(name + " [" + new Date().getTime() + "] : "
38.         + message);
39.   }
40. }
```

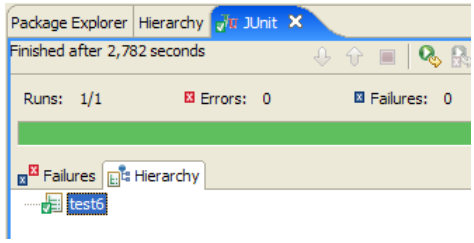
- lignes 19-22 : le constructeur du thread mémorise la personne qu'il doit insérer et la couche [dao] qu'il doit utiliser pour faire cette insertion.
- ligne 30 : la personne est insérée. Si une exception se produit, elle remonte à [test6].

Tests

Aux tests, on obtient les résultats suivants :



- le test [test4] échoue



- le test [test6] réussit

Le test [test4] échoue donc. Le nombre d'enfants est passé à 64 au lieu de 100 attendu. Que s'est-il passé ? Examinons les logs écran. Ils montrent l'existence d'exceptions lancées par Firebird :

```
1.  Exception in thread "Thread-62" org.springframework.jdbc.UncategorizedSQLException: SqlMapClient
    operation; uncategorized SQLException for SQL []; SQL state [HY000]; error code [335544336];
2.  --- The error occurred in personnes-firebird.xml.
3.  --- The error occurred while applying a parameter map.
4.  --- Check the Personne.updateOne-InlineParameterMap.
5.  --- Check the statement (update failed).
6.  --- Cause: org.firebirdsql.jdbc.FBSQLException: GDS Exception. 335544336. deadlock
7.  update conflicts with concurrent update; nested exception is
    com.ibatis.common.jdbc.exception.NestedSQLException:
8.  --- The error occurred in personnes-firebird.xml.
9.  --- The error occurred while applying a parameter map.
```

- ligne 1 – on a eu une exception Spring [org.springframework.jdbc.UncategorizedSQLException]. C'est une exception non contrôlée qui a été utilisée pour encapsuler une exception lancée par le pilote JDBC de Firebird, décrite ligne 6.
- ligne 6 – le pilote JDBC de Firebird a lancé une exception de type [org.firebirdsql.jdbc.FBSQLException] et de code d'erreur 335544336.
- ligne 7 : indique qu'on a eu un conflit d'accès entre deux threads qui voulaient mettre à jour en même temps la même ligne de la table [PERSONNES].

Ce n'est pas une erreur irrécupérable. Le thread qui intercepte cette exception peut retenter la mise à jour. Il faut pour cela modifier le code de [ThreadDaoMajEnfants] :

```
1.       try {
2.         // incrémente de 1 le nbre d'enfants de cette copie
3.         personne.setNbEnfants(nbEnfants + 1);
4.         // on essaie de modifier l'original
5.         dao.saveOne(personne);
6.         // on est passé - l'original a été modifié
7.         fini = true;
8.       } catch (DaoException ex) {
9.         // on récupère le code erreur
10.        codeErreur = ex.getCode();
11.        // si une erreur d'ID ou de version de code ereur 2, on réessaie la mise à jour
12.        switch (codeErreur) {
13.        case 2:
14.          suivi("version corrompue ou personne inexistante");
15.          break;
16.        default:
17.          // exception non gérée - on laisse remonter
18.          throw ex;
19.        }
```

- ligne 8 : on gère une exception de type [DaoException]. D'après ce qui a été dit, il nous faudrait gérer l'exception qui est apparue aux tests, le type [org.springframework.jdbc.UncategorizedSQLException]. On ne peut cependant pas se contenter de gérer ce type qui est un type générique de Spring destiné à encapsuler des exceptions qu'il ne connaît pas. Spring connaît les exceptions émises par les pilotes JDBC d'un certain nombre de SGBD tels Oracle, MySQL, Postgres, DB2, SQL Server, ... mais pas Firebird. Aussi toute exception lancée par le pilote JDBC de Firebird se trouve-t-elle encapsulée dans le type Spring [org.springframework.jdbc.UncategorizedSQLException] :

org.springframework.jdbc

Class UncategorizedSQLException

java.lang.Object
  └ java.lang.Throwable
    └ java.lang.Exception
      └ java.lang.RuntimeException
        └ org.springframework.core.NestedRuntimeException
          └ org.springframework.dao.DataAccessException
            └ org.springframework.dao.UncategorizedDataAccessException
              └ org.springframework.jdbc.UncategorizedSQLException

On voit ci-dessus, que la classe [UncategorizedSQLException] dérive de la classe [DataAccessException] que nous avons évoquée, paragraphe 2.3.3 – page 16. Il est possible de connaître l'exception qui a été encapsulée dans [UncategorizedSQLException] grâce à sa méthode [getSQLException] :

SQLException	getSQLException() Return the underlying SQLException.

Cette exception de type [SQLException] est celle lancée par la couche [iBATIS] qui elle même encapsule l'exception lancée par le pilote JDBC de la base de données. La cause exacte de l'exception de type [SQLException] peut être obtenue par la méthode :

Throwable	getCause() Returns the cause of this throwable or null if the cause is nonexistent or unknown.

On obtient l'objet de type [Throwable] qui a été lancé par le pilote JDBC :

java.lang

Class Throwable

java.lang.Object
  └ java.lang.Throwable

All Implemented Interfaces:
Serializable

Direct Known Subclasses:
Error, Exception

Le type [Throwable] est la classe parent de [Exception].

Ici il nous faudra vérifier que l'objet de type [Throwable] lancé par le pilote JDBC de Firebird et cause de l'exception [SQLException] lancée par la couche [iBATIS] est bien une exception de type [org.firebirdsql.gds.GDSException] et de code d'erreur 335544336. Pour récupérer le code erreur, nous pourrons utiliser la méthode [getErrorCode()] de la classe [org.firebirdsql.gds.GDSException].

Si nous utilisons dans le code de [ThreadDaoMajEnfants] l'exception [org.firebirdsql.gds.GDSException], alors ce thread ne pourra travailler qu'avec le SGBD Firebird. Il en sera de même du test [test4] qui utilise ce thread. Nous voulons éviter cela. En effet, nous souhaitons que nos tests JUnit restent valables quelque soit le SGBD utilisé. Pour arriver à ce résultat, on décide que la couche [dao] lancera une [DaoException] de code 4 lorsqu'une exception de type " conflit de mise à jour " est détectée et ce, quelque soit le SGBD sous-jacent. Ainsi, le thread [ThreadDaoMajEnfants] pourra-t-il être réécrit comme suit :

```
1.  package istia.st.springmvc.personnes.tests;
2.  ...
3.
4.  public class ThreadDaoMajEnfants extends Thread {
5.  ...
6.
7.    // coeur du thread
8.    public void run() {
9.  ...
10.     while (!fini) {
11.       // on récupère une copie de la personne d'idPersonne
12.       Personne personne = dao.getOne(idPersonne);
13.       nbEnfants = personne.getNbEnfants();
```

```
14. ...
15.      // attente terminée - on essaie de valider la copie
16.      // entre-temps d'autres threads ont pu modifier l'original
17.      int codeErreur = 0;
18.      try {
19.        // incrémente de 1 le nbre d'enfants de cette copie
20.        personne.setNbEnfants(nbEnfants + 1);
21.        // on essaie de modifier l'original
22.        dao.saveOne(personne);
23.        // on est passé - l'original a été modifié
24.        fini = true;
25.      } catch (DaoException ex) {
26.        // on récupère le code erreur
27.        codeErreur = ex.getCode();
28.        // si une erreur d'ID ou de version 2 ou un deadlock 4, on
29.        // réessaie la mise à jour
30.        switch (codeErreur) {
31.        case 2:
32.          suivi("version corrompue ou personne inexistante");
33.          break;
34.        case 4:
35.          suivi("conflit de mise à jour");
36.          break;
37.        default:
38.          // exception non gérée - on laisse remonter
39.          throw ex;
40.        }
41.      }
42.    }
43.    // suivi
44.    suivi("a terminé et passé le nombre d'enfants à " + (nbEnfants + 1));
45.  }
46. ...
47. }
```

- lignes 34-36 : l'exception de type [DaoException] de code 4 est interceptée. Le thread [ThreadDaoMajEnfants] va être forcé de recommencer la procédure de mise à jour à son début (ligne 10)

Notre couche [dao] doit donc être capable de reconnaître une exception de type " conflit de mise à jour ". Celle-ci est émise par un pilote JDBC et lui est spécifique. Cette exception doit être gérée dans la méthode [updatePersonne] de la classe [DaoImplCommon] :

```
1.  // modifier une personne
2.   protected void updatePersonne(Personne personne) {
3.     // on attend 10 ms - pour les tests mettre true au lieu de false
4.     if (true)
5.       wait(10);
6.     // modification
7.     int n = getSqlMapClientTemplate()
8.         .update("Personne.updateOne", personne);
9.     if (n == 0)
10.      throw new DaoException("La personne d'Id [" + personne.getId()
11.          + "] n'existe pas ou bien a été modifiée", 2);
12.  }
```

Les lignes 7-11 doivent être entourées par un try / catch. Pour le SGBD Firebird, il nous faut vérifier que l'exception qui a causé l'échec de la mise à jour est de type [org.firebirdsql.gds.GDSException] et a comme code d'erreur 335544336. Si on met ce type de test dans [DaoImplCommon], on va lier cette classe au SGBD Firebird, ce qui n'est évidemment pas souhaitable. Si on veut garder un caractère généraliste à la classe [DaoImplCommon], il nous faut la dériver et gérer l'exception dans une classe spécifique à Firebird. C'est ce que nous faisons maintenant.

2.4.2 La classe [DaoImplFirebird]

Son code est le suivant :

```
1.  package istia.st.springmvc.personnes.dao;
2.
3.  import istia.st.springmvc.personnes.entites.Personne;
4.
5.  public class DaoImplFirebird extends DaoImplCommon {
6.
7.   // modifier une personne
8.   protected void updatePersonne(Personne personne) {
9.     // on attend 10 ms - pour les tests mettre true au lieu de false
10.    if (true)
11.      wait(10);
12.    // modification
13.    try {
14.      // on modifie la personne qui a la bonne version
```

```
15.      int n = getSqlMapClientTemplate().update("Personne.updateOne",
16.          personne);
17.      if (n == 0)
18.        throw new DaoException("La personne d'Id [" + personne.getId()
19.            + "] n'existe pas ou bien a été modifiée", 2);
20.    } catch (org.springframework.jdbc.UncategorizedSQLException ex) {
21.      if (ex.getSQLException().getCause().getClass().isAssignableFrom(
22.          org.firebirdsql.jdbc.FBSQLException.class)) {
23.        org.firebirdsql.jdbc.FBSQLException cause = (org.firebirdsql.jdbc.FBSQLException) ex
24.            .getSQLException().getCause();
25.        if (cause.getErrorCode() == 335544336) {
26.          throw new DaoException(
27.              "Conflit d'accès au même enregistrement", 4);
28.        }
29.      } else {
30.        throw ex;
31.      }
32.    }
33.  }
34.
35.  // attente
36.  private void wait(int N) {
37.    // on attend N ms
38.    try {
39.      Thread.sleep(N);
40.    } catch (InterruptedException e) {
41.      // on affiche la trace de l'exception
42.      e.printStackTrace();
43.      return;
44.    }
45.  }
46.
47. }
```

- ligne 5 : la classe [DaoImplFirebird] dérive de [DaoImplCommon], la classe que nous venons d'étudier. Elle redéfinit, lignes 8-33, la méthode [updatePersonne] qui nous pose problème.
- lignes 20 : nous interceptons l'exception Spring de type [UncategorizedSQLException]
- lignes 21-22 : nous vérifions que l'exception sous-jacente de type [SQLException] et lancée par la couche [iBATIS] a pour cause une exception de type [org.firebirdsql.jdbc.FBSQLException]
- ligne 25 : on vérifie de plus que le code erreur de cette exception Firebird est 335544336, le code d'erreur du " deadlock ".
- lignes 26-27 : si toutes ces conditions sont réunies, une [DaoException] de code 4 est lancée.
- lignes 36-44 : la méthode [wait] permet d'arrêter le thread courant de N millisecondes. Elle n'a d'utilité que pour les tests.

Nous sommes prêts pour les tests de la nouvelle couche [dao].

2.4.3 Tests de l'implémentation [DaoImplFirebird]

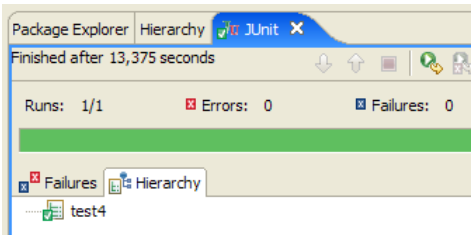
Le fichier de configuration des tests [spring-config-test-dao-firebird.xml] est modifié pour utiliser l'implémentation [DaoImplFirebird] :

```
1.  <?xml version="1.0" encoding="ISO_8859-1"?>
2.  <!DOCTYPE beans SYSTEM "http://www.springframework.org/dtd/spring-beans.dtd">
3.  <beans>
4.    <!-- la source de donnéees DBCP -->
5.    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
6.      destroy-method="close">
7.      <property name="driverClassName">
8.        <value>org.firebirdsql.jdbc.FBDriver</value>
9.      </property>
10.     <!-- attention : ne pas laisser d'espaces entre les deux balises <value> -->
11.     <property name="url">
12.       <value>jdbc:firebirdsql:localhost/3050:C:/data/2005-2006/webjava/dvp-spring-mvc/mvc-
    38/database/dbpersonnes.gdb</value>
13.     </property>
14.     <property name="username">
15.       <value>sysdba</value>
16.     </property>
17.     <property name="password">
18.       <value>masterkey</value>
19.     </property>
20.   </bean>
21.   <!-- SqlMapCllient -->
22.   <bean id="sqlMapClient"
23.     class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
24.     <property name="dataSource">
25.       <ref local="dataSource"/>
26.     </property>
27.     <property name="configLocation">
28.       <value>classpath:sql-map-config-firebird.xml</value>
29.     </property>
```

```
30.   </bean>
31.   <!-- la classes d'accè à la couche [dao] -->
32.   <bean id="dao" class="istia.st.springmvc.personnes.dao.DaoImplFirebird">
33.     <property name="sqlMapClient">
34.       <ref local="sqlMapClient"/>
35.     </property>
36.   </bean>
37. </beans>
```

- ligne 32 : la nouvelle implémentation [DaoImplFirebird] de la couche [dao].

Les résultats du test [test4] qui avait échoué précédemment sont les suivants :



[test4] a été réussi. Les dernières lignes de logs écran sont les suivantes :

```
1.  thread n° 36 [1145977145984] : fin attente
2.  thread n° 75 [1145977145984] : a terminé et passé le nombre d'enfants à 99
3.  thread n° 36 [1145977146000] : version corrompue ou personne inexistante
4.  thread n° 36 [1145977146000] : 99 -> 100 pour la version 100
5.  thread n° 36 [1145977146000] : début attente
6.  thread n° 36 [1145977146015] : fin attente
7.  thread n° 36 [1145977146031] : a terminé et passé le nombre d'enfants à 100
```

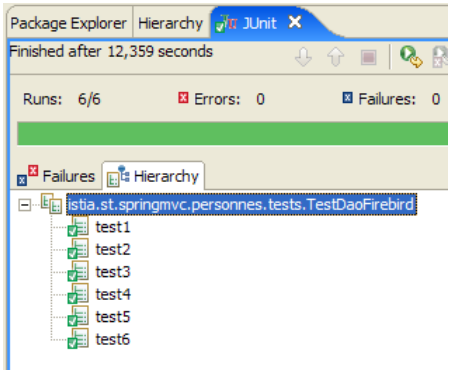
La dernière ligne indique que c'est le thread n° 36 qui a terminé le dernier. La ligne 3 montre un conflit de version qui a forcé le thread n° 36 à reprendre sa procédure de mise à jour de la personne (ligne 4). D'autres logs montrent des conflits d'accès lors des mises à jour :

```
1.  thread n° 52 [1145977145765] : version corrompue ou personne inexistante
2.  thread n° 75 [1145977145765] : conflit de mise à jour
3.  thread n° 36 [1145977145765] : version corrompue ou personne inexistante
```

La ligne 2 montre que le thread n° 75 a échoué lors de sa mise à jour à cause d'un conflit de mise à jour : lorsque la commande SQL [update] a été émise sur la table [PERSONNES], la ligne qu'il fallait mettre à jour était verrouillée par un autre thread. Ce conflit d'accès va obliger le thread n° 75 à retenter sa mise à jour.

Pour terminer avec [test4] on remarquera une différence notable avec les résultats du même test dans la version 1 où il avait échoué à cause de problèmes de synchronisation. Les méthodes de la couche [dao] de la version 1 n'étant pas synchronisées, des conflits d'accès apparaissaient. Ici, nous n'avons pas eu besoin de synchroniser la couche [dao]. Nous avons simplement géré les conflits d'accès signalés par Firebird.

Exécutons maintenant la totalité du test JUnit de la couche [dao] :

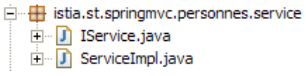


Il semble donc qu'on ait une couche [dao] valide. Pour la déclarer valide avec une forte probabilité, il nous faudrait faire davantage de tests encore. Néanmoins, nous la considèrerons comme opérationnelle.

2.5 La couche [service]

2.5.1 Les composantes de la couche [service]

La couche [service] est constituée des classes et interfaces suivantes :



- [IService] est l'interface présentée par la couche [service]
- [ServiceImpl] est une implémentation de celle-ci

L'interface [IService] est la suivante :

```
1.  package istia.st.springmvc.personnes.service;
2.
3.  import istia.st.springmvc.personnes.entites.Personne;
4.
5.  import java.util.Collection;
6.
7.  public interface IService {
8.    // liste de toutes les personnes
9.    Collection getAll();
10.
11.   // obtenir une personne particulière
12.   Personne getOne(int id);
13.
14.   // ajouter/modifier une personne
15.   void saveOne(Personne personne);
16.
17.   // supprimer une personne
18.   void deleteOne(int id);
19.
20.   // sauvegarder plusieurs personnes
21.   void saveMany(Personne[] personnes);
22.
23.   // supprimer plusieurs personnes
24.   void deleteMany(int ids[]);
25. }
```

- l'interface a les mêmes quatre méthodes que dans la version 1 mais elle en a deux de plus :
  - **saveMany** : permet de sauvegarder plusieurs personnes en même temps de façon atomique. Soit elles sont toutes sauvegardées, soit aucune ne l'est.
  - **deleteMany** : permet de supprimer plusieurs personnes en même temps de façon atomique. Soit elles sont toutes supprimées, soit aucune ne l'est.

  Ces deux méthodes ne seront pas utilisées par l'application web. Nous les avons rajoutées pour illustrer la notion de **transaction** sur une base de données. Les deux méthodes devront en effet être exécutées au sein d'une transaction pour obtenir l'atomicité désirée.

La classe [ServiceImpl] implémentant cette interface sera la suivante :

```
1.  package istia.st.springmvc.personnes.service;
2.
3.  import istia.st.springmvc.personnes.entites.Personne;
4.  import istia.st.springmvc.personnes.dao.IDao;
5.
6.  import java.util.Collection;
7.
8.  public class ServiceImpl implements IService {
9.
10.   // la couche [dao]
11.   private IDao dao;
12.
13.   public IDao getDao() {
14.     return dao;
15.   }
16.
17.   public void setDao(IDao dao) {
```

```
18.    this.dao = dao;
19.  }
20.
21.  // liste des personnes
22.  public Collection getAll() {
23.    return dao.getAll();
24.  }
25.
26.  // obtenir une personne en particulier
27.  public Personne getOne(int id) {
28.    return dao.getOne(id);
29.  }
30.
31.  // ajouter ou modifier une personne
32.  public void saveOne(Personne personne) {
33.    dao.saveOne(personne);
34.  }
35.
36.  // suppression d'une personne
37.  public void deleteOne(int id) {
38.    dao.deleteOne(id);
39.  }
40.
41.  // sauvegarder une collection de personnes
42.  public void saveMany(Personne[] personnes) {
43.    // on boucle sur le tableau des personnes
44.    for (int i = 0; i < personnes.length; i++) {
45.      dao.saveOne(personnes[i]);
46.    }
47.  }
48.
49.  // supprimer une collection de personnes
50.  public void deleteMany(int[] ids) {
51.    // ids : les id des personnes à supprimer
52.    for (int i = 0; i < ids.length; i++) {
53.      dao.deleteOne(ids[i]);
54.    }
55.  }
56. }
```

- les méthodes [getAll, getOne, insertOne, saveOne] font appel aux méthodes de la couche [dao] de même nom.
- lignes 42-47 : la méthode [saveMany] sauvegarde, une par une, les personnes du tableau passé en paramètre.
- lignes 50-55 : la méthode [deleteMany] supprime, une par une, les personnes dont on lui a passé le tableau des *id* en paramètre

Nous avons dit que les méthodes [saveMany] et [deleteMany] devaient se faire au sein d'une transaction pour assurer l'aspect tout ou rien de ces méthodes. Nous pouvons constater que le code ci-dessus ignore totalement cette notion de transaction. Celle-ci n'apparaîtra que dans le fichier de configuration de la couche [service].

2.5.2 Configuration de la couche [service]

Ci-dessus, ligne 11, on voit que l'implémentation [ServiceImpl] détient une référence sur la couche [dao]. Celle-ci, comme dans la version 1, sera initialisée par Spring au moment de l'instanciation de la couche [service - ServiceImpl]. Le fichier de configuration qui permettra l'instanciation de la couche [service] sera le suivant :

