JPERSISTENCETOOLS

MODULE GENERIQUE DE GESTION DE LA COUCHE D'ACCESS AUX DONNEES

Par **Jean-Jacques ETUNE NGI**

Dans le cadre d'un projet Open Source personnel de mise en place d'un outil permettant de s'affranchir de la couche d'accès aux données (DAO)

Date de rédaction : 13 Juillet 2011

Ce tutoriel a pour but de présenter l'outil de générique de gestion de la couche d'accès aux données *[JPersistenceTools]*. Ce document vous présentera donc le contexte dans lequel est né cet outil ainsi que son installation, sa configuration et son utilisation classique et avancée.

Table des matières

Introduction

Dans le monde du génie logiciel, de plus en plus d'équipes *(sinon toutes)* utilisent des outils de mapping **ORM** pour mettre en place leur couche d'accès aux données et plus précisément, dans le cadre des applications ***JAVA (J2SE ou J2EE/JEE5)***, le tandem ***JPA/HIBERNATE*** est très souvent utilisé.

Par ailleurs, les bonnes pratiques de développement exigent de développer une couche d'accès aux données par module *(dans le but d'augmenter la clarté, la testabilité et la facilité de maintenance localisée des modules)*, ce qui, dans le cadre d'un projet de grande envergure, pourra rendre non négligeable la charge de travail liée à la mise en place des modules DAO *(qui doivent aussi être testés)* de l'application cible, non pas du fait d'une difficulté insurmontable dans l'écriture d'une DAO, mais surtout du fait de la réplication des codes liés aux fonctionnalités attendues d'une DAO. En effet, dans le cadre des opérations de CRUD, et principalement en ce qui concerne la création, la mise à jour et la suppression, la logique d'implémentation est la même:

- *Vérification des Contraintes d'intégrité*
- *Vérification des Contraintes référentielles*
- *Opération proprement dite (Création, Mise à Jour, Suppression)*
- *Vérification des post-conditions*

Pour ce qui concerne les méthodes de filtre, dans le cas général, il s'agit de filtrer des entités persistantes en spécifiant des restrictions sur les propriétés simples et relationnelles de celles-ci.

C'est donc dans ce contexte que l'outil générique d'accès aux données s'inscrit. Son but est tout d'augmenter la productivité des équipes de développement en offrant la possibilité d'exprimer les contraintes précédentes de façon décorative et en prenant sur lui, de les vérifier automatiquement. De cette façon, le temps d'écriture de la couche DAO sera réduit de façon significative le temps de développement des DAO permettant aux développeurs de se concentrer sur les problèmes métier.

La problématique détaillée

Avez-vous déjà, comme moi, remarqué la répétition de code et de logique liée à l'écriture d'une DAO ?

En effet, il est tout à fait fréquent, par exemple lors de l'écriture d'une méthode de mise à jour d'un objet persistant, de vérifier avant tout que l'objet existe, ou encore qu'il est dans un état donnée ou plus généralement qu'il vérifie certaines de conditions pour que l'opération de persistance puisse être effectuée. Dans le cas contraire *(Si l'objet ne vérifie pas les conditions requises, une exception applicative est levée pour mentionner cette violation de contrainte)*

Prenons le cas d'un réseau d'entreprises *(Entreprise, Filiales, Agences, etc…)*, et choisissons de le modéliser sous la forme d'une structure hiérarchique *(arbre)* dans laquelle l'élément de base est le Nœud et a une référence sur son nœud parent. Un nœud peut donc être une Filiale, Une Agence, ou d'un autre type. Lors de la mise à jour d'un nœud, un certain nombre de contraintes pourraient être vérifiées :

- *L'objet existe-t-il ?*
- *Le nouveau code qui lui a été affecté est-il déjà attribué à un autre Nœud ?*
- *Le nœud est-il es état d'être modifié ?*
- *Le nœud a-t-il dans sa hiérarchie de fils un Nœud qui est en réalité son parent ?*

Quelque soit le lieu où sont inscrites ces vérifications *(au niveau DAO ou Métier)*, il n'en reste pas moins que du code sera écrit pour vérifier ces contraintes, et du code similaire sera certainement écrit pour vérifier d'autre règles concernant d'autres types d'objets persistants dans le même module et/ou dans des modules différents.

