Cours Spring Data

1 Présentation Générale

- Spring Data est un robuste framework de l'écosystème de Spring qui se compose de nombreux modules offrant d'accéder des solutions de données alternatives comme :

  - GemFire : Cache distribué.
  - MongoDB : Base NoSQL, orienté « document »
  - Neo4J : Base structurée autour de structures de Graphes.
  - Redis : Système de données par « clé/valeur »
  - Solr : Plateforme s'appuyant sur le moteur de recherche Lucène.
  - et JPA...

- Spring Data JPA met en place une surcouche d'accès à JPA, et fournit donc ainsi un ensemble cohérent de fonctionnalités avancées sur lesquels s'appuyer pour bâtir et consolider ses applications.

  http://projects.spring.io/spring-data-jpa/
  http://docs.spring.io/spring-data/jpa/docs/1.4.3.RELEASE/reference/html/

- Les principaux éléments offerts par Spring Data JPA sont les suivants :

  - L'aide à la construction de *Repository* Spring
    Spring Data JPA va automatiser l'instanciation des services DAO (annotés *@Repository* en Spring), en se basant sur la simple définition de leur interfaces.

    Les DAO ne sont donc plus des classes concrètes, mais des instances de *proxy AOP*, chargés d'intercepter des appels de DAO, pour les convertir en JPA.

  - La gestion de *Query* JPA « *Type Safe* »
    Une vérification des Types objets opérés par les *Query* est effectuée à la phase de compilation, en utilisant les *Meta-Data JPA* (fichiers avec le nom du modèle préfixés d'un '_') qui contiennent la définition des structures détenues par les classes d'Entités JPA (propriétés simples associées aux valeurs littérales, et relations *singular* ou *multiple*).

    Ces Meta-Data sont un apport très important à la structuration d'informations détenues sur des classes, leur mise en place a constitué une évolution importante par rapport au noyau réflexif de Java, dont une des dernières réelles évolution fût l'introduction des *Generics* en Java 5.

  - Transactionnel
    Pas de modification réelle, Spring Data JPA repose sur l'utilisation des @nnotations *@Transactionnal* de Spring Transaction, vues précédemment.

- Gestion de l'Audit des entités
  Observe (Design Pattern « *Observer* », famille « *Comportement* » les créations et modifications d'entités, et stocke les dates et identités utilisatrices associées, de façon dynamique et simplement configurable.

  La mise en place est très simple, elle passe par l'utilisation des @nnotations *@CreatedBy, @ModifiedBy*, et *@CreatedDate, @ModifiedDate, (*qui peuvent être placées, dans un mécanisme d'héritage pour les entités, sur les classes « hautes » du model)

  Elle nécessite aussi la mise en place d'une classe implémentant *AuditorAware* (à déclarer dans la configuration du Context), l'implémentation par défaut se basant sur l'intégration de Spring Security (ici, exemple de la puissance d'interaction entre les différents modules composant la plateforme Spring).

  Remarque : les processus transverses d'audit peuvent être écoutés par le biais d'un *Listener* à implémenter et référencer.

- Pagination/ Tri
  Spring Data JPA, de par sa surcouche JPA, permet de résoudre simplement ce problème classique (ex : charger la 3ème page d'un volume de 15 éléments), des accès de *DataModel* associés aux tableaux (en JSF comme d'autres technologies).

  De même, Spring Data JPA offre des primitives afin de gérer correctement et simplement les mécanismes de requêtes utilisant le *tri* sur des propriétés.

- Intégration de la technologie *QueryDsl*
  Cette technologie offre une surcouche facilitant la construction de *Query* JPA. Elle toutefois quelque peu redondante avec celle des *Specifications* !
  Son utilisation est laissée à la libre appréciation des développeurs.