```
1.  <?xml version="1.0" encoding="ISO_8859-1"?>
2.  <!DOCTYPE beans SYSTEM "http://www.springframework.org/dtd/spring-beans.dtd">
3.  <beans>
4.    <!-- la source de donnéees DBCP -->
5.    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
6.      destroy-method="close">
7.      <property name="driverClassName">
8.        <value>org.firebirdsql.jdbc.FBDriver</value>
9.      </property>
10.     <property name="url">
11.       <!-- attention : ne pas laisser d'espaces entre les deux balises <value> -->
12.       <value>jdbc:firebirdsql:localhost/3050:C:/data/2005-2006/webjava/dvp-spring-mvc/mvc-
    38/database/dbpersonnes.gdb</value>
13.     </property>
14.     <property name="username">
15.       <value>sysdba</value>
16.     </property>
17.     <property name="password">
18.       <value>masterkey</value>
19.     </property>
20.   </bean>
21.   <!-- SqlMapCllient -->
22.   <bean id="sqlMapClient"
23.     class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
24.     <property name="dataSource">
25.       <ref local="dataSource"/>
```

```
26.     </property>
27.     <property name="configLocation">
28.       <value>classpath:sql-map-config-firebird.xml</value>
29.     </property>
30.   </bean>
31.   <!-- la classes d'accès à la couche [dao] -->
32.   <bean id="dao" class="istia.st.springmvc.personnes.dao.DaoImplFirebird">
33.     <property name="sqlMapClient">
34.       <ref local="sqlMapClient"/>
35.     </property>
36.   </bean>
37.   <!-- gestionnaire de transactions -->
38.   <bean id="transactionManager"
39.     class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
40.     <property name="dataSource">
41.       <ref local="dataSource"/>
42.     </property>
43.   </bean>
44.   <!-- la classes d'accès à la couche [service] -->
45.   <bean id="service"
46.     class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
47.     <property name="transactionManager">
48.       <ref local="transactionManager"/>
49.     </property>
50.     <property name="target">
51.       <bean class="istia.st.springmvc.personnes.service.ServiceImpl">
52.         <property name="dao">
53.           <ref local="dao"/>
54.         </property>
55.       </bean>
56.     </property>
57.     <property name="transactionAttributes">
58.       <props>
59.         <prop key="get*">PROPAGATION_SUPPORTS,readOnly</prop>
60.         <prop key="save*">PROPAGATION_REQUIRED</prop>
61.         <prop key="delete*">PROPAGATION_REQUIRED</prop>
62.       </props>
63.     </property>
64.   </bean>
65. </beans>
```

- lignes 1-36 : configuration de la couche [dao]. Cette configuration a été expliquée lors de l'étude de la couche [dao] au paragraphe 2.3.2 – page 12.
- lignes 38-64 : configurent la couche [service]

Ligne 46, on peut voir que l'implémentation de la couche [service] est faite par le type [TransactionProxyFactoryBean]. On s'attendait à trouver le type [ServiceImpl]. [TransactionProxyFactoryBean] est un type prédéfini de Spring. Comment se peut-il qu'un type prédéfini puisse implémenter l'interface [IService] qui elle, est spécifique à notre application ?

Découvrons tout d'abord la classe [TransactionProxyFactoryBean] :

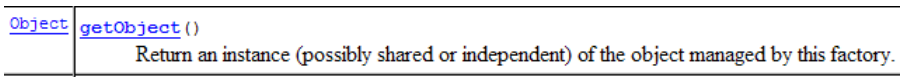
org.springframework.transaction.interceptor

Class TransactionProxyFactoryBean

java.lang.Object
  └ org.springframework.aop.framework.ProxyConfig
    └ org.springframework.transaction.interceptor.TransactionProxyFactoryBean

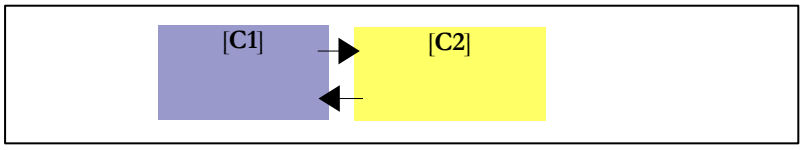
All Implemented Interfaces:
Serializable, BeanFactoryAware, FactoryBean, InitializingBean

Nous voyons qu'elle implémente l'interface [FactoryBean]. Nous avons déjà rencontré cette interface. Nous savons que lorsqu'une application demande à Spring, une instance d'un type implémentant [FactoryBean], Spring rend non pas une instance [I] de ce type, mais l'objet rendu par la méthode *[I].getObject()* :

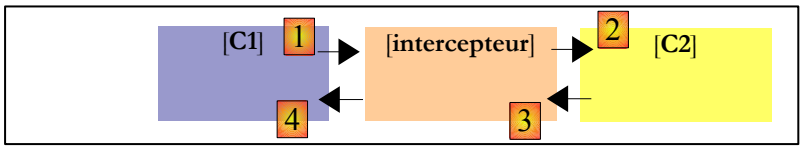

Object	getObject() Return an instance (possibly shared or independent) of the object managed by this factory.

Dans notre cas, la couche [service] va être implémentée par l'objet rendu par *[TransactionProxyFactoryBean].getObject()*. Quelle est la nature de cet objet ? Nous n'allons pas rentrer dans les détails car ils sont complexes. Ils relèvent de ce qu'on appelle Spring AOP (Aspect Oriented Programming). Nous allons tenter d'éclaircir les choses avec de simples schémas. AOP permet la chose suivante :

- on a deux classes C1 et C2, C1 utilisant l'interface [I2] présentée par C2 :



- grâce à AOP, on peut placer, de façon transparente pour les deux classes, un intercepteur entre les classes C1 et C2 :



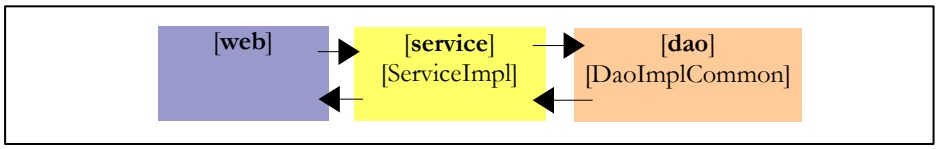
La classe [C1] a été compilée pour travailler avec l'interface [I2] que [C2] implémente. Au moment de l'exécution, AOP vient placer la classe [intercepteur] entre [C1] et [C2]. Pour que cela soit possible, il faut bien sûr que la classe [intercepteur] présente à [C1] la même interface [I2] que [C2].

A quoi cela peut-il servir ? La documentation Spring donne quelques exemples. On peut vouloir faire, par exemple, des logs à l'occasion des appels à une méthode M particulière de [C2], pour faire un audit de cette méthode. Dans [intercepteur], on écrira alors une méthode [M] qui fait ces logs. L'appel de [C1] à [C2].M va se passer ainsi (cf schéma ci-dessus) :

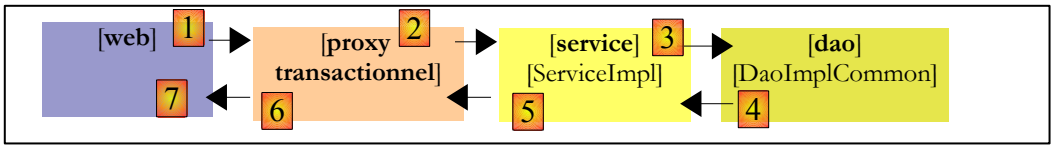
1. [C1] appelle la méthode M de [C2]. C'est en fait la méthode M de [intercepteur] qui sera appelée. Cela est possible si [C1] s'adresse à une interface [I2] plutôt qu'à une implémentation particulière de [I2]. Il suffit alors que [intercepteur] implémente [I2].
2. la méthode M de [intercepteur] fait les logs et appelle la méthode M de [C2] visée initialement par [C1].
3. la méthode M de [C2] s'exécute et rend son résultat à la méthode M de [intercepteur] qui peut éventuellement ajouter quelque chose à ce qui a été fait en 2.
4. la méthode M de [intercepteur] rend un résultat à la méthode appelante de [C1]

On voit que la méthode M de [intercepteur] peut faire quelque chose **avant** et **après** l'appel de la méthode M de [C2]. Vis à vis de [C1], elle enrichit donc la méthode M de [C2]. On peut donc voir la technologie AOP comme une façon d'enrichir l'interface présentée par une classe.

Comment ce concept s'applique-t-il à notre couche [service] ? Si on implémente la couche [service] directement avec une instance [ServiceImpl], notre application web aura l'architecture suivante :



Si on implémente la couche [service] avec une instance [TransactionProxyFactoryBean], on aura l'architecture suivante :



On peut dire que la couche [service] est instanciée avec deux objets :

- l'objet que nous appelons ci-dessus [proxy transactionnel] et qui est en fait l'objet rendu par la méthode [getObject] de [TransactionProxyFactoryBean]. C'est cet objet qui fera l'interface de la couche [service] avec la couche [web]. Il implémente par construction l'interface [IService].
- une instance [ServiceImpl] qui elle aussi implémente l'interface [IService]. Elle seule sait comment travailler avec la couche [dao], aussi est-elle nécessaire.

Imaginons que la couche [web] appelle la méthode [saveMany] de l'interface [IService]. Nous savons que fonctionnellement, les ajouts / mises à jour faits par cette méthode doivent l'être dans une transaction. Soit ils réussissent tous, soit aucun n'est fait. Nous

avons présenté la méthode [saveMany] de la classe [ServiceImpl] et nous avons pointé le fait qu'elle n'avait pas la notion de transaction. La méthode [saveMany] du [proxy transactionnel] va enrichir la méthode [saveMany] de la classe [ServiceImpl] avec cette notion de transaction. Suivons le schéma ci-dessus :

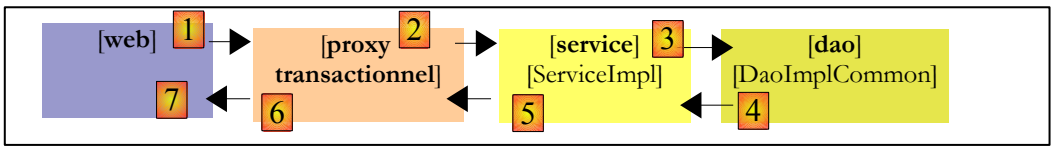
1. la couche [web] appelle la méthode [saveMany] de l'interface [IService].
2. la méthode [saveMany] de [proxy transactionnel] est exécutée. Elle commence une transaction. Il faut qu'elle ait les informations suffisantes pour le faire, notamment un objet [DataSource] pour obtenir une connexion au SGBD. Puis elle fait appel à la méthode [saveMany] de [ServiceImpl].
3. celle-ci s'exécute. Elle fait appel de façon répétée à la couche [dao] pour exécuter les insertions ou les mises à jour. Les ordres SQL exécutés à cette occasion le sont dans la transaction commencée en 2.
4. supposons qu'une de ces opérations échoue. La couche [dao] va laisser remonter une exception vers la couche [service], en l'occurrence la méthode [saveMany] de l'instance [ServiceImpl].
5. celle-ci ne fait rien et laisse remonter l'exception jusqu'à la méthode [saveMany] de [proxy transactionnel].
6. à réception de l'exception, la méthode [saveMany] de [proxy transactionnel] qui est propriétaire de la transaction fait un [rollback] de celle-ci pour annuler la totalité des mises à jour, puis laisse remonter l'exception jusqu'à la couche [web] qui sera chargée de la gérer.

A l'étape 4, nous avons supposé qu'une des insertions ou des mises à jour échouait. Si ce n'est pas le cas, en [5] aucune exception ne remonte. Idem en [6]. Dans ce cas, la méthode [saveMany] de [proxy transactionnel] fait un [commit] de la transaction pour valider la totalité des mises à jour.

Nous avons maintenant une idée plus précise de l'architecture mise en place par le bean [TransactionProxyFactoryBean]. Revenons sur la configuration de celui-ci :

```
1.  <!-- gestionnaire de transactions -->
2.  <bean id="transactionManager"
3.    class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
4.    <property name="dataSource">
5.      <ref local="dataSource"/>
6.    </property>
7.  </bean>
8.  <!-- la classes d'accès à la couche [service] -->
9.  <bean id="service"
10.   class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
11.   <property name="transactionManager">
12.     <ref local="transactionManager"/>
13.   </property>
14.   <property name="target">
15.     <bean class="istia.st.springmvc.personnes.service.ServiceImpl">
16.       <property name="dao">
17.         <ref local="dao"/>
18.       </property>
19.     </bean>
20.   </property>
21.   <property name="transactionAttributes">
22.     <props>
23.       <prop key="get*">PROPAGATION_REQUIRED,readOnly</prop>
24.       <prop key="save*">PROPAGATION_REQUIRED</prop>
25.       <prop key="delete*">PROPAGATION_REQUIRED</prop>
26.     </props>
27.   </property>
28. </bean>
```

Suivons cette configuration à la lumière de l'architecture qui est configurée :



- [proxy transactionnel] va gérer les transactions. Spring offre plusieurs stratégies de gestion de celles-ci. [proxy transactionnel] a besoin d'une référence sur le gestionnaire de transactions choisi.
- lignes 11 – 13 : définissent l'attribut [transactionManager] du bean [TransactionProxyFactoryBean] avec une référence sur un gestionnaire de transactions. Celui-ci est défini lignes 2 – 7.
- lignes 2-7 : le gestionnaire de transactions est de type [DataSourceTransactionManager] :

org.springframework.jdbc.datasource

Class DataSourceTransactionManager

java.lang.Object
└ org.springframework.transaction.support.AbstractPlatformTransactionManager
└ org.springframework.jdbc.datasource.DataSourceTransactionManager

All Implemented Interfaces:
Serializable, InitializingBean, PlatformTransactionManager

[DataSourceTransactionManager] est un gestionnaire de transactions adapté aux SGBD accédés via un objet [DataSource]. Il ne sait gérer que les transactions sur un unique SGBD. Il ne sait pas gérer des transactions distribuées sur plusieurs SGBD. Ici, nous n'avons qu'un seul SGBD. Aussi ce gestionnaire de transactions convient-il. Lorsque [proxy transactionnel] va démarrer une transaction, il va le faire sur une connexion attachée au thread. C'est cette connexion qui sera utilisée dans toutes les couches qui mènent à la base de données : [ServiceImpl, DaoImplCommon, SqlMapClientTemplate, JDBC].

La classe [DataSourceTransactionManager] a besoin de connaître la source de données auprès de laquelle elle doit demander une connexion pour l'attacher au thread. Celle-ci est définie lignes 4-6 : c'est la même source de données que celle utilisée par la couche [dao] (cf paragraphe 2.5.2, page 32).

- lignes 14-19 : l'attribut " **target** " indique la classe qui doit être interceptée, ici la classe [ServiceImpl]. Cette information est nécessaire pour deux raisons :
  - la classe [ServiceImpl] doit être instanciée puisque c'est elle qui assure le dialogue avec la couche [dao]
  - [TransactionProxyFactoryBean] doit générer un proxy qui présente à la couche [web] la même interface que [ServiceImpl].

- lignes 21-27 : indiquent quelles méthodes de [ServiceImpl], le proxy doit intercepter. L'attribut [**transactionAttributes**], ligne 21, indique quelles méthodes de [ServiceImpl] nécessitent une transaction et quels sont les attributs de celle-ci :
  - ligne 23 : les méthodes dont le nom commencent par **get** [getOne, getAll] s'exécutent dans une transaction d'attribut [PROPAGATION_REQUIRED,readOnly] :
    - PROPAGATION_REQUIRED : la méthode s'exécute dans une transaction s'il y en a déjà une attachée au thread, sinon une nouvelle est créée et la méthode s'exécute dedans.
    - readOnly : transaction en lecture seule

    Ici les méthodes [getOne] et [getAll] de [ServiceImpl] s'exécuteront dans une transaction alors qu'en fait ce n'est pas nécessaire. Il s'agit à chaque fois d'une opération constituée d'un unique ordre SELECT. On ne voit pas l'utilité de mettre ce SELECT dans une transaction.
  - ligne 24 : les méthodes dont le nom commencent par **save,** [saveOne, saveMany] s'exécutent dans une transaction d'attribut [PROPAGATION_REQUIRED].
  - ligne 25 : les méthodes [deleteOne] et [deleteMany] de [ServiceImpl] sont configurées de façon identique aux méthodes [saveOne, saveMany].

Dans notre couche [service], seules les méthodes [saveMany] et [deleteMany] ont besoin de s'exécuter dans une transaction. La configuration aurait pu être réduite aux lignes suivantes :

```
1.     <property name="transactionAttributes">
2.       <props>
3.         <prop key="saveMany">PROPAGATION_REQUIRED</prop>
4.         <prop key="deleteMany">PROPAGATION_REQUIRED</prop>
5.       </props>
6.   </property>
```

2.6 Tests de la couche [service]

Maintenant que nous avons écrit et configuré la couche [service], nous nous proposons de la tester avec des tests JUnit :

Le fichier de configuration [spring-config-test-service-firebird.xml] de la couche [service] est celui qui a été décrit au paragraphe 2.5.2, page 32.

Le test JUnit [TestServiceFirebird] est le suivant :