Par conséquent, le code d'une méthode DAO *(enregistrement, modification, suppression)* se présente souvent comme celui-ci :

```
…

Public EntityClass daoMethod(EntityClass entity) {

        // Démarrage de la Transaction (Si on n'est pas en environnement managé)

        // Vérification des règles DAO en pré-condition

        // Opération DAO

        // Vérification des règles DAO en post conditions (s'il y en a)

        // Validation ou annulation de la transaction (Si on n'est pas en environnement managé)

        // On retourne l'objet après l'opération DAO (Si on n'est pas en delete)

}
```

Ce squelette de code est donc répété de méthode en méthode, engageant alors beaucoup de temps, d'espace et de possibilité d'erreur dans le projet. Avec le Framework JPersistenceTools, vous n'aurez plus à écrire de méthode DAO, tout ce que vous aurez à faire, sera d'exprimer ces contraintes sous forme d'annotations *(à la façon JPA ou Hibernate Validator)* au-dessus de la classe de l'objet persistant sur lequel porte la méthode DAO et le Framework se chargera de les vérifier pour vous.

Présentation de JPersistenceTools

JPersistenceTools est comme présenté dans l'introduction, un outil générique d'accès aux données basé sur le standard JPA et l'implémentation HIBERNATE. C'est un projet MAVEN 3, Open Source et dont le REPOSITORY est hébergé sur le serveur de source GITHUB à l'adresse *(en lecture seule)* git://github.com/yashiro/JPersistenceTools.git.

JPersistenceTools peut se décomposer de la manière suivante:

1. Une API

   Cette API comprends un ensemble d'annotations représentent des marqueurs que vous utiliserez pour exprimer les contraintes à respecter et une interface **`IJPAGenericDAO`** déclarant les méthodes de base requises pour une DAO et dont l'interface de vos DAO devront hériter.

2. Un moteur d'évaluation des règles

   Ce moteur est tout simplement constitué de l'ensemble des classes implémentant la logique de gestion des marqueurs de l'API et d'une implémentation de l'interface **`IJPAGenericDAO`** nommée **`JPAGenericDAORulesBased`** et permettant, lors des opérations DAO, de prendre en compte les règles exprimées sur les Objets persistants.

NB : *Il est à noter que de la même manière qu'Hibernate Validator, il est possible d'enrichir l'ensemble des annotations fournis par défaut par le Framework.*

a. Annotations disponibles

Le Framework JPersistenceTools est livré avec un certains nombre d'annotation permettant l'expression de vos contraintes et dont voici quelques unes des plus importantes:

- `@SizeValidator`

  C'est une annotation permettant d'évaluer une expression (HQL, JPQL, EJBQL) et de vérifier que le nombre d'enregistrement retournés est compris dans in intervalle que spécifié par vous, sinon elle retourne une exception avec un message que vous indiquerez. Ses paramètres sont les suivants :
  - `type`, représentant le type d'expression (HQL, JPQL, EJBQL)
  - `expr` : représentant l'expression à évaluer écrite dans le type d'expression précédent
  - `min` : représentant le nombre minimum d'enregistrement (par défaut 0)
  - `max` : représentant le nombre maximum d'enregistrement (par défaut Long.MAX_VALUE)
  - `message` : représentant le message à envoyer en cas de règle non vérifiée
  - `parameters` : représentant un tableau de paramètre de message (utilisée dans le cadre de l'internationalisation)

- `mode` : représentant l'ensemble des types d'opérations pendant lesquelles la règle sera évaluée (SAVE, UPDATE, DELETE)
- `evaluationTime` : représentant les moments d'évaluation de la règle (PRE_CONDITION, POST_CONDITION)

- `@SizeValidators`
  C'est une annotation permettant de regrouper un tableau d'annotations de type `@SizeValidator`. Son seul paramètre est le suivant :
  - `value` : représentant un tableau de `@SizeValidator`

- `@NotEmptyDAOValidator`
  Possédant les même propriétés que la `@SizeValidator`, cette annotation est un raccourcis de cette dernière dans laquelle la valeur de min est 1.

- `@NotEmptyDAOValidators`
  C'est une annotation permettant de regrouper un tableau d'annotations `@NotEmptyDAOValidator` Son seul paramètre est le suivant :
  - `value` : représentant un tableau de `@NotEmptyDAOValidator`

- `@IdentityValidator`
  C'est une annotation qui ne fait rien du tout. Elle est principalement utilisée par le moteur pour gérer l'homogénéité des traitements.

b. Méthodes DAO disponibles

Les méthodes disponibles par défaut sont les suivantes :

- **`save`** : *Permettant d'enregistrer un objet persistant.*
- **`saveListResult`** : *Permettant d'enregistrer une liste d'objets persistants et retournant la liste des objets enregistrés.*
- **`update`** : *Permettant de mettre à jour un objet persistant.*
- **`delete`** : *Permettant de supprimer un objet persistant.*
- **`deleteList`** : *Permettant de supprimer une liste d'objets persistants et retournant la liste des objets non supprimés.*
- **`clean`** : *Permettant de vider la table mappée sur une classe persistante donnée.*
- **`filter`** : *Permettant de filtrer les données d'une entité suivant des critères divers et variés exprimés par le biais de différent conteneurs et en spécifiant aussi les modes de chargement des propriétés de chacun des éléments de la liste filtrée.*
- **`findByPrimaryKey`** : *permettant de rechercher une entité grâce à sa clé tout en spécifiant les propriétés de cette entité à charger immédiatement.*
- **`getEntityManager`** *: Permettant d'obtenir le gestionnaire d'entités.*

