JPersistenceTools

MODULE GENERIQUE DE GESTION DE LA COUCHE D'ACCES 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 générale 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`** fournissant un ensemble de fonctionnalité attendues d'une DAO 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.*

Présentation détaillée de l'API de JPersistenceTools

Dans cette partie, nous nous efforcerons de vous présenter de façon plus détaillée l'API exposée par le Framework et permettant d'interagir avec celui-ci.

1. Configuration des règles DAO

Cette configuration consiste à exprimer les contraintes que vous aimeriez que le Framework vérifie pour vous lors des opérations d'enregistrements, de mise à jour ou de suppression. Pour cela, un ensemble d'annotations sont fournies par défaut avec la plateforme pour vous y aider.

- **`@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 un intervalle spécifié par vous, sinon le moteur retourne une exception avec un message que vous indiquerez. Les paramètres de cette annotation sont les suivants :

  - **`type`**, représentant le type d'expression *(HQL, JPQL, EJBQL).* C'est une énumération de type **`ValidatorExpressionType`**
  - **`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).* C'est une énumération de type **`DAOMode`**
  - **`evaluationTime :`** représentant les moments d'évaluation de la règle *(PRE_CONDITION, POST_CONDITION).* C'est une énumération de type **`DAOValidatorEvaluationTime`**

  NB : Notez aussi que *les expressions peuvent référencer les propriétés de l'objet sur lequel sera évaluée la règle et ceci en se fait en utilisant une expression de type STRUTS ou JSP : ${attributNiveau0.attributNiveau1.attributNiveau2}. Par exemple supposons que vous vouliez vérifier que le code du véhicule d'un employé existe déjà en BD, vous auriez une expression du genre :*

```
expr="from Employe e where e.vehicule.code= ${vehicule.code}"
```

- **`@SizeValidators`**

  C'est une annotation permettant de regrouper un ensemble d'annotations de type `@SizeValidator.`
  Le but ici étant de pouvoir exprimer plusieurs règles de type `@SizeValidator` sur le même objet, permettant ainsi de paramétrer de façon complète les règles DAO liées à celui-ci.
  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 ensemble d'annotations `@NotEmptyDAOValidator,` l'objectif poursuivit étant le même que pour l'annotation `@SizeValidators.`
  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.

2. **Enregistrement des objets**

Ceci se fait par le biais de la méthode **`save`** proposée par le Framework, de signature complète **`public <T extends Object> T save(T entity)`**. Lorsque cette méthode est invoquée sur un objet, le moteur lancera dans un premier temps et dans le cas où l'objet n'est pas nul, la validation des règles DAO en pré-conditions *(toutes les règles dont le paramètre evaluationTime contient au moins la valeur PRE_CONDITION et dont le paramètre mode contient au moins la valeur SAVE)*. Dans le cas où les règles en pré-conditions sont vérifiées, le moteur tentera d'effectuer *(Dans la même transaction)* l'Opération d'enregistrement. Si celle si se passe bien, le moteur lancera enfin une validation des règles en post-condition *(toutes les règles dont le paramètre evaluationTime contient au moins la valeur POST_CONDITION et dont le paramètre mode contient au moins la valeur SAVE)*. Si une exception se produit lors de l'une de ces phases, elle sera remontée, sinon l'objet enregistré est retourné.

NB : Le Framework offre aussi une méthode d'enregistrement en batch **`saveList`** permettant d'enregistrer une liste d'objet tout en vous retournant un objet **`SaveListResult`** contenant la liste des objets enregistrés ainsi que une **`MAP`** contenant l'exception levée pour chaque objet non enregistré. Vous pouvez bien entendu demander un **`rollback`** lorsqu'une exception survient.

3. **Mise à jour des objets**

Ceci se fait par le biais de la méthode **`update`** proposée par le Framework, de signature complète **`public <T extends Object> T update(T entity)`**. Lorsque cette méthode est invoquée sur un objet, le moteur lancera dans un premier temps et dans le cas où l'objet n'est pas nul, la validation des règles DAO en pré-conditions *(toutes les règles dont le paramètre evaluationTime contient au moins la valeur PRE_CONDITION et dont le paramètre mode contient au moins la valeur UPDATE)*. Dans le cas où les règles en pré-conditions sont vérifiées, le moteur tentera d'effectuer *(Dans la même transaction)* l'Opération de mise à jour. Si celle si se passe bien, le moteur lancera enfin une validation des règles en post-condition *(toutes les règles dont le paramètre evaluationTime contient au moins la valeur POST_CONDITION et dont le paramètre mode contient au moins la valeur UPDATE)*. Si une exception se produit lors de l'une de ces phases, elle sera remontée, sinon l'objet mis à jour est retourné.

4. **Suppression des objets**

Ceci se fait par le biais de la méthode **`delete`** proposée par le Framework, de signature complète **`public <T extends Object> void delete(T entity)`**. Lorsque cette méthode est invoquée sur un objet, le moteur lancera dans un premier temps et dans le cas où l'objet n'est pas nul, la validation des règles DAO en pré-conditions *(toutes les règles dont le paramètre evaluationTime contient au moins la valeur PRE_CONDITION et dont le paramètre mode contient au moins la valeur DELETE)*. Dans le cas où les règles en pré-conditions sont vérifiées, le moteur tentera d'effectuer *(Dans la même transaction)* l'Opération de suppression. Si celle si se passe bien, le moteur lancera enfin une validation des règles en post-condition *(toutes les règles dont le paramètre evaluationTime contient au moins la valeur POST_CONDITION et dont le paramètre mode contient au moins la valeur DELETE)*. Si une exception se produit lors de l'une de ces phases, elle sera remontée.