```
1.  package istia.st.springmvc.personnes.tests;
2.
3.  ...
4.
5.  public class TestServiceFirebird extends TestCase {
6.
7.    // couche [service]
8.    private IService service;
9.
10.   public IService getService() {
11.     return service;
12.   }
13.
14.   public void setService(IService service) {
15.     this.service = service;
16.   }
17.
18.   // setup
19.   public void setUp() {
20.     service = (IService) (new XmlBeanFactory(new ClassPathResource(
21.         "spring-config-test-service-firebird.xml"))).getBean("service");
22.   }
23.
24.   // liste des personnes
25.   private void doListe(Collection personnes) {
26. ...
27.   }
28.
29.   // test1
30.   public void test1() throws ParseException {
31. ...
32.   }
33.
34.   // modification-suppression d'un élément inexistant
35.   public void test2() throws ParseException {
36. ...
37.   }
38.
39.   // gestion des versions de personne
40.   public void test3() throws ParseException, InterruptedException {
41. ...
42.   }
43.
44.   // optimistic locking - accès multi-threads
45.   public void test4() throws Exception {
46. ...
47.   }
48.
49.   // tests de validité de saveOne
50.   public void test5() throws ParseException {
51. ...
52.   }
53.
54.     // insertions multi-threads
55.   public void test6() throws ParseException, InterruptedException{
56. ...
57.   }
58.
59.   // tests de la méthode deleteMany
60.   public void test7() throws ParseException {
61.     // liste actuelle
62.     Collection personnes = service.getAll();
63.     int nbPersonnes1 = personnes.size();
64.     // affichage
65.     doListe(personnes);
66.     // création de trois personnes
67.     Personne p1 = new Personne(-1, "X", "X", new SimpleDateFormat(
68.         "dd/MM/yyyy").parse("01/02/2006"), true, 1);
69.     Personne p2 = new Personne(-1, "Y", "Y", new SimpleDateFormat(
70.         "dd/MM/yyyy").parse("01/03/2006"), false, 0);
71.     Personne p3 = new Personne(-2, "Z", "Z", new SimpleDateFormat(
72.         "dd/MM/yyyy").parse("01/04/2006"), true, 2);
73.     // ajout des 3 personnes - la personne p3 avec l'id -2  va provoquer
74.     // une exception
75.     boolean erreur = false;
76.     try {
77.       service.saveMany(new Personne[] { p1, p2, p3 });
78.     } catch (Exception ex) {
79.       erreur = true;
```

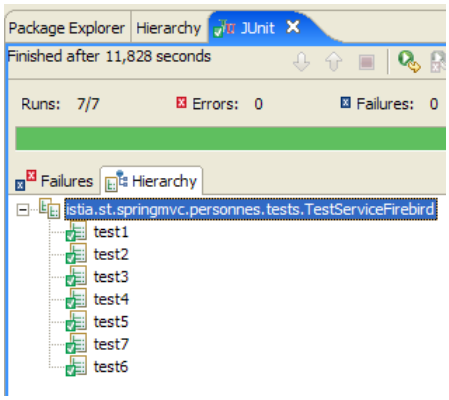
```
80.      System.out.println(ex.toString());
81.    }
82.    // vérification
83.    assertTrue(erreur);
84.    // nouvelle liste - le nombre d'éléments n'a pas du changer
85.    // à cause rollback automatique de la transaction
86.    int nbPersonnes2 = service.getAll().size();
87.    assertEquals(nbPersonnes1, nbPersonnes2);
88.    // ajout des deux personnes valides
89.    // on remet leur id à -1
90.    p1.setId(-1);
91.    p2.setId(-1);
92.    service.saveMany(new Personne[] { p1, p2 });
93.    // on récupère leurs id
94.    int id1 = p1.getId();
95.    int id2 = p2.getId();
96.    // vérifications
97.    p1 = service.getOne(id1);
98.    assertEquals(p1.getNom(), "X");
99.    p2 = service.getOne(id2);
100.   assertEquals(p2.getNom(), "Y");
101.   // nouvelle liste - on doit avoir 2 éléments de +
102.   int nbPersonnes3 = service.getAll().size();
103.   assertEquals(nbPersonnes1 + 2, nbPersonnes3);
104.   // suppression de p1 et p2 et d'une personne inexistante
105.   // une exception doit se produire
106.   erreur = false;
107.   try {
108.     service.deleteMany(new int[] { id1, id2, -1 });
109.   } catch (Exception ex) {
110.     erreur = true;
111.     System.out.println(ex.toString());
112.   }
113.   // vérification
114.   assertTrue(erreur);
115.   // nouvelle liste
116.   personnes = service.getAll();
117.   int nbPersonnes4 = personnes.size();
118.   // aucune personne n'a du être supprimée (rollback
119.   // automatique de la transaction)
120.   assertEquals(nbPersonnes4, nbPersonnes3);
121.   // on supprime les deux personnes valides
122.   service.deleteMany(new int[] { id1, id2 });
123.   // vérifications
124.   // personne p1
125.   erreur = false;
126.   int codeErreur = 0;
127.   try {
128.     p1 = service.getOne(id1);
129.   } catch (DaoException ex) {
130.     erreur = true;
131.     codeErreur = ex.getCode();
132.   }
133.   // on doit avoir une erreur de code 2
134.   assertTrue(erreur);
135.   assertEquals(2, codeErreur);
136.   // personne p2
137.   erreur = false;
138.   codeErreur = 0;
139.   try {
140.     p1 = service.getOne(id2);
141.   } catch (DaoException ex) {
142.     erreur = true;
143.     codeErreur = ex.getCode();
144.   }
145.   // on doit avoir une erreur de code 2
146.   assertTrue(erreur);
147.   assertEquals(2, codeErreur);
148.   // nouvelle liste
149.   personnes = service.getAll();
150.   int nbPersonnes5 = personnes.size();
151.   // vérification - on doit être revenu au point de départ
152.   assertEquals(nbPersonnes5, nbPersonnes1);
153.   // affichage
154.   doListe(personnes);
155. }
156.
157.}
```

- lignes 19-22 : le programme teste des couches [dao] et [service] configurées par le fichier [spring-config-test-service-firebird.xml], celui étudié dans la section précédente.
- les tests [test1] à [test6] sont identiques dans leur esprit à leurs homologues de même nom dans la classe de test [TestDaoFirebird] de la couche [dao]. La seule différence est que par configuration, les méthodes [saveOne] et [deleteOne] s'exécutent désormais dans une transaction.

- la méthode [test7] a pour but de tester les méthodes [saveMany] et [deleteMany]. On veut vérifier qu'elles s'exécutent bien dans une transaction. Commentons le code de cette méthode :

- lignes 62-63 : on compte le nombre de personnes [nbPersonnes1] actuellement dans la liste
- lignes 67-72 : on crée trois personnes
- lignes 73-83 : ces trois personnes sont sauvegardées par la méthode [saveMany] – ligne 77. Les deux premières personnes p1 et p2 ayant un id égal à -1 vont être ajoutées à la table [PERSONNES]. La personne p3 a elle un id égal à -2. Il ne s'agit donc pas d'une insertion mais d'une mise à jour. Celle-ci va échouer car il n'y a aucune personne avec un id égal à – 2 dans la table [PERSONNES]. La couche [dao] va donc lancer une exception qui va remonter jusqu'à la couche [service]. L'existence de cette exception est testée ligne 83.
- à cause de l'exception précédente, la couche [service] devrait faire un [rollback] de l'ensemble des ordres SQL émis pendant l'exécution de la méthode [saveMany], ceci parce que cette méthode s'exécute dans une transaction. Lignes 86-87, on vérifie que le nombre de personnes de la liste n'a pas bougé et que donc les insertions de p1 et p2 n'ont pas eu lieu.
- lignes 88-103 : on ajoute les seules personnes p1 et p2 et on vérifie qu'ensuite on a deux personnes de plus dans la liste.
- lignes 106-114 : on supprime un groupe de personnes constitué des personnes p1 et p2 qu'on vient d'ajouter et d'une personne inexistante (id= -1). La méthode [deleteMany] est utilisée pour cela, ligne 108. Cette méthode va échouer car il n'y a aucune personne avec un id égal à – 1 dans la table [PERSONNES]. La couche [dao] va donc lancer une exception qui va remonter jusqu'à la couche [service]. L'existence de cette exception est testée ligne 114.
- à cause de l'exception précédente, la couche [service] devrait faire un [rollback] de l'ensemble des ordres SQL émis pendant l'exécution de la méthode [deleteMany], ceci parce que cette méthode s'exécute dans une transaction. Lignes 116-117, on vérifie que le nombre de personnes de la liste n'a pas bougé et que donc les suppressions de p1 et p2 n'ont pas eu lieu.
- ligne 122 : on supprime un groupe constitué des seules personnes p1 et p2. Cela devrait réussir. Le reste de la méthode vérifie que c'est bien le cas.

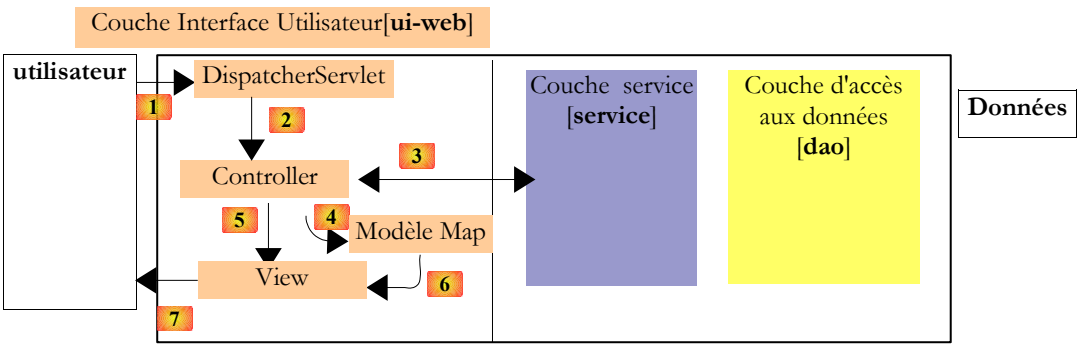
L'exécution des tests donne les résultats suivants :



Les sept tests ont été réussis. Nous considèrerons notre couche [service] comme opérationnelle.

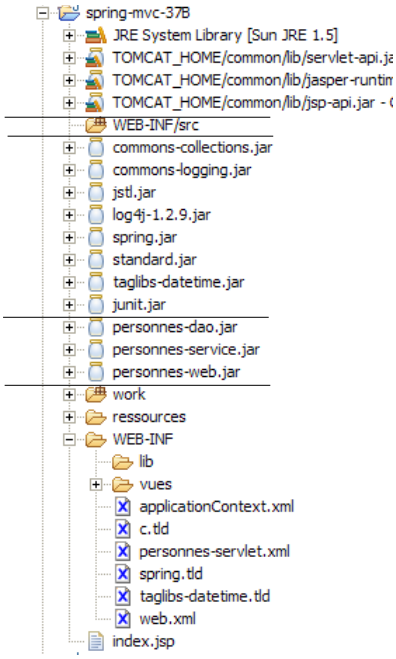
2.7 La couche [web]

Rappelons l'architecture générale de l'application web à construire :

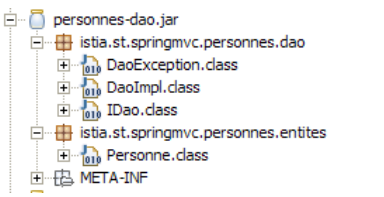


Nous venons de construire les couches [dao] et [service] permettant de travailler avec une base de données Firebird. Nous avons écrit une version 1 de cette application où les couches [dao] et [service] travaillaient avec une liste de personnes en mémoire. La couche [web] écrite à cette occasion **reste valide**. En effet, elle s'adressait à une couche [service] implémentant l'interface [IService]. La nouvelle couche [service] implémentant cette même interface, la couche [web] **n'a pas à être modifiée**.

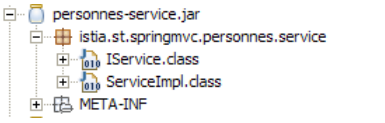
Dans le précédent article, la version 1 de l'application avait été testée avec le projet Eclipse [spring-mvc-37B] où les couches [web, service, dao] avaient été mises dans des archives .jar :



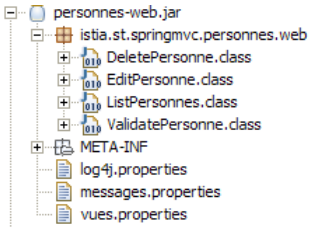
Le dossier [WEB-INF/src] était vide. Les classes des trois couches étaient dans les trois archives [personnes-*.jar] :



archive de la couche [dao]

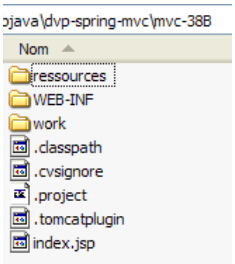


archive de la couche [service]



archive de la couche [web]

Pour tester la version 2, nous dupliquons le dossier [mvc-37B] du projet [spring-mvc-37B] en [mvc-38B] :



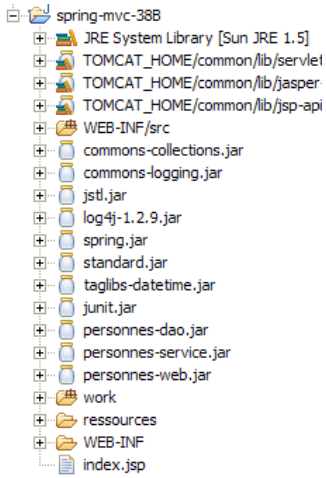
Nous modifions le fichier [.project] afin qu'il référence un projet Eclipse appelé [spring-mvc-38B] :

```
1.  <projectDescription>
2.    <name>spring-mvc-38B</name>
3.    <comment></comment>
4.  ...
```

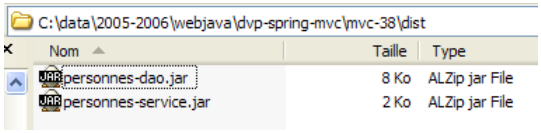
Nous faisons de même avec le fichier [.tomcatplugin] pour référencer une application web de contexte [/spring-mvc-38B] :

```
1.      <extraInfo></extraInfo>
2.      <webPath>/spring-mvc-38B</webPath>
3.  </tomcatProjectProperties>
```

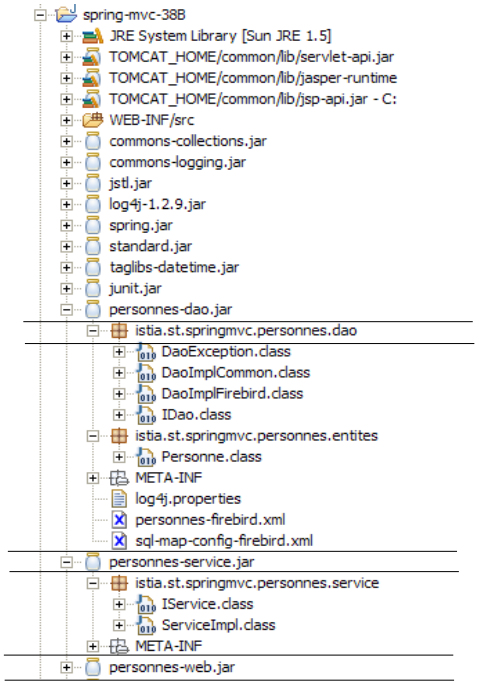
Ensuite nous importons ce projet sous Eclipse [File / Import / Existing projects] :



Dans le projet [spring-mvc-38], nous exportons [File / Export / Jar file] les couches [dao] et [service] respectivement dans les archives [personnes-dao.jar] et [personnes-service.jar] du dossier [dist] du projet :

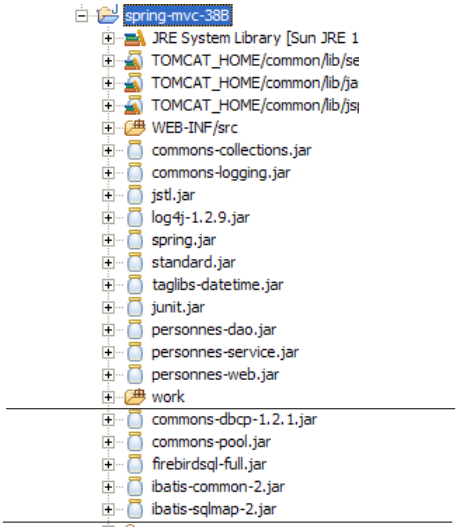


Nous copions ces deux fichiers, puis **sous Eclipse** nous les collons dans le dossier [WEB-INF/lib] où ils vont remplacer les archives de même nom de la version précédente.



En procédant de la même façon, nous copions les archives [commons-dbcp-*.jar, commons-pool-*.jar, firebirdsql-full.jar, ibatis-common-2.jar, ibatis-sqlmap-2.jar] du dossier [lib] du projet [spring-mvc-38] dans le dossier [WEB-INF/lib] du projet [spring-mvc-38B]. Ces archives sont nécessaires aux nouvelles couches [dao] et [service].

Ceci fait, nous incluons les nouvelles archives dans le Classpath du projet : [clic droit sur projet -> Properties -> Java Build Path -> Add Jars].



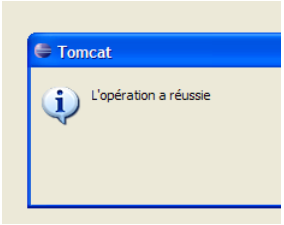
Le fichier [applicationContext.xml] configure les couches [dao] et [service] de l'application web. Dans la nouvelle version, il est identique au fichier [spring-config-test-service-firebird.xml] qui a servi pour configurer le test de la couche service dans le projet [spring-mvc-38]. On fait donc un copier / coller de l'un vers l'autre :

