3.3.4 Accès aux données avec JPA

A compléter

A revoir
Installation de JBossTools Hibernate (si partie toujours valide apres réfection)
Utilisation de plusieurs gestionnaires d'entités

Relu

Relecture RB

Sommaire :

Les exemples d'accès aux données de *esup-example* utilisent JPA. Il existe plusieurs implémentations de JPA. Celle qui est retenue est celle de *Hibernate*.

Note : L'implémentation Hibernate permet d'utiliser des fichiers de mapping (objet/base de données) spécifiques à Hibernate ce qui offre une compatibilité ascendante avec ESUP-Commons V1. De même, Hibernate permet potentiellement de faire des choses en plus de ce que la norme JPA prévoit. Mais ces deux fonctionnalités, si elles sont utilisées, ne permettent pas de passer à une autre implémentation de JPA. C'est la raison pour laquelle dans, *esup-example*, nous nous limitons à une utilisation stricte de JPA.

Le gestionnaire d'entités

En JPA, l'élément qui permet de manipuler les objets en base de données est le gestionnaire d'entités. Les objets sont annotés afin de préciser comment ils doivent être enregistés en base de données.

La déclaration se fait dans le fichier **resources/properties/dao/dao.xml** :

```
<bean class="org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor" />

        <bean id="txManager" class="org.springframework.orm.jpa.JpaTransactionManager">
                <property name="entityManagerFactory" ref="entityManagerFactory" />
        </bean>

        <bean id="entityManagerFactory" class="org.springframework.orm.jpa.
LocalContainerEntityManagerFactoryBean">
                <property name="dataSource" ref="${datasource.bean}" />
                <property name="jpaVendorAdapter" ref="jpaVendorAdapter" />
                <property name="persistenceXmlLocation" value="classpath:/properties/dao/persistence.xml" />
                <property name="jpaProperties" ref="jpaProperties" />
        </bean>

        <bean id="JDBCDataSource" class="org.apache.commons.dbcp.BasicDataSource"
                destroy-method="close"
                lazy-init="true">
                <property name="driverClassName" value="${jdbc.connection.driver_class}" />
                <property name="maxActive" value="100" />
                <property name="maxIdle" value="30" />
                <property name="maxWait" value="100" />
                <property name="url" value="${jdbc.connection.url}" />
                <property name="username" value="${jdbc.connection.username}" />
                <property name="password" value="${jdbc.connection.password}" />
        </bean>

        <jee:jndi-lookup id="JNDIDataSource" jndi-name="${jndi.datasource}" lookup-on-startup="false" expected-
type="javax.sql.DataSource"/>

        <bean id="jpaVendorAdapter"
                class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
                <property name="showSql" value="true" />
                <property name="generateDdl" value="true" />
                <property name="database" value="${jpa.database.type}" />
        </bean>

        <util:properties id="jpaProperties">
                <prop key="hibernate.cache.provider_class">org.hibernate.cache.NoCacheProvider</prop>
                <prop key="hibernate.cache.use_query_cache">false</prop>
                <prop key="hibernate.cache.use_second_level_cache">false</prop>
        </util:properties>

        <bean id="daoService" class="org.esupportail.example.dao.JPADaoServiceImpl"
                lazy-init="true">
        </bean>
```

Explications :

- Le bean**org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor** demande à Spring de rechercher les classes java contenant des annotations définissant la façon d'enregistrer les objets en base de données.
- Le bean **entityManagerFactory** permet la création du gestionnaire d'entités. Il a comme propriétés :
  - **dataSource** qui est la source de donnée à utiliser. Ici on utilise une variable Spring qui permettra de pointer vers un bean définissant une source de données JDBC (**JDBCDataSource**) ou JNDI (**JNDIDataSource**, gérée par le serveur d'applications)
  - **jpaVendorAdapter** qui précise l'implémentation JPA utilisée.
  - **PersistenceXmlLocation** qui permet de donner le nom du fichier de configuration JPA à utiliser (**/properties/dao/persistence.xml**). Ce dernier donne le nom du gestionnaire d'entités et les classes qui doit gérer.
  - **JpaProperties** qui pointe vers un bean permettant de donner des informations générales liées au comportement de JPA.
- Le bean **JDBCDataSource** définit localement un poll de connexions à la base de données.
- Le bean **JNDIDataSource** permet de pointer vers un poll de connexions à la base de données géré par le serveur d'applications. Il a comme propriétés :
  - **jndi-name** qui correspond au nom du poll dans la configuration JNDI du serveur d'applications. Ici on utilise unevariable Spring.
  - **lookup-on-startup="false"** permet de préciser que l'on ne fera pas la recherche JNDI du poll de connexions au démarrage de l'application. C'est utile car l'on utilise une variable Spring pour savoir si on doit utiliser une source de données JDBC ou JNDI. Aussi il ne faut pas activer systématiquement la recherche alors que la configuration n'y fera peut-être pas référence.
- Le bean **jpaProperties** permet de donner des informations générales liées au comportement de JPA.
- Le bean **daoService** permet de définir la classe qui, dans notre application, contiendra les méthodes d'accès aux données.