NB : De même que l'enregistrement, le Framework dispose d'une méthode de suppression en batch : **`deleteList`**, permettant de supprimer une liste d'objets et retournant la liste des Objets non supprimés. De la même façon, vous pouvez spécifier le rollback au cas où une exception surviendrait.

5. **Recherche des Objets**

Sans doute l'opération présentant le plus de variante dans une application, l'implémentation fournie ici permet d'exprimer un maximum de restrictions sur les données recherchées tout en permettant au développeur de gérer le chargement des propriétés ainsi que le nombre de ligne maximum à retourner et bien sur, la position du premier enregistrement. Ceci se fera par le biais de la méthode

```
public <T extends Object> List<T> filter (Class<T> entityClass,
AliasesContainer alias, RestrictionsContainer restrictions,
OrderContainer orders, LoaderModeContainer loaderModes,
int firstResult, int maxResult)
```

Détaillons un tout petit peu ses paramètres :

- ***entityClass***

  Il s'agit tout simplement de la classe persistante sur laquelle porte la recherche.

- ***alias***

  Le Framework se basant sur l'API Criteria offerte par Hibernate, le paramètre **`alias`**, de type **`AliasesContainer`** permet de renseigner tous les alias de propriétés qui pourront être utilisées dans la gestion des restrictions, des chargements ou encore de l'ordre de rangement des résultats du filtre.
  Cette propriété est particulièrement utilisée lorsque le développeur voudrait interroger un attribut d'une propriété relationnelle, par exemple, on peu rechercher uniquement les Employés possédants un véhicule de couleur rouge: pour y arriver, il nous faudra effectuer une restriction sur l'attribut **`couleur`** de la propriété relationnelle **`véhicule`** de l'objet **`employé,`** sur lequel portera notre filtre**`.`** Pour que ça marche, il faudra créer un alias *(vehiculeAlias)* sur la propriété véhicule et par la suite créer une restriction en utilisant cet alias (*vehiculeAlias.couleur = 'rouge'*).

  Obtenir un conteneur d'alias est très simple, il suffit d'invoquer la méthode **`getInstance()`** de la classe **`AliasesContainer`**. Pour le reste, vous pourrez ajouter tous les alias que vous voulez par le biais de la méthode **`add()`** de l'instance obtenue par la méthode **`getInstance().`**

  Pour illustrer l'exemple précédent voici un bout de code pour le mettre en œuvre

```
AliasesContainer aliasesCont = AliasesContainer.getInstance() ;
aliasesCont.add("vehicule","vehiculeAlias");

RestrictionsContainer restrictionsCont = RestrictionsContainer.getInstance() ;
restrictionsCont.add(Restrictions.eq("vehiculeAlias.couleur", "rouge"));
```

- ***restrictions***

  Ce paramètre sera sans doute votre meilleur compagnon lors de l'utilisation du Framework, il sert principalement à exprimer toutes les restrictions nécessaire à votre filtre et par rapport aux propriétés de la classe cible du filtre ***`(age > 10, sexe== Sexe.MASCULIN, etat=Etat.MARIE, etc.)`***. L'expression des restrictions se fait en utilisant tout ce que l'API Criteria propose. Par exemple, la classe ***`Restrictions`*** fournie par Hibernate pour la construction des restrictions.

  Obtenir un conteneur de restrictions est très simple, il suffit d'invoquer la méthode **`getInstance()`** de la classe **`RestrictionsContainer`**. Pour le reste, vous pourrez ajouter toutes vos restrictions par le biais de la méthode **`add()`** de l'instance obtenue par la méthode **`getInstance().`**

- orders

  Ce paramètre vous servira à spécifier l'ordre dans lequel les objets filtrés doivent être retournés *(C'est la clause **`order by`** des requêtes HQL)*
  Le conteneur d'ordre s'obtient et s'utilise de la même façon que les deux premiers.

- ***loaderModes***

  Ce paramètre vous permettra de contrôler les modes de chargement des propriétés des objets filtrés. En effet, vous pourrez donc demander qu'au chargement des objets, certaines propriétés soient chargées immédiatement tandis que d'autres pas. Si aucun conteneur de mode de chargement n'est spécifié ou si le mode de chargement d'une propriété n'est pas spécifié dans le conteneur d'ordre, le Framework le chargera en utilisant les propriétés du mapping par défaut c'est-à-dire, le mode de chargement spécifié lors de la configuration du pont ORM via JPA ou Hibernate.
  Pour le reste, l'obtention du conteneur de modes de chargement se fait de la même façon que les autres et son utilisation aussi.
  Il est à noter que la méthode d'ajout de mode prend en paramètre le nom de la propriété à configurer et le mode de chargement de celle-ci, qui est de type `org.hibernate.FetchMode.`

- ***firstResult***

  Ce paramètre permet de spécifier le numéro de l'index du premier enregistrement ;

- ***maxResult***

  Ce paramètre permet de spécifier le nombre maximum d'enregistrement à retourner.

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 ce Framework *(Configuration des règles, implémentation de la DAO et utilisation des fonctionnalités hérités du Framework)* sont disponibles dans le module de test contenu dans les sources du Framework.