```
1.  <?xml version="1.0" encoding="ISO_8859-1"?>
2.  <!DOCTYPE beans SYSTEM "http://www.springframework.org/dtd/spring-beans.dtd">
3.  <beans>
4.    <!-- la source de donnéees DBCP -->
5.    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
6.      destroy-method="close">
7.      <property name="driverClassName">
8.        <value>org.firebirdsql.jdbc.FBDriver</value>
9.      </property>
10.     <property name="url">
11.       <!-- attention : ne pas laisser d'espaces entre les deux balises <value> -->
12.       <value>jdbc:firebirdsql:localhost/3050:C:/data/2005-2006/webjava/dvp-spring-mvc/mvc-
    38/database/dbpersonnes.gdb</value>
13.     </property>
14.     <property name="username">
15.       <value>sysdba</value>
16.     </property>
17.     <property name="password">
18.       <value>masterkey</value>
19.     </property>
20.   </bean>
21.   <!-- SqlMapCllient -->
22.   <bean id="sqlMapClient"
23.     class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
24.     <property name="dataSource">
25.       <ref local="dataSource"/>
26.     </property>
27.     <property name="configLocation">
28.       <value>classpath:sql-map-config-firebird.xml</value>
29.     </property>
30.   </bean>
31.   <!-- la classes d'accès à la couche [dao] -->
32.   <bean id="dao" class="istia.st.springmvc.personnes.dao.DaoImplFirebird">
33.     <property name="sqlMapClient">
34.       <ref local="sqlMapClient"/>
35.     </property>
36.   </bean>
37.   <!-- gestionnaire de transactions -->
38.   <bean id="transactionManager"
39.     class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
40.     <property name="dataSource">
41.       <ref local="dataSource"/>
42.     </property>
43.   </bean>
44.   <!-- la classes d'accès à la couche [service] -->
45.   <bean id="service"
46.     class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
47.     <property name="transactionManager">
48.       <ref local="transactionManager"/>
49.     </property>
50.     <property name="target">
51.       <bean class="istia.st.springmvc.personnes.service.ServiceImpl">
52.         <property name="dao">
53.           <ref local="dao"/>
54.         </property>
```

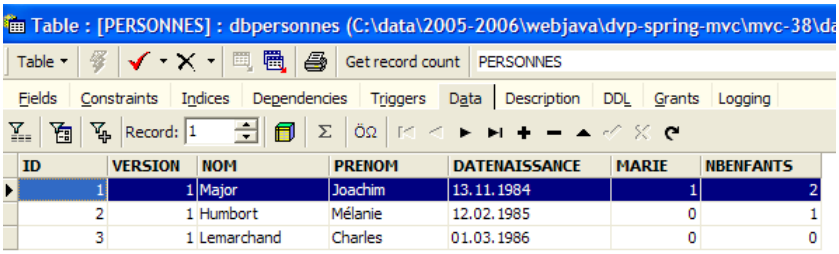
```
55.       </bean>
56.     </property>
57.     <property name="transactionAttributes">
58.       <props>
59.         <prop key="get*">PROPAGATION_SUPPORTS,readOnly</prop>
60.         <prop key="save*">PROPAGATION_REQUIRED</prop>
61.         <prop key="delete*">PROPAGATION_REQUIRED</prop>
62.       </props>
63.     </property>
64.   </bean>
65. </beans>
```

- ligne 12 : l'url de la base de données Firebird. Nous continuons à utiliser la base qui a servi aux tests des couches [dao] et [service]

Nous déclarons le contexte de la nouvelle application web [clic droit sur projet -> Projet Tomcat -> Mise à jour du contexte] :

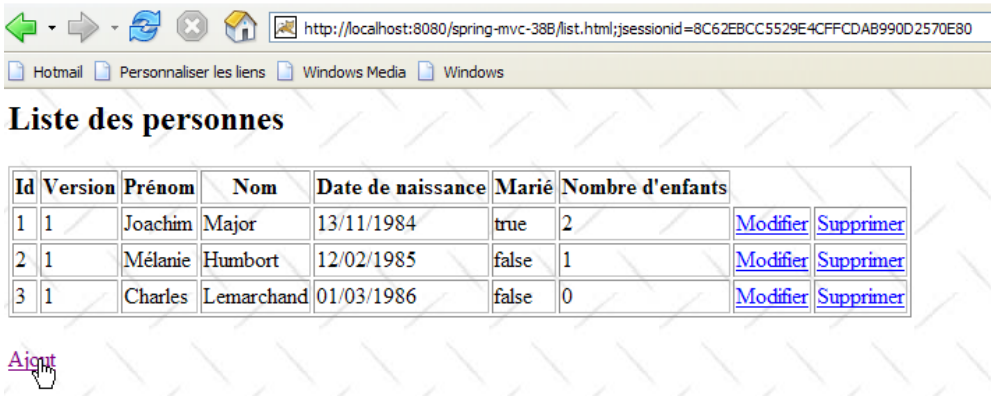


Nous sommes prêts pour les tests. Le SGBD Firebird est lancé. Le contenu de la table [PERSONNES] est alors le suivant :

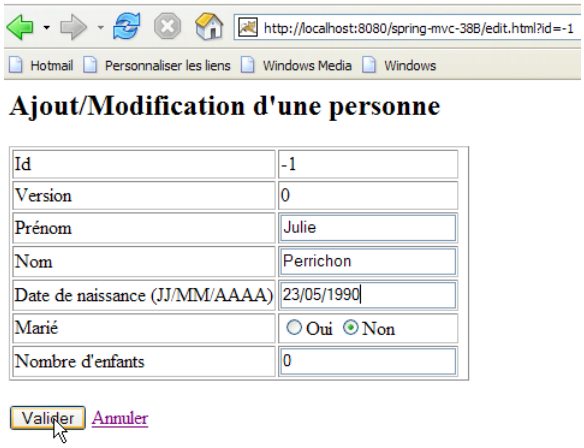


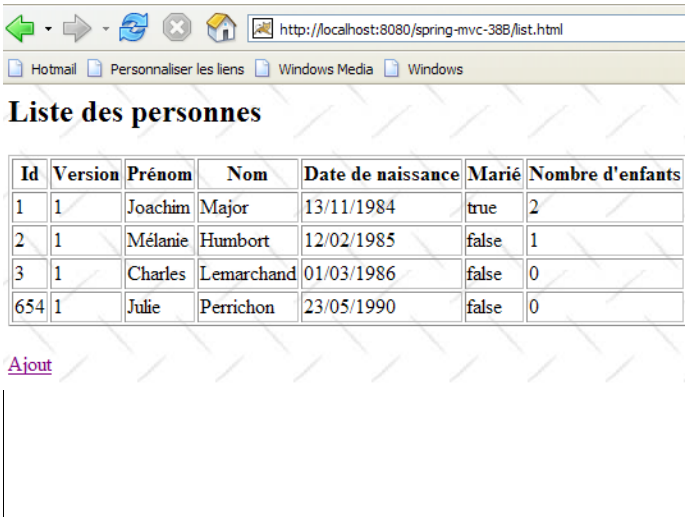
ID	VERSION	NOM	PRENOM	DATENAISSANCE	MARIE	NBENFANTS
1	1	Major	Joachim	13.11.1984	1	2
2	1	Humbort	Mélanie	12.02.1985	0	1
3	1	Lemarchand	Charles	01.03.1986	0	0

Tomcat est lancé à son tour. Avec un navigateur, nous demandons l'url [http://localhost:8080/spring-mvc-38B] :

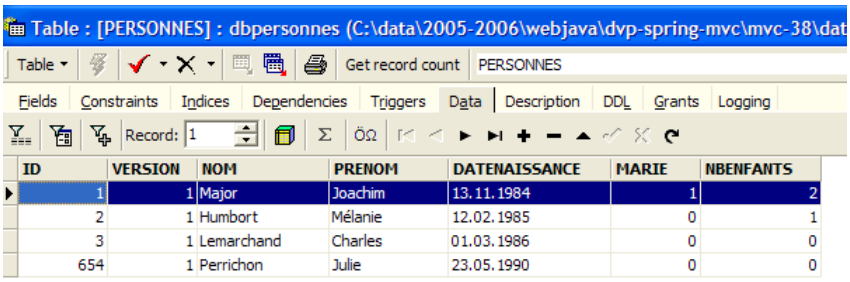


Nous ajoutons une nouvelle personne avec le lien [Ajout] :



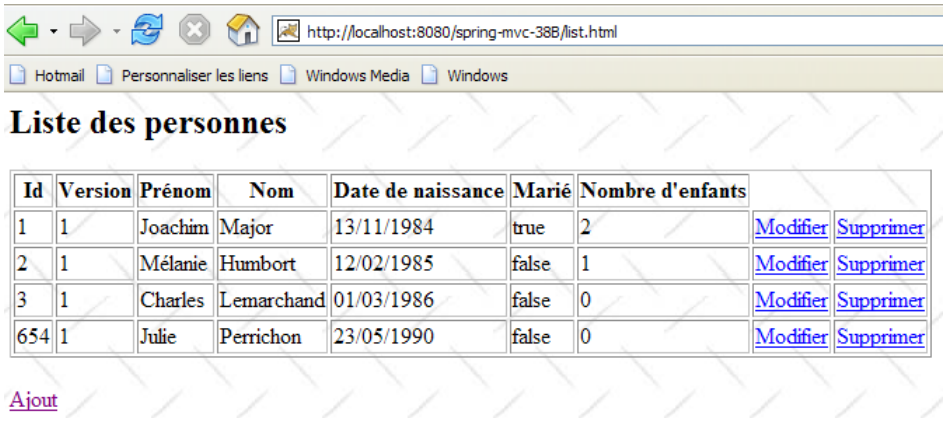


Nous vérifions l'ajout dans la base de données :

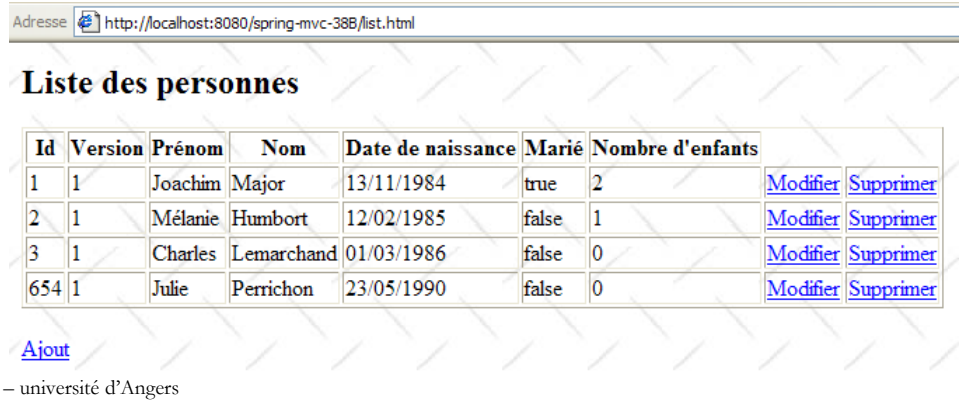


Le lecteur est invité à faire d'autres tests [modification, suppression].

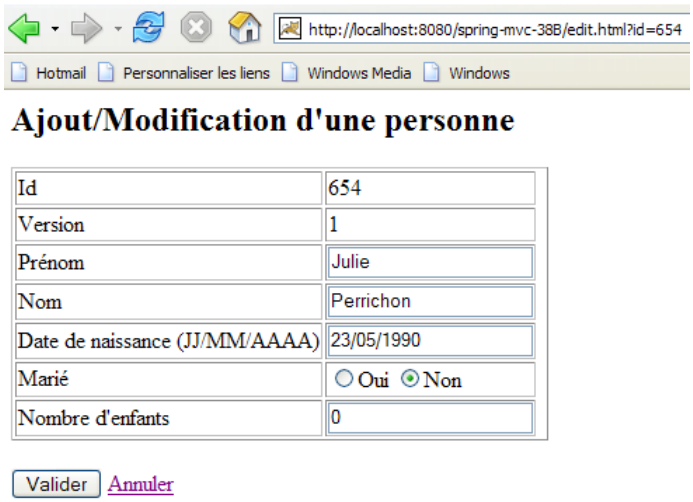
Faisons maintenant le test de conflits de version qui avait été fait dans la version 1. [Firefox] sera le navigateur de l'utilisateur U1. Celui-ci demande l'url [http://localhost:8080/spring-mvc-38B/list.html] :



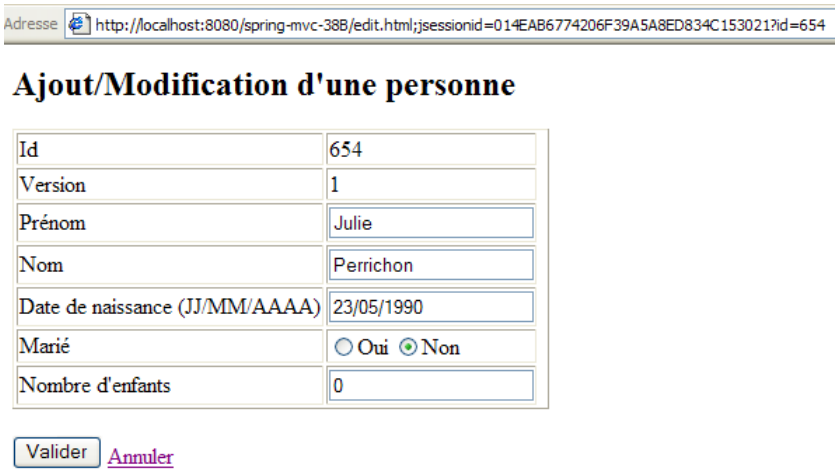
[IE] sera le navigateur de l'utilisateur U2. Celui-ci demande la même Url :



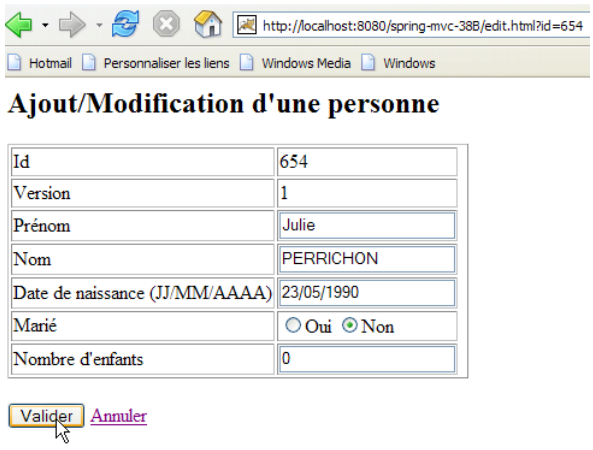
L'utilisateur U1 entre en modification de la personne [Perrichon] :



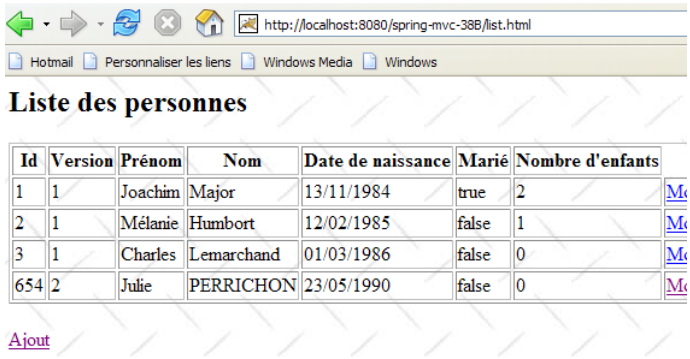
L'utilisateur U2 fait de même :



L'utilisateur U1 fait des modifications et valide :

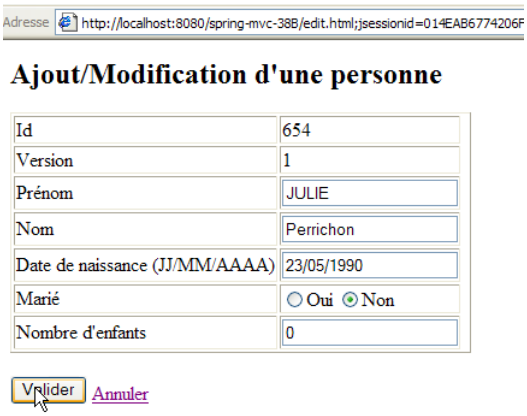


validation

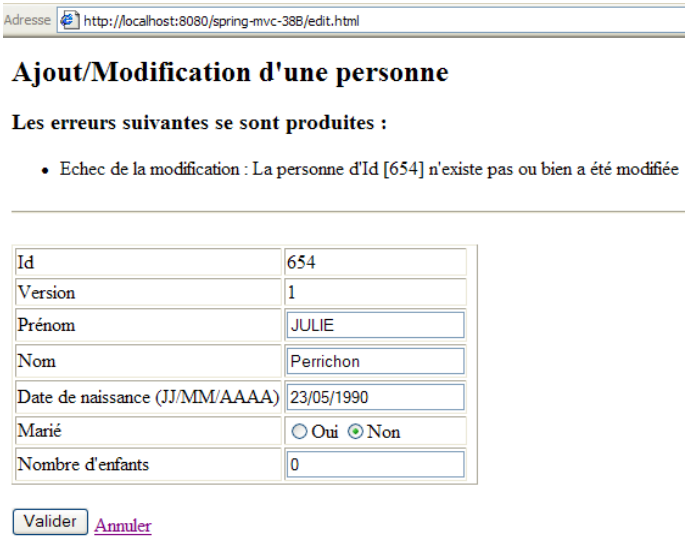


réponse du serveur

L'utilisateur U2 fait de même :

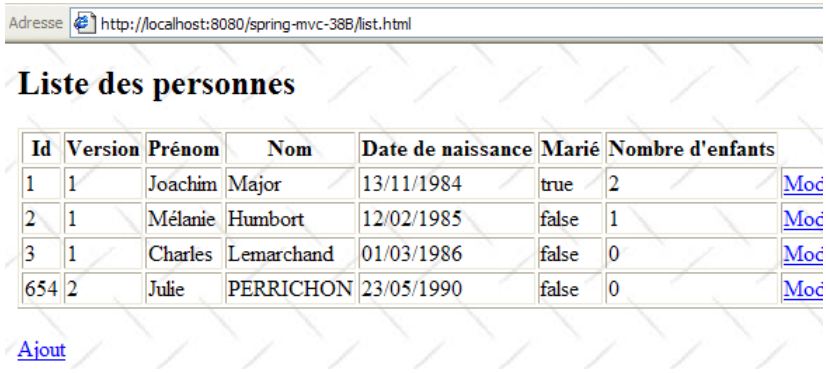


validation



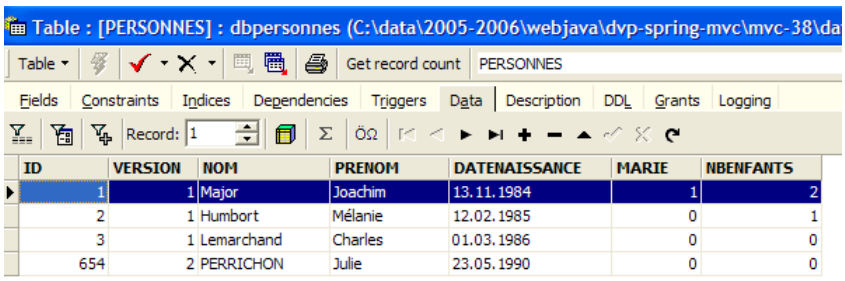
le conflit de version a été détecté

L'utilisateur U2 revient à la liste des personnes avec le lien [Annuler] du formulaire :



Il trouve la personne [Perrichon] telle que U1 l'a modifiée (nom passé en majuscules).

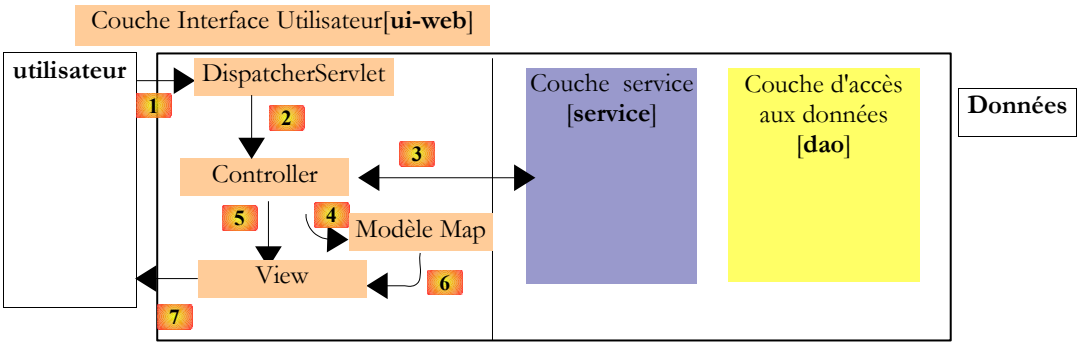
Et la base de données dans tout ça ? Regardons :



La personne n° 654 a bien son nom en majuscules suite à la modification faite par U1.

2.8 Conclusion

Rappelons ce que nous voulions faire. Nous avions une application web avec l'architecture 3tier suivante :



où les couches [dao] et [service] travaillaient avec une liste de données en mémoire qui était donc perdue lorsque le serveur web était arrêté. C'était la version 1. Dans la version 2, les couches [service] et [dao] ont été réécrites pour que la liste de personnes soit dans une table de base de données. Elle est donc désormais persistante. On se propose maintenant de voir l'impact qu'a sur notre application le changement de SGBD. Pour cela, nous allons construire trois nouvelles versions de notre application web :

- version 3 : le SGBD est Postgres
- version 4 : le SGBD est MySQL
- version 5 : le SGBD est SQL Server Express 2005

Les changements se font aux endroits suivants :

- la classe [DaoImplFirebird] implémente des fonctionnalités de la couche [dao] liées au SGBD Firebird. Si ce besoin persiste, elle sera remplacée respectivement par les classes [DaoImplPostgres], [DaoImplMySQL] et [DaoImplSqlExpress].
- le fichier de mapping [personnes-firebird.xml] d'iBATIS pour le SGBD Firebird va être remplacé respectivement par les fichiers de mapping [personnes-postgres.xml], [personnes-mysql.xml] et [personnes-sqlexpress.xml].
- la configuration de l'objet [DataSource] de la couche [dao] est spécifique à un SGBD. Elle va donc changer à chaque version.
- le pilote JDBC du SGBD change également à chaque version

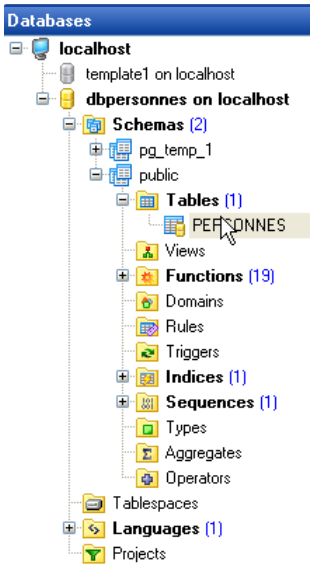
En-dehors de ces points, tout reste à l'identique. Dans la suite, nous décrivons ces nouvelles versions en ne nous attachant qu'aux seules nouveautés amenées par chacune d'elles.

3 Spring MVC dans une architecture 3tier – Exemple 4 - Postgres

3.1 La base de données Postgres

Dans cette version, nous allons installer la liste des personnes dans une table de base de données Postgres 8.x [http://www.postgres.org]. Dans ce qui suit, les copies d'écran proviennent du client EMS PostgreSQL Manager Lite [http://www.sqlmanager.net/fr/products/postgresql/manager], un client d'administration gratuit du SGBD Postgres.

La base de données s'appelle [dbpersonnes]. Elle contient une table [PERSONNES] :

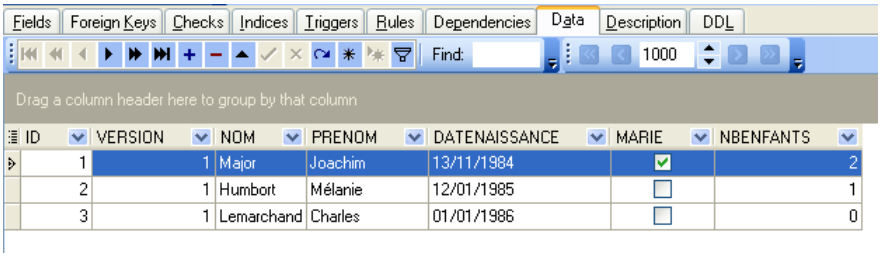


La table [PERSONNES] contiendra la liste des personnes gérée par l'application web. Elle a été construite avec les ordres SQL suivants :

```
1.  CREATE TABLE "public"."PERSONNES" (
2.    "ID" INTEGER NOT NULL,
3.    "VERSION" INTEGER NOT NULL,
4.    "NOM" VARCHAR(30) NOT NULL,
5.    "PRENOM" VARCHAR(30) NOT NULL,
6.    "DATENAISSANCE" DATE NOT NULL,
7.    "MARIE" BOOLEAN NOT NULL,
8.    "NBENFANTS" INTEGER NOT NULL,
9.    CONSTRAINT "PERSONNES_pkey" PRIMARY KEY("ID"),
10.   CONSTRAINT "PERSONNES_chk_NBENFANTS" CHECK ("NBENFANTS" >= 0),
11.   CONSTRAINT "PERSONNES_chk_NOM" CHECK (("NOM")::text <> ''::text),
12.   CONSTRAINT "PERSONNES_chk_PRENOM" CHECK (("PRENOM")::text <> ''::text)
13. ) WITH OIDS;
```

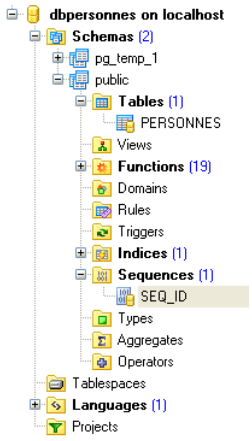
Nous n'insistons pas sur cette table qui est analogue à la table [PERSONNES] de type Firebird étudiée précédemment. On notera cependant que les noms des colonnes et des tables sont entre guillemets. Par ailleurs, ces noms sont sensibles à la casse. Il est possible que ce mode de fonctionnement de Postgres 8.x soit configurable. Je n'ai pas creusé la question.

La table [PERSONNES] pourrait avoir le contenu suivant :

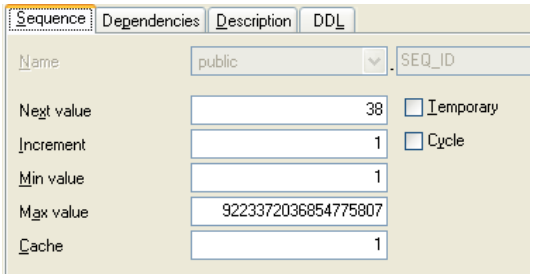


ID	VERSION	NOM	PRENOM	DATENAISSANCE	MARIE	NBENFANTS
1	1	Major	Joachim	13/11/1984	☑	2
2	1	Humbort	Mélanie	12/01/1985	☐	1
3	1	Lemarchand	Charles	01/01/1986	☐	0

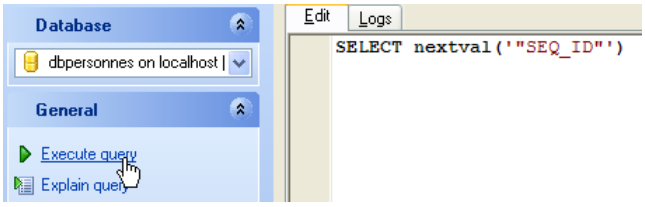
Outre la table [PERSONNES], la base [dbpersonnes] a, un objet appelé séquence et nommé [SEQ_ID]. Ce générateur délivre des nombres entiers successifs que nous utiliserons pour donner sa valeur à la clé primaire [ID] de la classe [PERSONNES]. Prenons un exemple pour illustrer son fonctionnement :



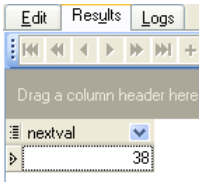
- double-cliquer sur [SEQ_ID]



- la prochaine valeur du générateur est 38

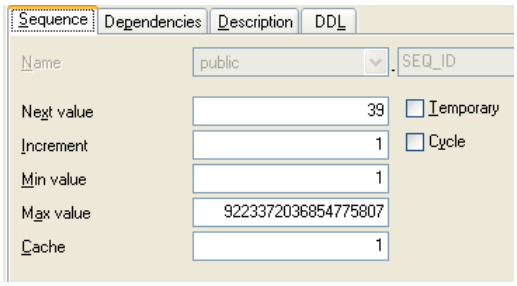


- émettons l'ordre SQL ci-dessus (F12) ->



- la valeur obtenue est la valeur [Next value] de la séquence [SEQ_ID]

On peut constater que la valeur [Next value] de la séquence [SEQ_ID] a changé (double-clic dessus + F5 pour rafraîchir) :



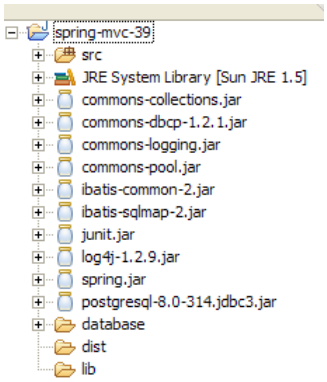
L'ordre SQL

