Cours Spring DAO Services

1 Présentation Générale

- Le framework Spring offre des supports importants dans le cadre d'utilisation de services DAO (J2EE Pattern « *Data Access Object* », famille « *Intégration* »), situés en back-end des architectures J2EE.

- Les classes *Template* servent de modèle, et de point d'entrée, pour l'exécution des processus JDBC sous-jacents.

  Le package *org.springframework* met à disposition les éléments suivants:

  - jdbc.core.JdbcTemplate:
  - jdbc.core.namedParam.NamedParameterJdbcTemplate:
  - jdbc.core.simple.SimpleJdbcTemplate:
  - orm.hibernate.HibernateTemplate
  - orm.hibernate3.HibernateTemplate
  - orm.ibatis.SqlMapClientTemplate
  - orm.jdo.JdoTemplate
  - orm.jpa.JpaTemplate
  - orm.toplink.TopLinkTemplate
  - jca.cci.core.CciTemplate

  afin de pouvoir connecter des DAO avec les technologies : JDBC (bas niveau), Spring JDBC, Hibernate (2.x, 3.x), IBatis, JDO, JPA, TopLink, et CCI.

- En complément de ces « *Data Access Template* », pour chacun des éléments, Spring fournit des classes « *DAO-Support* », représentant les *Types* de haut niveau desquels les DAO clients doivent hériter.

  - Ces *DAO-Support* permettent d'invoquer des *accessors/getters* pour récupérer l'instance de Template correspondant/associé, servant à construire les transactions ultérieures opérées par les DAO ; ainsi que d'autres méthodes pour obtenir certains objets attenants (par ex la *Connection* JDBC)

    Ex : *JdbcDaoSupport :* méthodes *getJdbcTemplate(), getConnection()*.

- Spring fournit aussi un ensemble de classes d'*Exception*, afin de masquer celles issues des Framework utilisés, ce qui est cohérent avec le principe de *« Façade »* (Design Pattern, famille « *Structuration* ») exposé précédemment
  (en terme de séparation de responsabilité sur les couches applicatives).

  Les thèmes suivants ont des exceptions qui leurs sont associés :
  Lock, Concurrency, Data Access/Integrity/Retrieval, Result, Permission, Type, ...

- Spring JDBC fournit une légère surcouche de JDBC (comparée à celle d'Hibernate considérée comme « lourde »), adapté à des traitements simples/de masses, (cf TP).

- La configuration Spring (XML ou Classe) doit déclarer une *DataSource* (contenant l'ensemble des paramètres requis pour se connecter à une DB), qui va servir à alimenter un *Template* Spring afin de le rendre opérationnel.

```
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
   <property name="driverClassName" value="com.mysql.jdbc.Driver" />
   <property name="url" value="jdbc:mysql://localhost:3306/products" />
   <property name="username" value="root" />
   <property name="password" value="nbuser" />
</bean>

<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate" >
   <property name="dataSource" ref="dataSource" />
</bean>
```

Remarque :

On pourrait paramétrer la configuration de la *DataSource* de façon dynamique, afin qu'elle récupère ses valeurs littérales (*url, driver, username, password*) dans des variables d'environnement (ou bien dans un fichier de configuration), ce qui permet d'externaliser ces éléments de configuration, et d'obtenir plus de souplesse lors des phases de déploiement.

```
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
   <property name="location">
      <value>db.properties</value>
   </property>
</bean>

<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
   <property name="locations">
      <list>
         <value>classpath:db.properties</value> <!-- classpath*:db.properties -->
      </list>
   </property>
</bean>

<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
   <property name="driverClassName" value="${db.driverClassName}" />
   <property name="url" value="${db.url}" />
   <property name="username" value="${db.username}" />
   <property name="password" value="${db.password}" />
</bean>
```

```
db.driverClassName=com.mysql.jdbc.Driver
db.url=jdbc:mysql://localhost:3306/products
db.username=root
db.password=nbuser
```

Fichier database.properties / Variables d'environnement Système

- Les classes de services DAO doivent respecter le design par interface, afin de pouvoir, le cas échéant (par exemple lors d'une migration de service DAO), être aisément remplacé par d'autres instances offrant le même niveau de services fonctionnel, puisque remplissant le contrant d'interface imposée par le *IDAO*.
    - Cf TP : passage d'un DAO Mock/factice, à un DAO Spring JDBC Template, puis finalement à un DAO Hibernate, en ayant toujours les mêmes méthodes, et comportements, à disposition.