Utilisation de plusieurs gestionnaires d'entités.

Si on souhaite accéder à n bases de données il est possible de déclarer n gestionnaires d'entités.

TODO

Mapping avec la base de données

Le fichier de configuration JPA (**/properties/dao/persistence.xml**) contient la liste des classes dont il doit assurer la persistance.

En JPA il est possible de définir se mapping directement sous forme d'annotations dans le code java des classes à persister. C'est la solution préconisée dans le cadre de ESUP-Commons V2.

Pour avoir une information complète sur ces annotations Cf. http://docs.jboss.org/hibernate/annotations/3.5/reference/en/html/entity.html

Voici un extrait de la classe User :

```
@Entity
public class User implements Serializable {

        /**
         * Id of the user.
         */
        @Id
        private String id;

    /**
         * Display Name of the user.
         */
    private String displayName;

    /**
         * True for administrators.
         */
    private boolean admin;

    /**
     * The prefered language.
     */
    private String language;

    /**
     * information recorded during database insert
     * used in esup-example to illustrate open session in view mechanism
     */
    @OneToMany(cascade={CascadeType.ALL})
    private List<Information> informations;
```

Explications :

- L'annotation **@Entity** précise que la Classe est une entité à persister. Par défaut tous les propriétés de la classe seront persistées. Un règle de nommage par défaut (si pas de présence d'une annotation sur la propriété pour une déclaration explicite) sera appliquée afin de déterminer le nom de la colonne en base de données.
- L'annotation **@Id** précise que cette propriété sera utilisée comme clé primaire de la table dans la base de données.
    - Note : En JPA comme en Hibernate il est recommandé d'utiliser une clé physique (celle qui sera utilisée comme clé primaire de la table dans la base de données) différente de la clé métier utilisé par l'application. Il est aussi recommandé de créer des méthodes equals et hcode basée sur la clé métier.
      Ici **User** est un cas particulier où la clé physique est aussi la clé métier dans la mesure où la clé n'est pas calculée (non utilisation de l'annotation **@GeneratedValue** en plus de*@Id*). Néanmoins cette pratique reste déconseillée car si nous devions faire évoluer notre besoin métier (par exemple utiliser une autre propriété de la classe -en plus de **id**- en tant que clé métier) nous n'arriverions pas à le faire (la contrainte d'unicité physique portant exclusivement sur **id**)
- L'annotation

  ```
  @OneToMany(cascade={CascadeType.ALL})
  ```

  défini une association un à plusieurs. La cascade permet de préciser que si on fait une opération, typiquement un effacement, sur le père alors elle sera répercutée sur les enfants.

Utilisation de EJB QL

EJB QL est un langage d'interrogation de base de données de *JPA*. Il est orienté objet et a une forme proche du *SQL* mais travaille sur les objets *Java* et pas sur des noms de tables de votre base de données.

Note : Contrairement à *SQL*, le mot clé **SELECT** de *HQL* n'est pas obligatoire. S'il n'est pas utilisé ce sont des objets qui sont retournés. S'il est utilisé, il est possible de seulement retourner les propriétés de ces objets.

Après le mot-clé **FROM**, on n'utilise pas des noms de tables mais des noms de classes. Ce nom de classe est sensible à la casse comme en *Java*. Le nom du *package* n'est pas obligatoire car *Hibernate* a un mécanisme d'*auto-import*.

Exemple d'une requête *EJB QL* simple qui récupère toutes les instances de la classe **User** de la base de données :

```
"SELECT user FROM User user"
```

Pour une information complète sur l'utilisation de *EJB QL* Cf. http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/EJBQL.html