```
SELECT nextval('"SEQ_ID"')
```

permet donc d'avoir la valeur suivante de la séquence [SEQ_ID]. Nous l'utiliserons dans le fichier [personnes-postgres.xml] qui rassemble les ordres SQL émis sur le SGBD.

3.2 Le projet Eclipse des couches [dao] et [service]

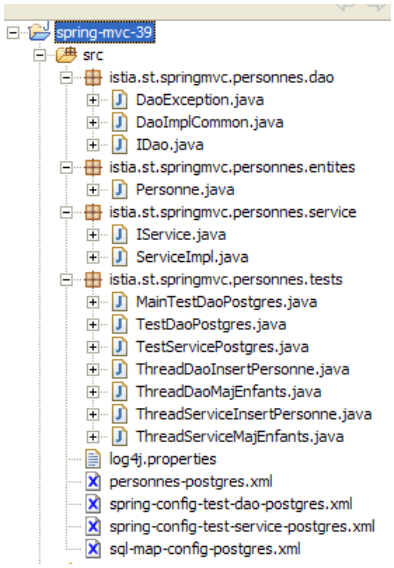
Pour développer les couches [dao] et [service] de notre application avec la base de données Postgres 8.x, nous utiliserons le projet Eclipse [spring-mvc-39] suivant :



Le projet est un simple projet Java, pas un projet web Tomcat.

Dossier [src]

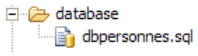
Ce dossier contient les codes source des couches [dao] et [service] :



Tous les fichiers ayant [postgres] dans leur nom ont pu subir ou non une modfication vis à vis de la version Firebird.Dans ce qui suit, nous décrivons les fichiers modifiés.

Dossier [database]

Ce dossier contient le script de création de la base de données Postgres des personnes :

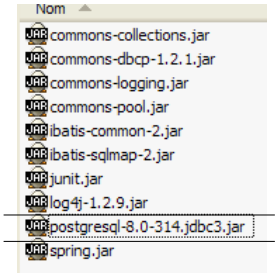



```
1.  -- EMS PostgreSQL Manager Lite 3.1.0.1
2.  -- ---------------------------------------
3.  -- Host     : localhost
4.  -- Database : dbpersonnes
5.
6.
7.
8.  --
9.  -- Definition for function plpgsql_call_handler (OID = 17230) :
10. --
11. ...
12. --
13. -- Structure for table PERSONNES (OID = 17254) :
14. --
15. CREATE TABLE "PERSONNES" (
16.     "ID" integer NOT NULL,
17.     "VERSION" integer NOT NULL,
18.     "NOM" varchar(30) NOT NULL,
19.     "PRENOM" varchar(30) NOT NULL,
20.     "DATENAISSANCE" date NOT NULL,
21.     "MARIE" boolean NOT NULL,
```

```
22.     "NBENFANTS" integer NOT NULL,
23.     CONSTRAINT "PERSONNES_chk_NBENFANTS" CHECK (("NBENFANTS" >= 0)),
24.     CONSTRAINT "PERSONNES_chk_NOM" CHECK ((("NOM")::text <> ''::text)),
25.     CONSTRAINT "PERSONNES_chk_PRENOM" CHECK ((("PRENOM")::text <> ''::text))
26. );
27. --
28. -- Definition for sequence SEQ_ID (OID = 17261) :
29. --
30. CREATE SEQUENCE "SEQ_ID"
31.     INCREMENT BY 1
32.     NO MAXVALUE
33.     NO MINVALUE
34.     CACHE 1;
35. --
36. -- Data for blobs (OID = 17254) (LIMIT 0,3)
37. --
38. INSERT INTO "PERSONNES" ("ID", "VERSION", "NOM", "PRENOM", "DATENAISSANCE", "MARIE", "NBENFANTS")
    VALUES (1, 1, 'Major', 'Joachim', '1984-11-13', true, 2);
39. INSERT INTO "PERSONNES" ("ID", "VERSION", "NOM", "PRENOM", "DATENAISSANCE", "MARIE", "NBENFANTS")
    VALUES (2, 1, 'Humbort', 'Mélanie', '1985-01-12', false, 1);
40. INSERT INTO "PERSONNES" ("ID", "VERSION", "NOM", "PRENOM", "DATENAISSANCE", "MARIE", "NBENFANTS")
    VALUES (3, 1, 'Lemarchand', 'Charles', '1986-01-01', false, 0);
41. --
42. -- Definition for index PERSONNES_pkey (OID = 17256) :
43. --
44. ALTER TABLE ONLY "PERSONNES"
45.     ADD CONSTRAINT "PERSONNES_pkey" PRIMARY KEY ("ID");
46. --
47. -- Data for sequence public."SEQ_ID" (OID = 17261)
48. --
49. SELECT pg_catalog.setval('"SEQ_ID"', 37, true);
50. --
51. -- Comments
52. --
53. COMMENT ON SCHEMA public IS 'Standard public schema';
```

Dossier [lib]

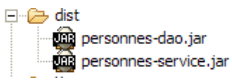
Ce dossier contient les archives nécessaires à l'application :



On notera la présence du pilote jdbc du SGBD Postgres 8.x. Toutes ces archives font partie du *Classpath* du projet Eclipse.

Dossier [dist]

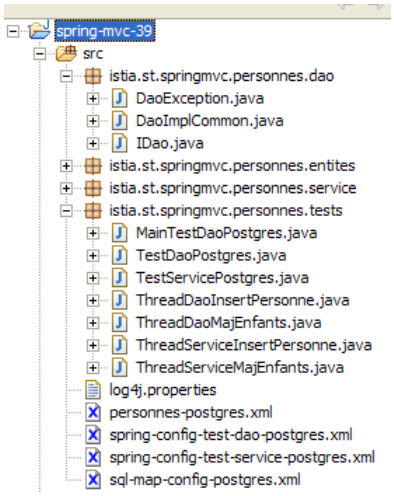
Ce dossier contient les archives issues de la compilation des classes de l'application :



- [personnes-dao.jar] : archive de la couche [dao]
- [personnes-service.jar] : archive de la couche [service]

3.3 La couche [dao]

La couche [dao] est la suivante :



Nous ne présentons que ce qui change vis à vis de la version [Firebird].

Le fichier de mapping [personne-postgres.xml] est le suivant :

```
1.  <?xml version="1.0" encoding="UTF-8" ?>
2.
3.  <!DOCTYPE sqlMap
4.      PUBLIC "-//iBATIS.com//DTD SQL Map 2.0//EN"
5.      "http://www.ibatis.com/dtd/sql-map-2.dtd">
6.
7.  <!-- attention - Postgresql 8 demande l'orthographe exacte des noms de colonnes
8.    et des tables ainsi que des guillemets autour de ces noms -->
9.
10.  <sqlMap>
11.    <!-- alias classe [Personne] -->
12.    <typeAlias alias="Personne.classe"
13.      type="istia.st.springmvc.personnes.entites.Personne"/>
14.    <!-- mapping table [PERSONNES] - objet [Personne] -->
15.    <resultMap id="Personne.map"
16.      class="istia.st.springmvc.personnes.entites.Personne">
17.      <result property="id" column="ID" />
18.      <result property="version" column="VERSION" />
19.      <result property="nom" column="NOM"/>
20.      <result property="prenom" column="PRENOM"/>
21.      <result property="dateNaissance" column="DATENAISSANCE"/>
22.      <result property="marie" column="MARIE"/>
23.      <result property="nbEnfants" column="NBENFANTS"/>
24.    </resultMap>
25.    <!-- liste de toutes les personnes -->
26.    <select id="Personne.getAll" resultMap="Personne.map" > select "ID",
27.      "VERSION", "NOM", "PRENOM", "DATENAISSANCE", "MARIE", "NBENFANTS" FROM
28.      "PERSONNES"</select>
29.    <!-- obtenir une personne en particulier -->
30.    <select id="Personne.getOne" resultMap="Personne.map" >select "ID",
31.      "VERSION", "NOM", "PRENOM", "DATENAISSANCE", "MARIE", "NBENFANTS" FROM
32.      "PERSONNES" WHERE "ID"=#value#</select>
33.    <!-- ajouter une personne -->
34.    <insert id="Personne.insertOne" parameterClass="Personne.classe">
35.      <selectKey keyProperty="id">
36.        SELECT nextval('"SEQ_ID"') as value
37.      </selectKey>
38.      insert into "PERSONNES"("ID", "VERSION",
39.      "NOM", "PRENOM", "DATENAISSANCE", "MARIE", "NBENFANTS") VALUES(#id#,
40.      #version#, #nom#, #prenom#, #dateNaissance#, #marie#, #nbEnfants#)
41.      </insert>
42.    <!-- mettre à jour une personne -->
43.    <update id="Personne.updateOne" parameterClass="Personne.classe"> update
44.      "PERSONNES" set "VERSION"=#version#+1, "NOM"=#nom#, "PRENOM"=#prenom#,
45.      "DATENAISSANCE"=#dateNaissance#, "MARIE"=#marie#, "NBENFANTS"=#nbEnfants#
46.      WHERE "ID"=#id# and "VERSION"=#version#</update>
47.    <!-- supprimer une personne -->
48.    <delete id="Personne.deleteOne" parameterClass="int"> delete FROM
49.      "PERSONNES" WHERE "ID"=#value# </delete>
50.  </sqlMap>
```

C'est le même contenu que [personnes-firebird.xml] aux détails près suivants :

- les noms des colonnes et des tables sont entre guillemets et ces noms sont sensibles à la casse
- l'ordre SQL " Personne.insertOne " a changé lignes 34-41. La façon de générer la clé primaire avec Postgres est différente de celle utilisée avec Firebird :

- ligne 36 : l'ordre SQL [SELECT nextval('"SEQ_ID"')] fournit la clé primaire. La syntaxe [as value] est obligatoire. [value] représente la clé obtenue. Cette valeur va être affectée au champ de l'objet [Personne] désigné par l'attribut [keyProperty] (line 35), ici le champ [id].
- les ordres SQL de la balise <insert> sont exécutés dans l'ordre où ils sont rencontrés. Donc le SELECT est fait avant l'INSERT. Au moment de l'opération d'insertion, le champ [id] de l'objet [Personne] aura donc été mis à jour par l'ordre SQL SELECT.
- lignes 38-40 : insertion de l'objet [Personne]

La classe d'implémentation [DaoImplCommon] de la couche [dao] est celle étudiée dans la version [Firebird].

La configuration de la couche [dao] a été adaptée au SGBD [Postgres]. Ainsi, le fichier de configuration [spring-config-test-dao-postgres.xml] est-il le suivant :

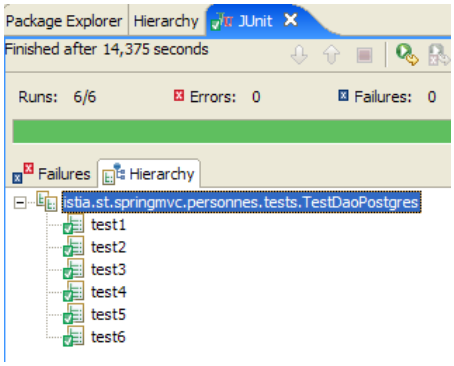
```
1.  <?xml version="1.0" encoding="ISO_8859-1"?>
2.  <!DOCTYPE beans SYSTEM "http://www.springframework.org/dtd/spring-beans.dtd">
3.  <beans>
4.    <!-- la source de donnéees DBCP -->
5.    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
6.      destroy-method="close">
7.      <property name="driverClassName">
8.        <value>org.postgresql.Driver</value>
9.      </property>
10.     <property name="url">
11.       <value>jdbc:postgresql:dbpersonnes</value>
12.     </property>
13.     <property name="username">
14.       <value>postgres</value>
15.     </property>
16.     <property name="password">
17.       <value>postgres</value>
18.     </property>
19.   </bean>
20.   <!-- SqlMapCllient -->
21.   <bean id="sqlMapClient"
22.     class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
23.     <property name="dataSource">
24.       <ref local="dataSource"/>
25.     </property>
26.     <property name="configLocation">
27.       <value>classpath:sql-map-config-postgres.xml</value>
28.     </property>
29.   </bean>
30.   <!-- la classes d'accè à la couche [dao] -->
31.   <bean id="dao" class="istia.st.springmvc.personnes.dao.DaoImplCommon">
32.     <property name="sqlMapClient">
33.       <ref local="sqlMapClient"/>
34.     </property>
35.   </bean>
36. </beans>
```

- lignes 5-19 : le bean [dataSource] désigne maintenant la base [Postgres] [dbpersonnes] dont l'administrateur est [postgres] avec le mot de passe [postgres]. Le lecteur modifiera cette configuration selon son propre environnement.
- ligne 31 : la classe [DaoImplCommon] est la classe d'implémentation de la couche [dao]

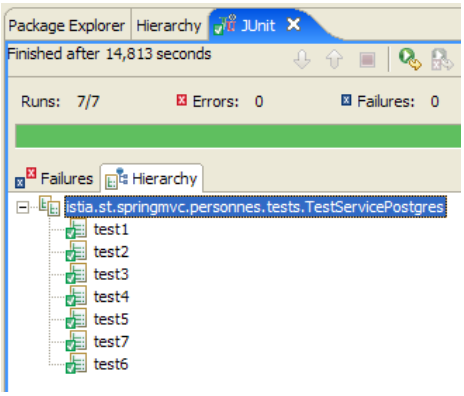
Ces modifications faites, on peut passer aux tests.

3.4 Les tests des couches [dao] et [service]

Les tests des couches [dao] et [service] sont les mêmes que pour la version [Firebird]. Les résultats obtenus sont les suivants :



- couche [dao]



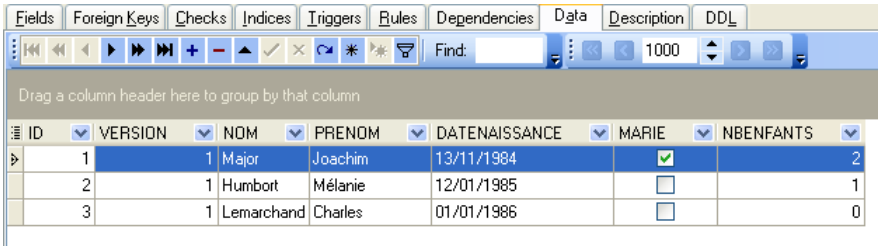
- couche [service]

On constate que les tests ont été passés avec succès avec l'implémentation [DaoImplCommon]. Nous n'aurons pas à dériver cette classe comme il avait été nécessaire de le faire avec le SGBD [Firebird].

3.5 Tests de l'application [web]

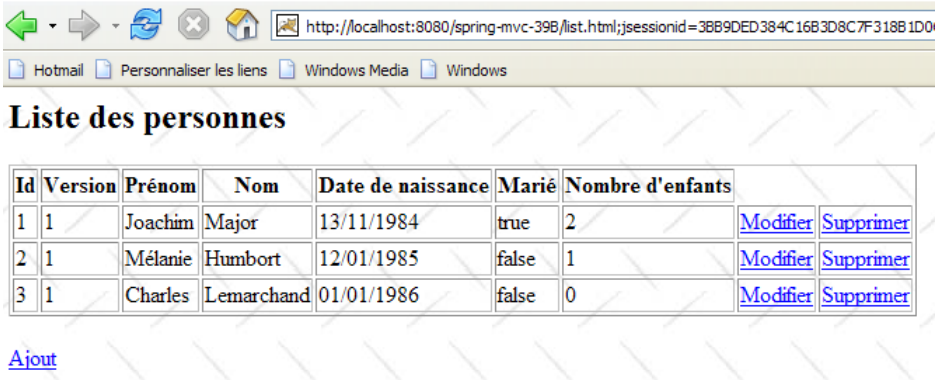
Pour tester l'application web avec le SGBD [Postgres], nous construisons un projet Eclipse [spring-mvc-39B] de façon analogue à celle utilisée pour construire le projet [spring-mvc-38B] avec la base Firebird (cf page 39).

Le SGBD Postgres est lancé. Le contenu de la table [PERSONNES] est alors le suivant :



ID	VERSION	NOM	PRENOM	DATENAISSANCE	MARIE	NBENFANTS
1	1	Major	Joachim	13/11/1984	☑	2
2	1	Humbort	Mélanie	12/01/1985	☐	1
3	1	Lemarchand	Charles	01/01/1986	☐	0

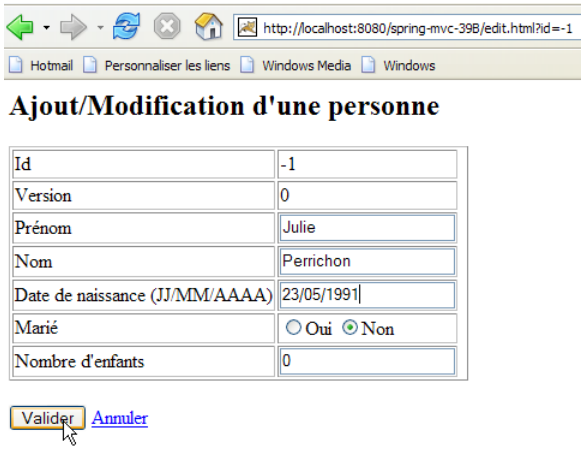
Tomcat est lancé à son tour. Avec un navigateur, nous demandons l'url [http://localhost:8080/spring-mvc-39B] :

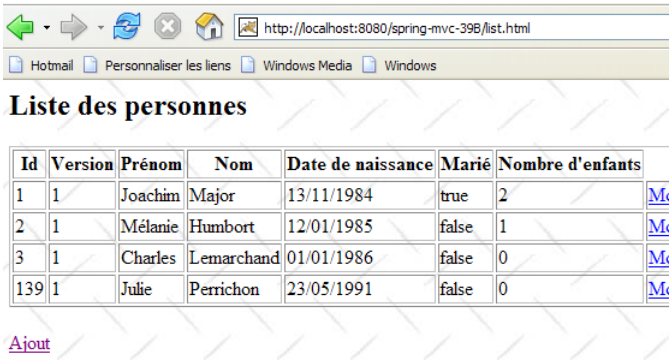


Id	Version	Prénom	Nom	Date de naissance	Marié	Nombre d'enfants		
1	1	Joachim	Major	13/11/1984	true	2	Modifier	Supprimer
2	1	Mélanie	Humbort	12/01/1985	false	1	Modifier	Supprimer
3	1	Charles	Lemarchand	01/01/1986	false	0	Modifier	Supprimer

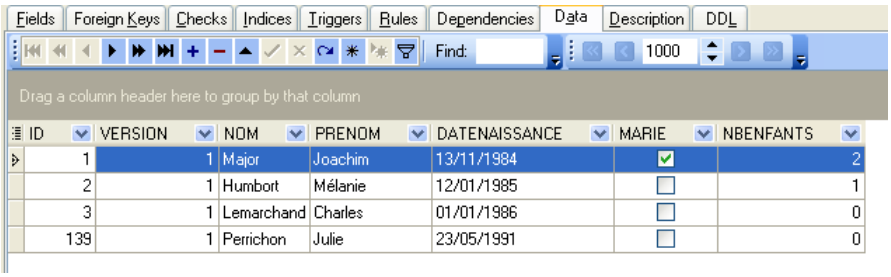
Ajout

Nous ajoutons une nouvelle personne avec le lien [Ajout] :





Nous vérifions l'ajout dans la base de données :



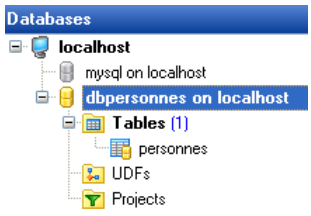
Le lecteur est invité à faire d'autres tests [modification, suppression].

4 Spring MVC dans une architecture 3tier – Exemple 4 - MySQL

4.1 La base de données MySQL

Dans cette version, nous allons installer la liste des personnes dans une table de base de données MySQL 4.x. Nous avons utilisé le paquetage [Apache – MySQL – PHP] disponible à l'url [http://www.easyphp.org]. Dans ce qui suit, les copies d'écran proviennent du client EMS MySQL Manager Lite [http://www.sqlmanager.net/fr/products/mysql/manager], un client d'administration gratuit du SGBD MySQL.

La base de données s'appelle [dbpersonnes]. Elle contient une table [PERSONNES] :



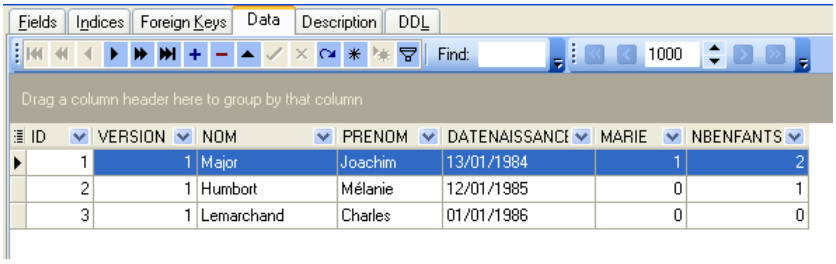
La table [PERSONNES] contiendra la liste des personnes gérée par l'application web. Elle a été construite avec les ordres SQL suivants :