En général il arrive ((très) souvent) que l'on complète aussi le design des DAO en fournissant une classe Abstraite déclarant dans sa signature des « *Generics ».*

Cf TP : *AbstractDAOSpring<T>*, ici *<T>* étant associé à une classe du modèle, mais on peut aussi ajouter d'autres paramètres dans la définition des *generics*, comme :

- Le type *<PK>* associé à la Primary Key
  (très utile lorsqu'une entité est défini selon une clé composée)
- Un type de *DataObject* particulier renvoyé par un DAO

Et ainsi obtenir un templating final *<T, PK, DO>*.

Remarques :

- Ne pas confondre les Templates Generics et ceux de Spring JDBC/DAO.
- Pour rappel, en Java les *Generics* sont exprimés et évalués uniquement à la compilation, on ne peut pas templater par *Generics* au moment du *Runtime*, c'est une restriction de la plateforme.

2 Contenu du TP

- 3 packages de DAO :

  - Chacune des classes concrètes de DAO (associée à un type du modèle, ici *Product* ou *Genre*) implémente son interface I***DAO associée (donc commune aux 3 implémentations, afin d'assurer l'iso-fonctionnalité pour chacune des solutions, et de permettre de basculer de l'une à l'autre en changeant – commentant/décommantant - simplement les *@nnotations* d'injection en en-tête de classes).

  - Un package de DAO Mock, *fr.shortcircuit.service.dao*, contenant des DAO renvoyant des objets instanciés statiquement (au démarrage de l'application).

  - Un package de DAO implémenté avec la surcouche Spring JDBC, présentant :
    - Une classe Abstraite *AbstractDAOSpring*, templatée par *Generics*, héritant de JdbcDaoSupport, proposant un constructeur incluant en paramètre une *DataSource*, une liste de RowMapper (structures simples d'*inner classes* permettant de mapper/lire/convertir des *ResultSet* JDBC en valeurs littérales, ou bien en objets de modèle), et des helpers méthodes (gestion de transactions).

      Remarque : Spring JDBC impose la gestion « programmative » des transactions (à travers l'utilisation d'un *TransactionTemplate,* c'est au DAO d'implémenter la gestion de la Transaction).
    - 2 classes concrètes : *ProductDAOSpring* et *GenreDAOSpring*, aux constructeurs auto-cablés par une *@nnotation* Spring *@Autowired* qui se charge d'injecter dans le constructeur la *DataSource,* elle même identifiée par l'*@nnotation @Qualifier*.

    - Un exemple de jointure entre *Product* et *Genre* dans *ProductDAOSpring*, remontant des objets mappés par l'interface de constants *IProductMapperConstants* qui se charge de « chaîner » (Design Pattern « *Chain of Responsibility* », famille « *Comportement* ») les appels vers les *ParameterizedRowMapper* de Spring JDBC

      (*productWithGenreMapper* appel *productLitteralMapper.mapRow* pour mapper les valeurs simples d'un objet Product, et complète ensuite sa construction en appelant *IGenreMapperConstants.genreLitteralMapper*. *mapRow* pour mapper la relation vers un objet Genre).

    - Une primitive de code, en commentaire, sur comment faire l'insert d'une nouvelle entrée en base, et récupérer au travers d'un objet *KeyHolder* la valeur de la PK du nouvel objet créé.

- Un package de DAO implémenté avec Hibernate possédant :
  - Une classe abstraite *AbstractDAOHibernate*, partageant pour ses sous classes une *SessionFactory* (issue du package *org.hibernate*), injectée par *@nnotation @Autowired*, et déclarée comme obligatoire, puisqu'essentiel au bon fonctionnement des DAO.
  - Les classes concrètes associées aux entités *Product* et *Genre*, et notamment la classe *ProductDAOHibernate* qui présente les fonctionnalités suivantes :

    - *Query* Hibernate SQL/HQL en lecture (grammaire !=)
    - *Query* Hibernate par API de *Criteria* (abstraction objet)
    - Update d'entité.

    Remarque : les politiques de mapping Hibernate sont déclarés dans des fichiers de mapping .hbm, dans le TP suivants seront abordés les @nnotations de mapping