NB : *Pour plus d'informations sur les annotations et les méthodes disponibles, veuillez vous reporter à la Javadoc du Framework. De plus, les expressions peuvent référencer les propriétés de l'objet sur lequel sera évaluée la règle et ceci en mettant ces propriétés entre parenthèses comme par exemple*

expr= " from EntityClass c where c.nom= ${prop.champ.souschamp}"

Installation de JPersistenceTools

Comme annoncé dans la section précédente, JPersistenceTools se présente comme un projet MAVEN 3, Open Source disponible en lecture seule sur le REPOSITORY GITHUB à l'adresse suivante : git://github.com/yashiro/JPersistenceTools.git. Pour installer ce module dans votre application il faut :

- Tirer le code du REPOSITORY distant en utilisant la commande

  ```
  git clone git://github.com/yashiro.JPersistenceTools.git
  ```

- Vous positionner dans le répertoire JPersistenceTools via `cd JPersistenceTools`
- Lancer la commande suivante : `mvn clean install javadoc:jar`.

(Bien sûr pour ce faire, vous devez avoir installé sur votre poste le gestionnaire de version GIT et l'outil MAVEN 3). Une fois effectuées, vous trouverez dans le répertoire ***`target`*** les archives **`JPersistenceTools-2.0.jar`** et **`JPersistenceTools-2.0-javadoc.jar:`** vous les ajouterez à votre ***classpath.***

Intégration de JPersistenceTools

L'intégration de JPersistenceTools de fait en 3 étapes :

- **Exprimer les règles DAO**

  Ceci se fera par décoration des classes persistantes à l'aide d'annotations de validation

- **Faire Hériter l'interface de votre DAO de l'interface `IJPAGenericDAO`**

  Pour chacune des DAO que vous aurez besoin d'écrire, il faudra faire hériter l'interface de celle-ci de l'interface `IJPAGenericDAO`

- **Faire hériter l'implémentation de votre DAO de la classe `JPAGenericDAORuleBased`**

  L'implémentation de chacune de vos DAO devra hériter de la classe abstraire et implémenter la méthode `getEntityManager` indiquant au moteur comment obtenir le gestionnaire d'entités pour effectuer vos opération persistantes.

C'est tout! Vous n'avez aucune méthode a écrire, vos DAO sont prêtes à l'emploi et disposent de tout ce qui est attendu d'un DAO. Par ailleurs, et ceci dans des cas très spécifiques, vous pouvez toujours enrichir vos DAO par des méthodes personnelles tout en utilisant comme base les fonctionnalités déjà disponibles du fait des différents héritages.

Utilisation avancée de JPersistenceTools

Dans des cas spécifiques, il peut arriver que les annotations prévues par le Framework ne suffisent pas à exprimer votre règle. Deux solutions existent pour gérer ce problème.

a. Gérer manuellement cette contrainte

Il s'agit ici tout simplement, d'obtenir le gestionnaire d'entités via la méthode getEntityManager de votre DAO et de l'utiliser pour gérer manuellement ce cas spécifique.

b. Etendre le jeu d'annotation par vos propres annotations

Dans le cas où cette règle pourrait être réutilisée sur d'autres objets, vous pouvez créer une nouvelle annotation et y encapsuler la logique de traitement de la règle. Celle-ci sera automatiquement reconnue par le Framework et évaluée lors des opérations DAO.
Pour ajouter une annotation DAO au Framework, voici les étapes à suivre :

1. Créer une annotation et l'annoter par
   - `@Target(ElementType.TYPE)` : *Pour qu'elle s'applique aux classes*
   - `@Retention(RetentionPolicy.RUNTIME)` : *Pour qu'elle soit présente à l'exécution*
   - `@Documented`
   - `@Inherited` : *pour que les classes filles de la classe sur laquelle elle sera appliquée voient cette annotation*
   - `@DAOValidatorRule` : *Pour spécifier la classe implémentant la règle de validation de cette annotation. C'est une classe qui doit implémenter l'interface IDAOValidator<? extends Annotation>*
2. Implémenter la classe de gestion de cette règle

   C'est cette classe qui est spécifiée comme valeur de l'annotation `@DAOValidatorRule`

Une fois ces deux étapes effectuées, vous pouvez annoter les classes en utilisant votre annotation.

Quelques exemples

Des exemples d'utilisation de cette DAO sont disponibles à l'adresse

git://github.com/yashiro/JPersistenceToolsSamples.git