```
1.  CREATE TABLE `personnes` (
2.    `ID` int(11) NOT NULL auto_increment,
3.    `VERSION` int(11) NOT NULL default '0',
4.    `NOM` varchar(30) NOT NULL default '',
5.    `PRENOM` varchar(30) NOT NULL default '',
6.    `DATENAISSANCE` date NOT NULL default '0000-00-00',
7.    `MARIE` tinyint(4) NOT NULL default '0',
8.    `NBENFANTS` int(11) NOT NULL default '0',
9.    PRIMARY KEY  (`ID`)
10. ) ENGINE=InnoDB DEFAULT CHARSET=latin1
```

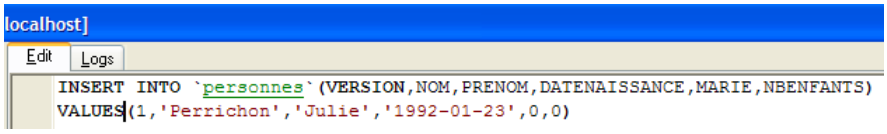
MySQL 4.x semble plus pauvre que les deux SGBD précédents. Je n'ai pas pu mettre de contraintes (checks) à la table.

- ligne 10 : la table doit avoir le type [InnoDB] et non le type [MyISAM] qui ne supporte pas les transactions.
- ligne 2 : la clé primaire est de type **auto_increment**. Si on insère une ligne sans valeur pour la colonne ID de la table, MySQL générera automatiquement un nombre entier pour cette colonne. Cela va nous éviter de générer les clés primaires nous-mêmes.

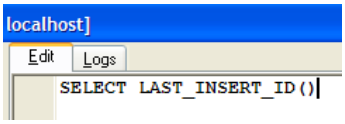
La table [PERSONNES] pourrait avoir le contenu suivant :



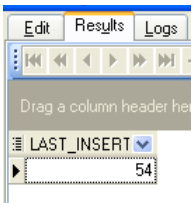
Nous savons que lors de l'insertion d'un objet [Personne] par notre couche [dao], le champ [id] de cet objet est égal à -1 avant l'insertion et a une valeur différente de -1 ensuite, cette valeur étant la clé primaire affectée à la nouvelle ligne insérée dans la table [PERSONNES]. Voyons sur un exemple comment nous allons pouvoir connaître cette valeur.



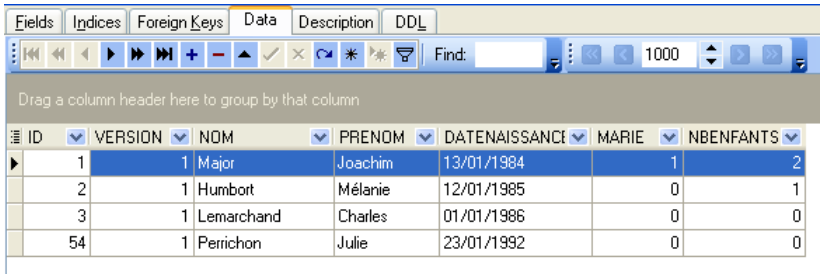
- on exécute l'ordre d'insertion ci-dessus (F12). On notera qu'on ne donne pas de valeur au champ ID ->



- on exécute l'ordre SELECT ci-dessus pour connaître la dernière valeur insérée dans le champ ID de la table ->



- le résultat



- vérification

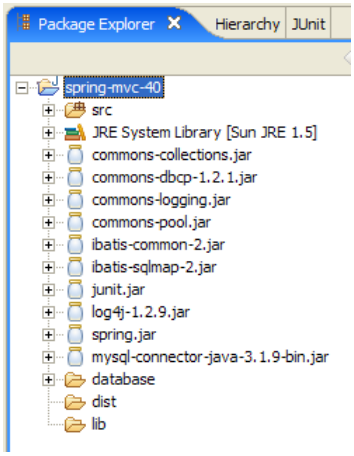
L'ordre SQL

```
SELECT LAST_INSERT_ID()
```

permet de connaître la dernière valeur insérée dans le champ ID de la table. Elle est à émettre **après l'insertion**. C'est une différence avec les SGBD [Firebird] et [Postgres] où on demandait la valeur de la clé primaire de la personne ajoutée **avant l'insertion**. Nous l'utiliserons dans le fichier [personnes-mysql.xml] qui rassemble les ordres SQL émis sur la base de données.

4.2 Le projet Eclipse des couches [dao] et [service]

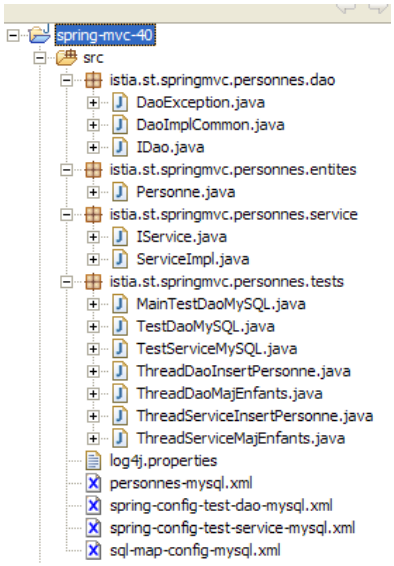
Pour développer les couches [dao] et [service] de notre application avec la base de données MySQL, nous utiliserons le projet Eclipse [spring-mvc-40] suivant :



Le projet est un simple projet Java, pas un projet web Tomcat.

Dossier [src]

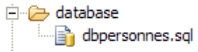
Ce dossier contient les codes source des couches [dao] et [service] :



Tous les fichiers ayant [mysql] dans leur nom ont pu subir ou non une modification vis à vis des versions Firebird et Postgres. Dans ce qui suit, nous décrivons ceux qui ont été modifiés.

Dossier [database]

Ce dossier contient le script de création de la base de données MySQL des personnes :

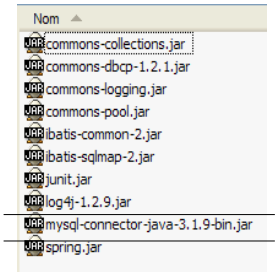



```
1.  # EMS MySQL Manager Lite 3.2.0.1
2.  # ---------------------------------------
3.  # Host     : localhost
4.  # Port     : 3306
5.  # Database : dbpersonnes
6.
7.
8.  SET FOREIGN_KEY_CHECKS=0;
9.
10. CREATE DATABASE `dbpersonnes`
11.     CHARACTER SET 'latin1'
12.     COLLATE 'latin1_swedish_ci';
13.
14. USE `dbpersonnes`;
15.
16. #
17. # Structure for the `personnes` table :
```

```
18. #
19.
20. CREATE TABLE `personnes` (
21.   `ID` int(11) NOT NULL auto_increment,
22.   `VERSION` int(11) NOT NULL default '0',
23.   `NOM` varchar(30) NOT NULL default '',
24.   `PRENOM` varchar(30) NOT NULL default '',
25.   `DATENAISSANCE` date NOT NULL default '0000-00-00',
26.   `MARIE` tinyint(4) NOT NULL default '0',
27.   `NBENFANTS` int(11) NOT NULL default '0',
28.   PRIMARY KEY  (`ID`)
29. ) ENGINE=InnoDB DEFAULT CHARSET=latin1;
30.
31. #
32. # Data for the `personnes` table  (LIMIT 0,500)
33. #
34.
35. INSERT INTO `personnes` (`ID`, `VERSION`, `NOM`, `PRENOM`, `DATENAISSANCE`, `MARIE`, `NBENFANTS`)
    VALUES
36.   (1,1,'Major','Joachim','1984-01-13',1,2),
37.   (2,1,'Humbort','Mélanie','1985-01-12',0,1),
38.   (3,1,'Lemarchand','Charles','1986-01-01',0,0);
39.
40. COMMIT;
```

Dossier [lib]

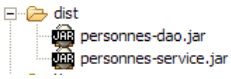
Ce dossier contient les archives nécessaires à l'application :



On notera la présence du pilote jdbc du SGBD MySQL. Toutes ces archives font partie du *Classpath* du projet Eclipse.

Dossier [dist]

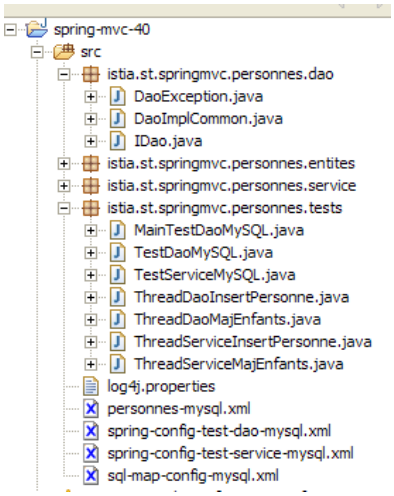
Ce dossier contiendra les archives issues de la compilation des classes de l'application :



- [personnes-dao.jar] : archive de la couche [dao]
- [personnes-service.jar] : archive de la couche [service]

4.3 La couche [dao]

La couche [dao] est la suivante :



Nous ne présentons que ce qui change vis à vis de la version [Firebird].

Le fichier de mapping [personne-mysql.xml] est le suivant :

```
1.  <?xml version="1.0" encoding="UTF-8" ?>
2.
3.  <!DOCTYPE sqlMap
4.      PUBLIC "-//iBATIS.com//DTD SQL Map 2.0//EN"
5.      "http://www.ibatis.com/dtd/sql-map-2.dtd">
6.
7.  <sqlMap>
8.    <!-- alias classe [Personne] -->
9.    <typeAlias alias="Personne.classe"
10.     type="istia.st.springmvc.personnes.entites.Personne"/>
11.   <!-- mapping table [PERSONNES] - objet [Personne] -->
12.   <resultMap id="Personne.map"
13.     class="istia.st.springmvc.personnes.entites.Personne">
14.     <result property="id" column="ID" />
15.     <result property="version" column="VERSION" />
16.     <result property="nom" column="NOM"/>
17.     <result property="prenom" column="PRENOM"/>
18.     <result property="dateNaissance" column="DATENAISSANCE"/>
19.     <result property="marie" column="MARIE"/>
20.     <result property="nbEnfants" column="NBENFANTS"/>
21.   </resultMap>
22.   <!-- liste de toutes les personnes -->
23.   <select id="Personne.getAll" resultMap="Personne.map" > select ID, VERSION, NOM,
24.     PRENOM, DATENAISSANCE, MARIE, NBENFANTS FROM PERSONNES</select>
25.   <!-- obtenir une personne en particulier -->
26.     <select id="Personne.getOne" resultMap="Personne.map" >select ID, VERSION, NOM,
27.     PRENOM, DATENAISSANCE, MARIE, NBENFANTS FROM PERSONNES WHERE ID=#value#</select>
28.   <!-- ajouter une personne -->
29.   <insert id="Personne.insertOne" parameterClass="Personne.classe">
30.     insert into
31.     PERSONNES(VERSION, NOM, PRENOM, DATENAISSANCE, MARIE, NBENFANTS)
32.     VALUES(#version#, #nom#, #prenom#, #dateNaissance#, #marie#,
33.     #nbEnfants#)
34.     <selectKey keyProperty="id">
35.       select LAST_INSERT_ID() as value
36.     </selectKey>
37.   </insert>
38.   <!-- mettre à jour une personne -->
39.   <update id="Personne.updateOne" parameterClass="Personne.classe"> update
40.     PERSONNES set VERSION=#version#+1, NOM=#nom#, PRENOM=#prenom#, DATENAISSANCE=#dateNaissance#,
41.     MARIE=#marie#, NBENFANTS=#nbEnfants# WHERE ID=#id# and
42.     VERSION=#version#</update>
43.   <!-- supprimer une personne -->
44.   <delete id="Personne.deleteOne" parameterClass="int"> delete FROM PERSONNES WHERE
45.     ID=#value# </delete>
46.   <!-- obtenir la valeur de la clé primaire [id] de la dernière personne insérée -->
47.   <select id="Personne.getNextId" resultClass="int">select
48.     LAST_INSERT_ID()</select>
49. </sqlMap>
```

C'est le même contenu que [personnes-firebird.xml] aux détails près suivants :

- l'ordre SQL " Personne.insertOne " a changé lignes 29-37 :

- l'ordre SQL d'insertion est exécuté avant l'ordre SELECT qui va permettre de récupérer la valeur de la clé primaire de la ligne insérée
- l'ordre SQL d'insertion n'a pas de valeur pour la colonne ID de la table [PERSONNES]

Cela reflète l'exemple d'insertion que nous avons commenté page 56.

On notera qu'il y a peut-être là une source possible de problèmes entre threads concurrents. Imaginons deux threads Th1 et Th2 qui font une insertion en même temps. Il y a au total quatre ordres SQL à émettre. Supposons qu'ils soient faits dans l'ordre suivant :

1. insertion I1 de Th1
2. insertion I2 de Th2
3. select S1 de Th1
4. select S2 de Th2

En 3, Th1 récupère la clé primaire générée lors de la dernière insertion, donc celle de Th2 et non la sienne. Je ne sais pas si la méthode [insert] d'iBATIS est protégée pour ce cas de figure. Nous allons supposer qu'elle le gère proprement. Si ce n'était pas le cas, il nous faudrait dériver la classe d'implémentation [DaoImplCommon] de la couche [dao] en une classe [DaoImplMySQL] où la méthode [insertPersonne] serait synchronisée. Ceci ne résoudrait le problème que pour les threads de notre application. Si ci-dessus, Th1 et Th2 sont des threads de deux applications différentes, il faudrait alors résoudre le problème à la fois avec des transactions et un niveau d'étanchéité adéquat (isolation level) entre transactions. Le niveau [serializable] dans lequel les transactions sont exécutées comme si elles s'exécutaient séquentiellement serait approprié.

On notera que ce problème n'existe pas avec les SGBD Firebird et Postgres qui eux font le SELECT avant l'INSERT. Si on a par exemple la séquence :

1. select S1 de Th1
2. select S2 de Th2
3. insertion I1 de Th1
4. insertion I2 de Th2

Aux étapes 1 et 2, Th1 et Th2 récupèrent des valeurs de clé primaire auprès du même générateur. Cette opération est normalement atomique, et Th1 et Th2 vont récupérer deux valeurs différentes. Si l'opération n'était pas atomique, et que Th1 et Th2 récupéraient deux valeurs identiques, l'insertion faite en 4 par Th2 échouerait pour cause de doublon de clé primaire. C'est une erreur tout à fait récupérable et Th2 peut retenter l'insertion.

On va laisser l'opération " Personne.insertOne " telle qu'elle est actuellement dans le fichier [personnes-mysql.xml] mais le lecteur doit avoir conscience qu'il y a là potentiellement un problème.

La classe d'implémentation [DaoImplCommon] de la couche [dao] est celle des deux versions précédentes.

La configuration de la couche [dao] a été adaptée au SGBD [MySQL]. Ainsi, le fichier de configuration [spring-config-test-dao-mysql.xml] est-il le suivant :

```
1.  <?xml version="1.0" encoding="ISO_8859-1"?>
2.  <!DOCTYPE beans SYSTEM "http://www.springframework.org/dtd/spring-beans.dtd">
3.  <beans>
4.    <!-- la source de donnéees DBCP -->
5.    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
6.      destroy-method="close">
7.      <property name="driverClassName">
8.        <value>com.mysql.jdbc.Driver</value>
9.      </property>
10.     <property name="url">
11.       <value>jdbc:mysql://localhost/dbpersonnes</value>
12.     </property>
13.     <property name="username">
14.       <value>root</value>
15.     </property>
16.     <property name="password">
17.       <value></value>
18.     </property>
19.   </bean>
20.   <!-- SqlMapCllient -->
21.   <bean id="sqlMapClient"
22.     class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
23.     <property name="dataSource">
24.       <ref local="dataSource"/>
25.     </property>
26.     <property name="configLocation">
27.       <value>classpath:sql-map-config-mysql.xml</value>
28.     </property>
```

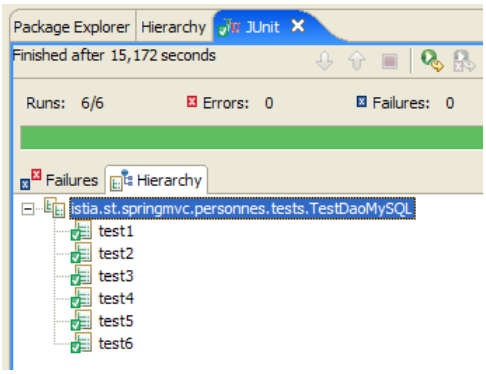
```
29.   </bean>
30.   <!-- la classes d'accè à la couche [dao] -->
31.   <bean id="dao" class="istia.st.springmvc.personnes.dao.DaoImplCommon">
32.     <property name="sqlMapClient">
33.       <ref local="sqlMapClient"/>
34.     </property>
35.   </bean>
36.</beans>
```

- lignes 5-19 : le bean [dataSource] désigne maintenant la base [MySQL] [dbpersonnes] dont l'administrateur est [root] sans mot de passe. Le lecteur modifiera cette configuration selon son propre environnement.
- ligne 31 : la classe [DaoImplCommon] est la classe d'implémentation de la couche [dao]

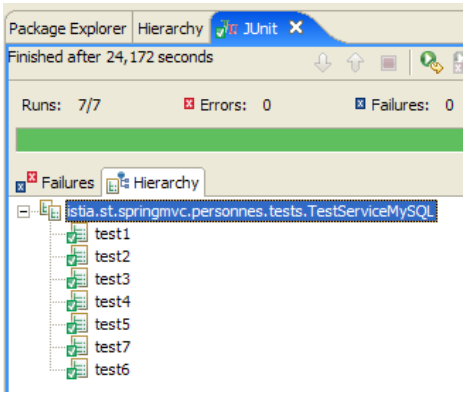
Ces modifications faites, on peut passer aux tests.

4.4 Les tests des couches [dao] et [service]

Les tests des couches [dao] et [service] sont les mêmes que pour la version [Firebird]. Les résultats obtenus sont les suivants :



- couche [dao]



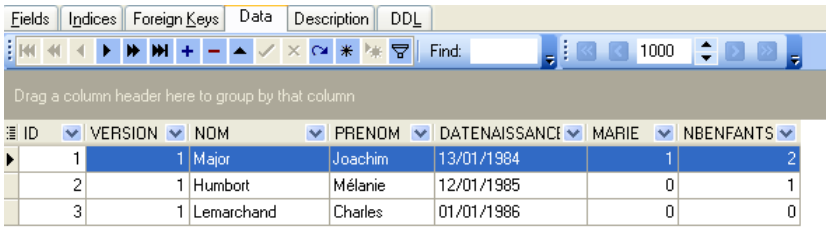
- couche [service]

On constate que les tests ont été passés avec succès avec l'implémentation [DaoImplCommon]. Nous n'aurons pas à dériver cette classe comme il avait été nécessaire de le faire avec le SGBD [Firebird].

4.5 Tests de l'application [web]

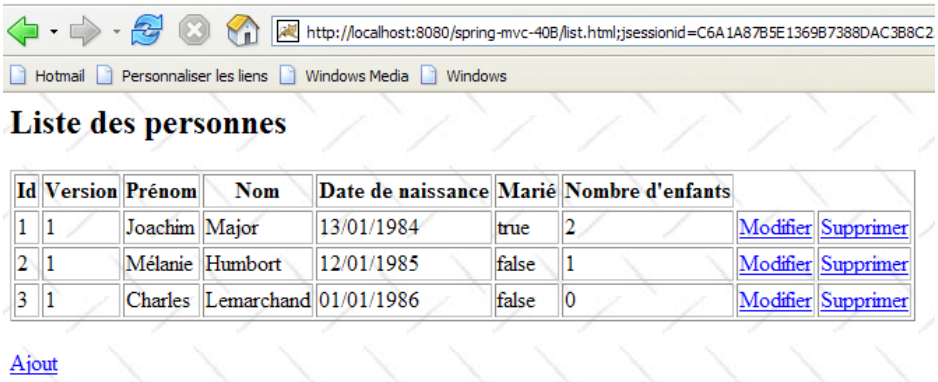
Pour tester l'application web avec le SGBD [MySQL], nous construisons un projet Eclipse [spring-mvc-40B] de façon analogue à celle utilisée pour construire le projet [spring-mvc-38B] avec la base Firebird (cf page 39).

Le SGBD MySQL est lancé. Le contenu de la table [PERSONNES] est alors le suivant :

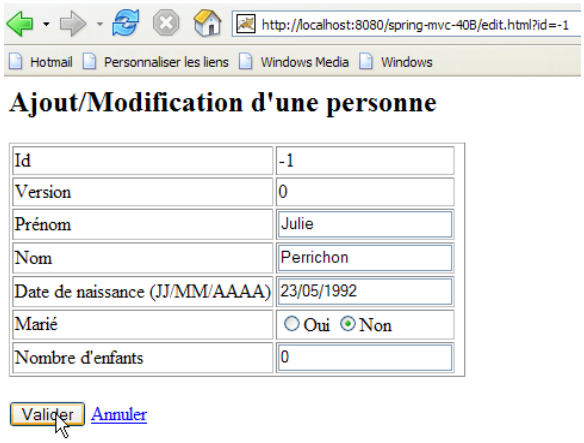


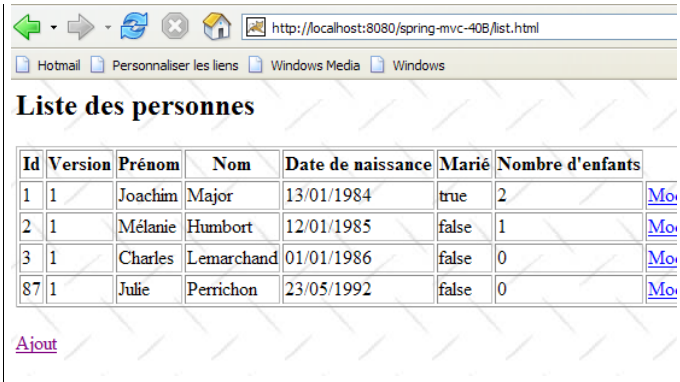
ID	VERSION	NOM	PRENOM	DATENAISSANCE	MARIE	NBENFANTS
1	1	Major	Joachim	13/01/1984	1	2
2	1	Humbort	Mélanie	12/01/1985	0	1
3	1	Lemarchand	Charles	01/01/1986	0	0

Tomcat est lancé à son tour. Avec un navigateur, nous demandons l'url [http://localhost:8080/spring-mvc-40B] :

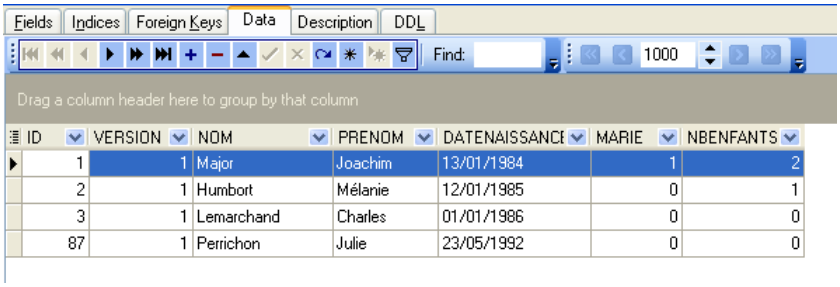


Nous ajoutons une nouvelle personne avec le lien [Ajout] :





Nous vérifions l'ajout dans la base de données :



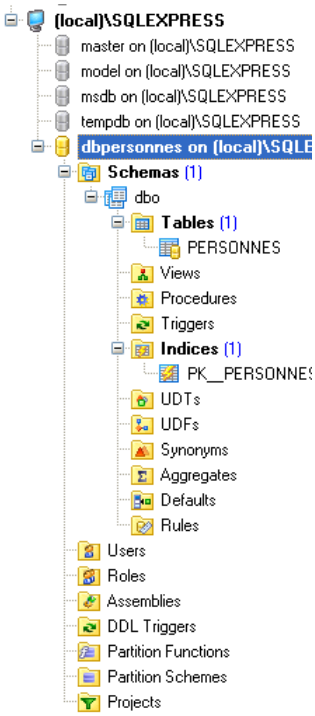
Le lecteur est invité à faire d'autres tests [modification, suppression].

5 Spring MVC dans une architecture 3tier – Exemple 5 – SQL Server Express

5.1 La base de données SQL Server Express

Dans cette version, nous allons installer la liste des personnes dans une table de base de données SQL Server Express 2005 disponible à l'url [http://msdn.microsoft.com/vstudio/express/sql/]. Dans ce qui suit, les copies d'écran proviennent du client EMS Manager Lite pour SQL Server Express [http://www.sqlmanager.net/fr/products/mssql/manager], un client d'administration gratuit du SGBD SQL Server Express.

La base de données s'appelle [dbpersonnes]. Elle contient une table [PERSONNES] :

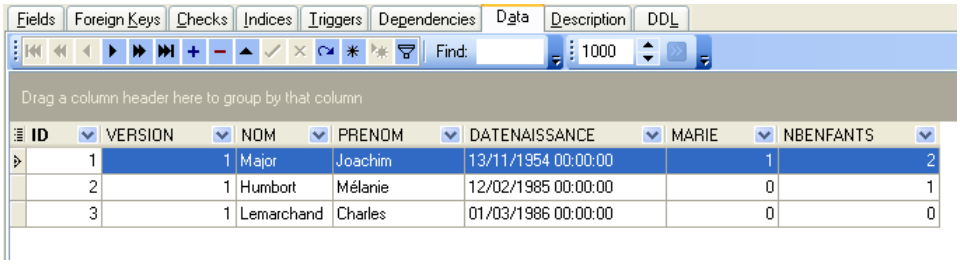


La table [PERSONNES] contiendra la liste des personnes gérée par l'application web. Elle a été construite avec les ordres SQL suivants :

```
1.  CREATE TABLE [dbo].[PERSONNES] (
2.    [ID] int IDENTITY(1, 1) NOT NULL,
3.    [VERSION] int NOT NULL,
4.    [NOM] varchar(30) COLLATE French_CI_AS NOT NULL,
5.    [PRENOM] varchar(30) COLLATE French_CI_AS NOT NULL,
6.    [DATENAISSANCE] datetime NOT NULL,
7.    [MARIE] tinyint NOT NULL,
8.    [NBENFANTS] tinyint NOT NULL,
9.    PRIMARY KEY CLUSTERED ([ID]),
10.   CONSTRAINT [PERSONNES_ck_NOM] CHECK ([NOM]<>''),
11.   CONSTRAINT [PERSONNES_ck_PRENOM] CHECK ([PRENOM]<>''),
12.   CONSTRAINT [PERSONNES_ck_NBENFANTS] CHECK ([NBENFANTS]>=(0))
13. )
14. ON [PRIMARY]
15. GO
```

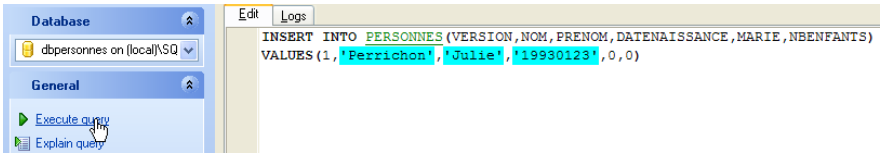
- ligne 2 : la clé primaire [ID] est de type entier. L'attribut **IDENTITY** indique que si on insère une ligne sans valeur pour la colonne ID de la table, SQL Express générera lui-même un nombre entier pour cette colonne. Dans IDENTITY(1, 1), le premier paramètre est la première valeur possible pour la clé primaire, le second, l'incrément utilisé dans la génération des nombres.

La table [PERSONNES] pourrait avoir le contenu suivant :

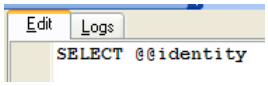


ID	VERSION	NOM	PRENOM	DATENAISSANCE	MARIE	NBENFANTS
1	1	Major	Joachim	13/11/1954 00:00:00	1	2
2	1	Humbort	Mélanie	12/02/1985 00:00:00	0	1
3	1	Lemarchand	Charles	01/03/1986 00:00:00	0	0

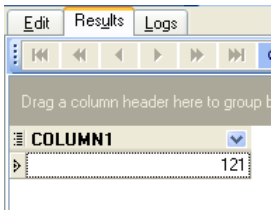
Nous savons que lors de l'insertion d'un objet [Personne] par notre couche [dao], le champ [id] de cet objet est égal à -1 avant l'insertion et a une valeur différente de -1 ensuite, cette valeur étant la clé primaire affectée à la nouvelle ligne insérée dans la table [PERSONNES]. Voyons sur un exemple comment nous allons pouvoir connaître cette valeur.



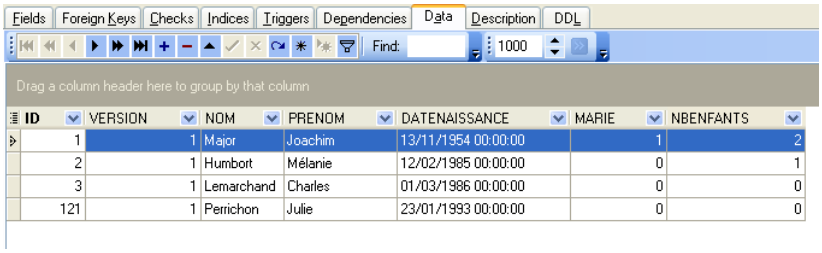
- on exécute l'ordre d'insertion ci-dessus (F12). On notera qu'on ne donne pas de valeur au champ ID ->



- on exécute l'ordre SELECT ci-dessus pour connaître la dernière valeur insérée dans le champ ID de la table ->



- le résultat



- vérification

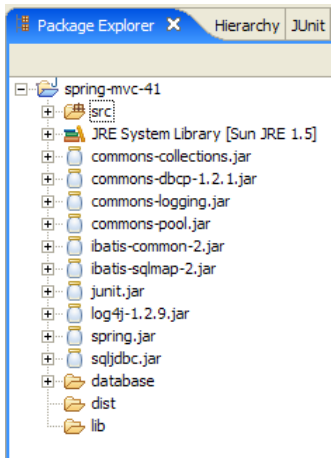
L'ordre SQL

```
SELECT @@IDENTITY
```

permet de connaître la dernière valeur insérée dans le champ ID de la table. Elle est à émettre **après l'insertion**. C'est une différence avec les SGBD [Firebird] et [Postgres] où on demandait la valeur de la clé primaire de la personne ajoutée **avant l'insertion**, mais c'est analogue à la génération de clé primaire du SGBD MySQL. Nous l'utiliserons dans le fichier [personnes-sqlexpress.xml] qui rassemble les ordres SQL émis sur la base de données.

5.2 Le projet Eclipse des couches [dao] et [service]

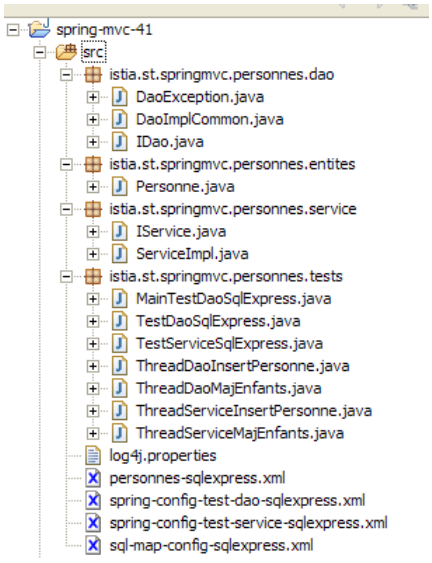
Pour développer les couches [dao] et [service] de notre application avec la base de données[SQL Server Express], nous utiliserons le projet Eclipse [spring-mvc-41] suivant :



Le projet est un simple projet Java, pas un projet web Tomcat.

Dossier [src]

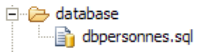
Ce dossier contient les codes source des couches [dao] et [service] :



Tous les fichiers ayant [sqlexpress] dans leur nom ont pu subir ou non une modification vis à vis des versions Firebird, Postgres et MySQL. Dans ce qui suit, nous ne décrivons que ceux qui ont été modifiés.

Dossier [database]

Ce dossier contient le script de création de la base de données SQL Express des personnes :

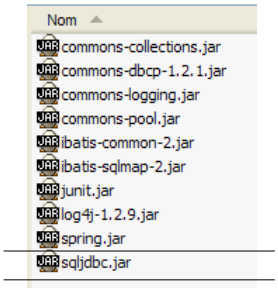



```
1.  -- SQL Manager 2005 Lite for SQL Server (2.2.0.1)
2.  -- ---------------------------------------
3.  -- Host     : (local)\SQLEXPRESS
4.  -- Database : dbpersonnes
5.
6.
7.  --
8.  -- Structure for table PERSONNES :
9.  --
10.
11. CREATE TABLE [dbo].[PERSONNES] (
12.   [ID] int IDENTITY(1, 1) NOT NULL,
13.   [VERSION] int NOT NULL,
14.   [NOM] varchar(30) COLLATE French_CI_AS NOT NULL,
15.   [PRENOM] varchar(30) COLLATE French_CI_AS NOT NULL,
16.   [DATENAISSANCE] datetime NOT NULL,
17.   [MARIE] tinyint NOT NULL,
18.   [NBENFANTS] tinyint NOT NULL,
19.   CONSTRAINT [PERSONNES_ck_NBENFANTS] CHECK ([NBENFANTS]>=(0)),
20.   CONSTRAINT [PERSONNES_ck_NOM] CHECK ([NOM]<>''),
21.   CONSTRAINT [PERSONNES_ck_PRENOM] CHECK ([PRENOM]<>'')
22. )
23. ON [PRIMARY]
24. GO
25.
26. --
27. -- Data for table PERSONNES  (LIMIT 0,500)
28. --
29.
30. SET IDENTITY_INSERT [dbo].[PERSONNES] ON
31. GO
32.
33. INSERT INTO [dbo].[PERSONNES] ([ID], [VERSION], [NOM], [PRENOM], [DATENAISSANCE], [MARIE],
    [NBENFANTS])
34. VALUES
35.   (1, 1, 'Major', 'Joachim', '19541113', 1, 2)
36. GO
37.
38. INSERT INTO [dbo].[PERSONNES] ([ID], [VERSION], [NOM], [PRENOM], [DATENAISSANCE], [MARIE],
    [NBENFANTS])
39. VALUES
40.   (2, 1, 'Humbort', 'Mélanie', '19850212', 0, 1)
41. GO
42.
43. INSERT INTO [dbo].[PERSONNES] ([ID], [VERSION], [NOM], [PRENOM], [DATENAISSANCE], [MARIE],
    [NBENFANTS])
44. VALUES
45.   (3, 1, 'Lemarchand', 'Charles', '19860301', 0, 0)
```

```
46. GO
47.
48. SET IDENTITY_INSERT [dbo].[PERSONNES] OFF
49. GO
50.
51. --
52. -- Definition for indices :
53. --
54.
55. ALTER TABLE [dbo].[PERSONNES]
56. ADD PRIMARY KEY CLUSTERED ([ID])
57. WITH (
58.   PAD_INDEX = OFF,
59.   IGNORE_DUP_KEY = OFF,
60.   STATISTICS_NORECOMPUTE = OFF,
61.   ALLOW_ROW_LOCKS = ON,
62.   ALLOW_PAGE_LOCKS = ON)
63. ON [PRIMARY]
64. GO
```

Dossier [lib]

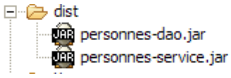
Ce dossier contient les archives nécessaires à l'application :



On notera la présence du pilote jdbc [sqljdbc.jar] du SGBD [Sql Server Express]. Toutes ces archives font partie du *Classpath* du projet Eclipse.

Dossier [dist]

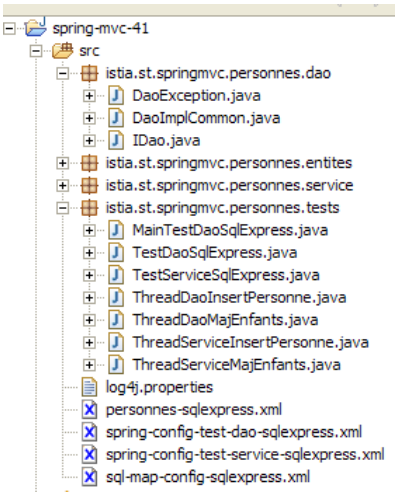
Ce dossier contiendra les archives issues de la compilation des classes de l'application :



- [personnes-dao.jar] : archive de la couche [dao]
- [personnes-service.jar] : archive de la couche [service]

5.3 La couche [dao]

La couche [dao] est la suivante :



Nous ne présentons que ce qui change vis à vis de la version [Firebird].

Le fichier de mapping [personne-sqlexpress.xml] est le suivant :

```
1.  <?xml version="1.0" encoding="UTF-8" ?>
2.
3.  <!DOCTYPE sqlMap
4.      PUBLIC "-//iBATIS.com//DTD SQL Map 2.0//EN"
5.      "http://www.ibatis.com/dtd/sql-map-2.dtd">
6.
7.  <sqlMap>
8.    <!-- alias classe [Personne] -->
9.    <typeAlias alias="Personne.classe"
10.     type="istia.st.springmvc.personnes.entites.Personne"/>
11.   <!-- mapping table [PERSONNES] - objet [Personne] -->
12.   <resultMap id="Personne.map"
13.     class="istia.st.springmvc.personnes.entites.Personne">
14.     <result property="id" column="ID" />
15.     <result property="version" column="VERSION" />
16.     <result property="nom" column="NOM"/>
17.     <result property="prenom" column="PRENOM"/>
18.     <result property="dateNaissance" column="DATENAISSANCE"/>
19.     <result property="marie" column="MARIE"/>
20.     <result property="nbEnfants" column="NBENFANTS"/>
21.   </resultMap>
22.   <!-- liste de toutes les personnes -->
23.   <select id="Personne.getAll" resultMap="Personne.map" > select ID, VERSION, NOM,
24.     PRENOM, DATENAISSANCE, MARIE, NBENFANTS FROM PERSONNES</select>
25.   <!-- obtenir une personne en particulier -->
26.     <select id="Personne.getOne" resultMap="Personne.map" >select ID, VERSION, NOM,
27.     PRENOM, DATENAISSANCE, MARIE, NBENFANTS FROM PERSONNES WHERE ID=#value#</select>
28.   <!-- ajouter une personne -->
29.   <insert id="Personne.insertOne" parameterClass="Personne.classe">
30.     insert into
31.     PERSONNES(VERSION, NOM, PRENOM, DATENAISSANCE, MARIE, NBENFANTS)
32.     VALUES(#version#, #nom#, #prenom#, #dateNaissance#, #marie#,
33.     #nbEnfants#)
34.     <selectKey keyProperty="id">
35.       select @@IDENTITY as value
36.     </selectKey>
37.   </insert>
38.   <!-- mettre à jour une personne -->
39.   <update id="Personne.updateOne" parameterClass="Personne.classe"> update
40.     PERSONNES set VERSION=#version#+1, NOM=#nom#, PRENOM=#prenom#, DATENAISSANCE=#dateNaissance#,
41.     MARIE=#marie#, NBENFANTS=#nbEnfants# WHERE ID=#id# and
42.     VERSION=#version#</update>
43.   <!-- supprimer une personne -->
44.   <delete id="Personne.deleteOne" parameterClass="int"> delete FROM PERSONNES WHERE
45.     ID=#value# </delete>
46.   <!-- obtenir la valeur de la clé primaire [id] de la dernière personne insérée -->
47.   <select id="Personne.getNextId" resultClass="int">select
48.     LAST_INSERT_ID()</select>
49. </sqlMap>
```

C'est le même contenu que [personnes-firebird.xml] aux détails près suivants :

- l'ordre SQL " Personne.insertOne " a changé lignes 29-37 :
  - l'ordre SQL d'insertion est exécuté avant l'ordre SELECT qui va permettre de récupérer la valeur de la clé primaire de la ligne insérée

- l'ordre SQL d'insertion n'a pas de valeur pour la colonne ID de la table [PERSONNES]

Cela reflète l'exemple d'insertion que nous avons commenté page 63. A noter, qu'on retrouve ici le problème d'insertions simultanées par des threads différents décrit pour MySQL au paragraphe 4.3, page 60.

La classe d'implémentation [DaoImplCommon] de la couche [dao] est celle des trois versions précédentes.

La configuration de la couche [dao] a été adaptée au SGBD [SQL Express]. Ainsi, le fichier de configuration [spring-config-test-dao-sqlexpress.xml] est-il le suivant :

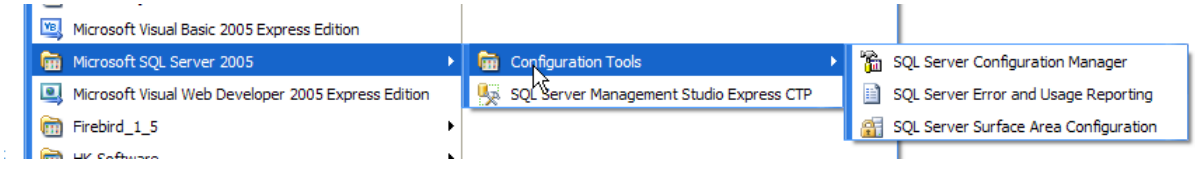
```
1.  <?xml version="1.0" encoding="ISO_8859-1"?>
2.  <!DOCTYPE beans SYSTEM "http://www.springframework.org/dtd/spring-beans.dtd">
3.  <beans>
4.    <!-- la source de donnéees DBCP -->
5.    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
6.      destroy-method="close">
7.      <property name="driverClassName">
8.        <value>com.microsoft.sqlserver.jdbc.SQLServerDriver</value>
9.      </property>
10.     <property name="url">
11.       <value>jdbc:sqlserver://localhost\\SQLEXPRESS:4000;databaseName=dbpersonnes</value>
12.     </property>
13.     <property name="username">
14.       <value>sa</value>
15.     </property>
16.     <property name="password">
17.       <value>msde</value>
18.     </property>
19.   </bean>
20.   <!-- SqlMapCllient -->
21.   <bean id="sqlMapClient"
22.     class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
23.     <property name="dataSource">
24.       <ref local="dataSource"/>
25.     </property>
26.     <property name="configLocation">
27.       <value>classpath:sql-map-config-sqlexpress.xml</value>
28.     </property>
29.   </bean>
30.   <!-- la classes d'accè à la couche [dao] -->
31.   <bean id="dao" class="istia.st.springmvc.personnes.dao.DaoImplCommon">
32.     <property name="sqlMapClient">
33.       <ref local="sqlMapClient"/>
34.     </property>
35.   </bean>
36. </beans>
```

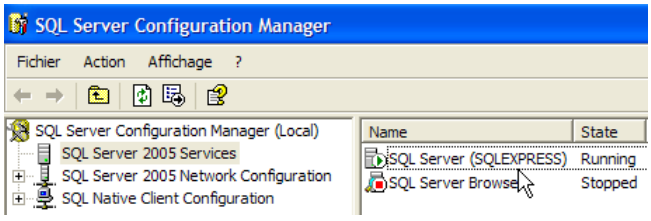
- lignes 5-19 : le bean [dataSource] désigne maintenant la base [SQL Express] [dbpersonnes] dont l'administrateur est [sa] avec le mot de passe [msde]. Le lecteur modifiera cette configuration selon son propre environnement.
- ligne 31 : la classe [DaoImplCommon] est la classe d'implémentation de la couche [dao]

La ligne 11 mérite des explications :

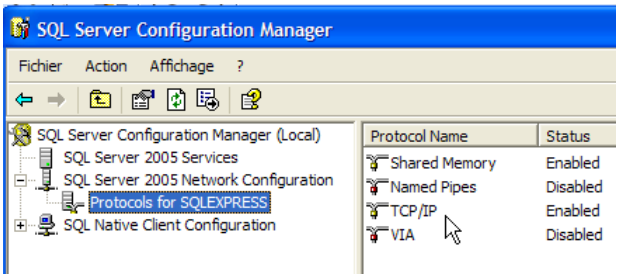
```
    <value>jdbc:sqlserver://localhost\\SQLEXPRESS:4000;databaseName=dbpersonnes</value>
```

- **//localhost** : indique que le serveur SQL Express est sur la même machine que notre application Java
- **\\SQLEXPRESS** : est le nom d'une instance de SQL Server. Il semble que plusieurs instances peuvent s'exécuter en même temps. Il semble donc logique de nommer l'instance à laquelle on s'adresse. Ce nom peut être obtenu grâce [SQL Server Configuration Manager] installé normalement en même temps que SQL Express :

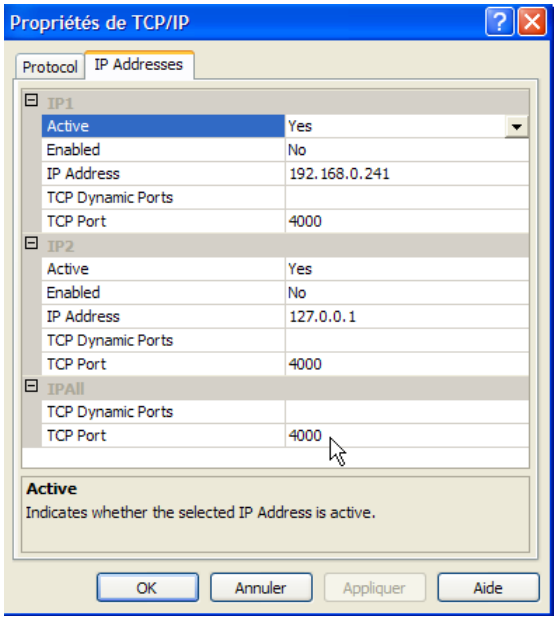




- **4000** : port d'écoute de SQL Express. Cela est dépendant de la configuration du serveur. Par défaut, il travaille avec des ports dynamiques, donc pas connus à l'avance. On ne précise alors pas de port dans l'url JDBC. Ici, nous avons travaillé avec un port fixe, le port 4000. Cela s'obtient par configuration :

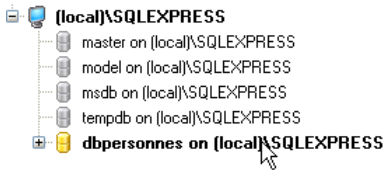


- [clic droit sur TCP/IP -> Propriétés] ->



- on laisse les champs [TCP Dynamic Ports] vides
- on met les champs [TCP Port] à 4000

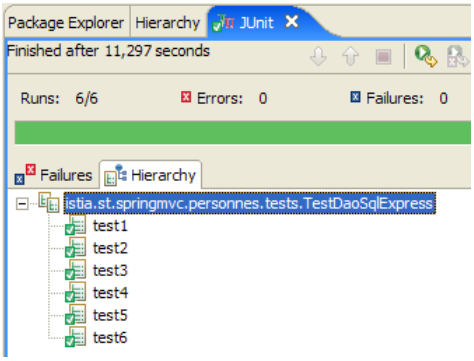
- l'attribut **dataBaseName** fixe la base de données avec laquelle on veut travailler. C'est celle qui a été créée avec le client EMS :



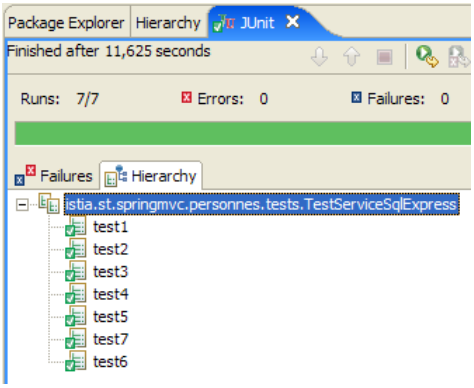
Ces modifications faites, on peut passer aux tests.

5.4 Les tests des couches [dao] et [service]

Les tests des couches [dao] et [service] sont les mêmes que pour la version [Firebird]. Les résultats obtenus sont les suivants :



- couche [dao]



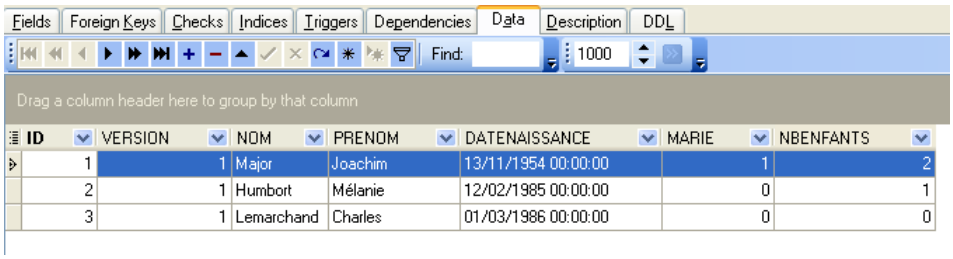
- couche [service]

On constate que les tests ont été passés avec succès avec l'implémentation [DaoImplCommon]. Nous n'aurons pas à dériver cette classe comme il avait été nécessaire de le faire avec le SGBD [Firebird].

5.5 Tests de l'application [web]

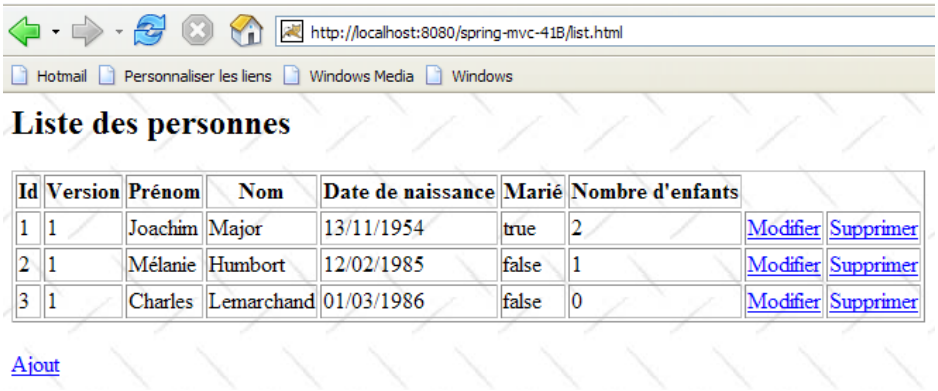
Pour tester l'application web avec le SGBD [SQL Server Express], nous construisons un projet Eclipse [spring-mvc-41B] de façon analogue à celle utilisée pour construire le projet [spring-mvc-38B] avec la base Firebird (cf page 39).

Le SGBD SQL server Express est lancé. Le contenu de la table [PERSONNES] est alors le suivant :

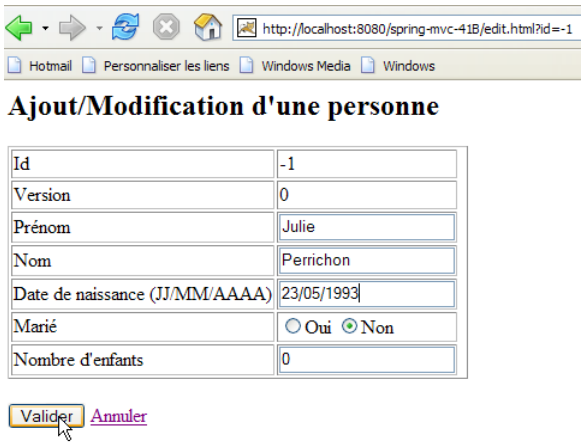


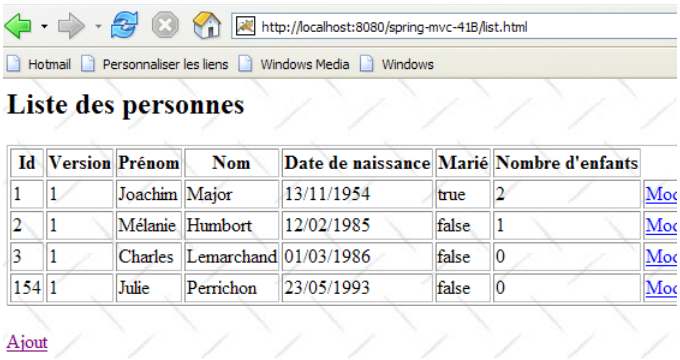
ID	VERSION	NOM	PRENOM	DATENAISSANCE	MARIE	NBENFANTS
1	1	Major	Joachim	13/11/1954 00:00:00	1	2
2	1	Humbort	Mélanie	12/02/1985 00:00:00	0	1
3	1	Lemarchand	Charles	01/03/1986 00:00:00	0	0

Tomcat est lancé à son tour. Avec un navigateur, nous demandons l'url [http://localhost:8080/spring-mvc-41B] :

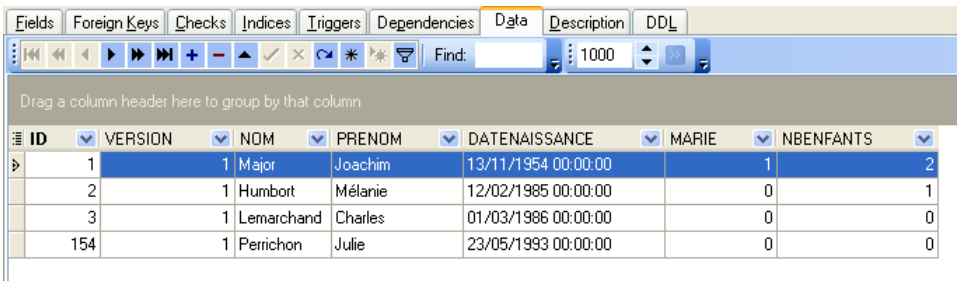


Nous ajoutons une nouvelle personne avec le lien [Ajout] :





Nous vérifions l'ajout dans la base de données :

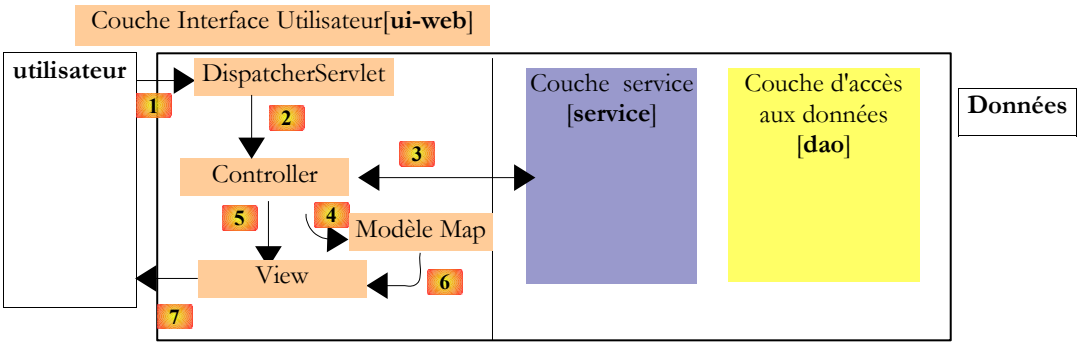


ID	VERSION	NOM	PRENOM	DATENAISSANCE	MARIE	NBENFANTS
1	1	Major	Joachim	13/11/1954 00:00:00	1	2
2	1	Humbort	Mélanie	12/02/1985 00:00:00	0	1
3	1	Lemarchand	Charles	01/03/1986 00:00:00	0	0
154	1	Perrichon	Julie	23/05/1993 00:00:00	0	0

Le lecteur est invité à faire d'autres tests [modification, suppression].

6 Conclusion

Rappelons le but de cet article. Dans l'article qui précédait, nous avions mis en oeuvre Spring MVC dans une architecture 3tier [web, metier, dao] sur un exemple basique de gestion d'une liste de personnes.



Dans cette version 1, la liste des personnes était maintenue en mémoire et disparaissait au déchargement de l'application web. Dans les quatre nouvelles versions étudiées ici, la liste des personnes est maintenue dans une table de base de données. Nous avons utilisé quatre SGBD différents : Firebird, Postgres, MySQL et SQL Server Express. Nous n'avons développé que les seules couches [service] et [dao], la couche [web] restant celle développée pour la version 1.

Ce que nous avons voulu montrer dans cet article, c'est l'apport de Spring pour la construction des couches [dao] et [service]. Grâce à l'intégration de Spring avec iBATIS, nous avons pu construire quatre versions qui ne diffèrent que par leurs fichiers de configuration. La même classe [DaoImplCommon] a été utilisée pour implémenter la couche [dao] dans les quatre versions. Pour gérer un problème spécifique au SGBD Firebird, nous avons été amenés à dériver cette classe mais pas à la modifier.

Enfin, nous avons montré comment Spring nous permettait de gérer les transactions de façon déclarative au niveau de la couche [service].

7 Le code de l'article

Comme pour les précédents articles, le lecteur trouvera le code des exemples de ce document sous la forme d'un fichier zippé sur le site de l'article. Les règles de déploiement du fichier zippé sont à relire dans l'article 1. Une fois un projet importé dans [Eclipse] :

- copier le contenu du dossier [lib] du zip dans le dossier [WEB-INF/lib] du projet
- s'assurer que le dossier [work] existe sinon le créer : [clic droit sur projet / Projet Tomcat / Créer le dossier work]
- nettoyer le projet [Project / clean / clean selected projects]

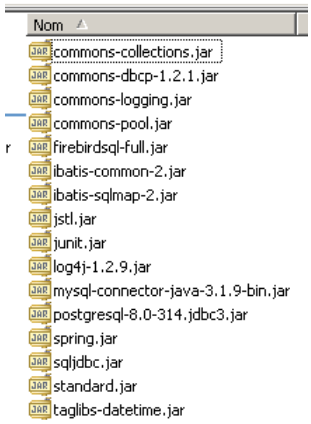
Pour rejouer les tests, le lecteur devra adapter la configuration du [DataSource] trouvé dans les différents fichiers de configuration à son propre environnement. Cet objet renseigne :

1. le **nom de classe** du pilote JDBC du SGBD utilisé
2. **l'url** de la base de données à exploiter
3. le **propriétaire** des connexions qui sont ouvertes sur la base
4. le **mot de passe** de ce dernier.

Les informations 2 à 4 sont dépendantes de l'environnement utilisé pour les tests.

Le code est livré sous la forme d'un zip contenant les dossiers suivants :

Le dossier [lib] rassemble les archives utilisées par les différents projets [mvc-xx, mvc-xxB] :



Les dossiers [lib] des différents projets ont été eux vidés afin de diminuer la taille du zip. Il faudra donc aller chercher les archives nécessaires à un projet dans le dossier [lib] ci-dessus. Dans les dossiers [lib] des projets web [mvc-xxB], on a laissé les archives des couches [dao], [service] et [web] de l'application construite par les projets associés [mvc-xx] :

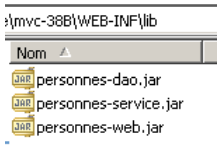


Table des matières

1 RAPPELS 2

2 SPRING MVC DANS UNE ARCHITECTURE 3TIER – EXEMPLE 2 - FIREBIRD 2

2.1 La base de données Firebird 2
2.2 Le projet Eclipse des couches [dao] et [service] 4
2.3 La couche [dao] 6
2.3.1 Les composantes de la couche [dao] 6
2.3.2 La couche d'accès aux données [iBATIS] 8
2.3.3 La classe [DaoImplCommon] 14
2.4 Tests de la couche [dao] 20
2.4.1 Tests de l'implémentation [DaoImplCommon] 20
2.4.2 La classe [DaoImplFirebird] 28
2.4.3 Tests de l'implémentation [DaoImplFirebird] 29
2.5 La couche [service] 31
2.5.1 Les composantes de la couche [service] 31
2.5.2 Configuration de la couche [service] 32
2.6 Tests de la couche [service] 36
2.7 La couche [web] 39
2.8 Conclusion 46

3 SPRING MVC DANS UNE ARCHITECTURE 3TIER – EXEMPLE 4 - POSTGRES 47

3.1 La base de données Postgres 47
3.2 Le projet Eclipse des couches [dao] et [service] 49
3.3 La couche [dao] 51
3.4 Les tests des couches [dao] et [service] 53
3.5 Tests de l'application [web] 54

4 SPRING MVC DANS UNE ARCHITECTURE 3TIER – EXEMPLE 4 - MYSQL 55

4.1 La base de données MySQL 55
4.2 Le projet Eclipse des couches [dao] et [service] 56
4.3 La couche [dao] 58
4.4 Les tests des couches [dao] et [service] 61
4.5 Tests de l'application [web] 61

5 SPRING MVC DANS UNE ARCHITECTURE 3TIER – EXEMPLE 5 – SQL SERVER EXPRESS 62

5.1 La base de données SQL Server Express 62
5.2 Le projet Eclipse des couches [dao] et [service] 64
5.3 La couche [dao] 66
5.4 Les tests des couches [dao] et [service] 69
5.5 Tests de l'application [web] 70

6 CONCLUSION 71

7 LE CODE DE L'ARTICLE 71