- Configuration
  La mise en place de la configuration Spring Data JPA dans le *Contexte* Spring s'effectue de façon simple et offre des options sur les stratégies/politiques de constructions d'objets de type *Repository* :

  Héritage depuis une classe racine, par ex, qui permet d'avoir des DAO possédant un niveau fonctionnel étendu en terme de service de persistance, que l'on va ensuite « décorer » (Design Pattern « *Decorator* » famille « *Structuration* ») en définissant des signatures de méthodes supplémentaires dans l'interface, ces dernières interceptées et interprétées par le Proxy AOP, en charge de la gestion du *Repository*.

- Lock
  Spring Data JPA offre une @nnotation *@Lock(LockModeType)* pour les méthodes finders.

- Coté JPA, Spring Data offre plusieurs interfaces dont les interfaces des *Repository* peuvent hériter, afin de disposer automatiquement lors de leur génération des implémentations fonctionnelles :

    - *CrudRepository<T, ID extends Serializable>*

      Point d'entrée des interfaces de services DAO.
      CrudRepository est définie par des templates *Generics,* prototypant dans sa signature de classe :

        - La définition d'un Type <*T*> associé à une classe de modèle (elle même étant par ailleurs une *Entity* JPA, une classe déclarant @*Entity*).
        - Et un type <*ID*>, permettant de définir le type de la clé primaire associée à l'entité, soit un *Number (Integer, Long)*, soit pour une *Clé Composée* au travers d'une classe dédiée, dans les deux cas, le type est forcément Serializable (contrainte imposée).

      CrudRepository fournit une implémentation standard et par défaut pour les méthodes usuelles suivantes : *save, findOne, findAll, count, delete, exists, ...*
      Permettant ainsi de disposer de méthodes principales éprouvées dans le cadre des service DAO (inutile de les ré-implémenter)

    - *PagingAndSortingRepository<T, ID extends Serializable>*
      Hérite de l'interface *CrudRepository*, en suivant la même définition de templating *Generics*, et fournit des méthodes supplémentaires pour le tri *findAll(Sort)*, et la pagination findAll(Pageable).

      Rappel : Les pages, comme les tableaux, ont un index qui démarre à 0.

    - *JpaRepository<T, ID extends Serializable>*
      Hérite de l'interface *PagingAndSortingRepository*, en lui ajoutant des méthodes spécifiques :

        - Au suppression par *Batch*, c'est à dire l'exécution dans une seule requête de plusieurs traitements.
        - Des *findAll* supplémentaires, avec des listes d'ids.
        - Des méthodes relatives au *flush*, c'est à dire la politique de forcer l'*EntityManager* JPA à exécuter l'ensemble des requêtes associées aux objets qu'il détient.

      Cette interface, la plus avancée, regroupe l'ensemble de toutes les fonctionnalités proposées par Spring Data JPA.

      Elle sera héritée par notre interface principale de *Repository* : *IRepositoryDAO* (parmi d'autres liens d'héritage, puisque les interfaces sont le seul cas d'héritage multiple possible en Java, Cf TP).

- Spring Data met en place des *Query*, générant automatiquement des implémentations de *Finder* basées sur les signatures de méthodes et une nomenclature de propriétés.

  Cf TP : L'interface *IProductRepository* définit et les méthodes (générées ensuite) :

  - *findByPrice*
    Recherche sur la propriété *Product.price*
  - *findByDesignationLike*
    Recherche sur la propriété *Product.designation* tout en opérant avec une politique de recherche SQL *Like*.
  - *findByGenreNom*

    Recherche sur la propriété Product.*genre.nom* : en se servant donc de la relation détenue par la classe Product.

  Les operateurs disponibles de *Query* sont les suivants:

• *By*	Opérateur principale actant le Where SQL
• *Dinstinct*	Spécifie un Elément unique
• *Or, And*	Critère d'agrégation ou d'exclusion
• *OrderBy, Asc, Desc*	Acte le tri par propriété et politique Asc/Desc
• *LessThan, GreaterThan*	Règles numériques
• *Between, After, Before*	Règle pour des dates
• *Like, NotLike*	Like Sql
• *IsNull, isNotNull, NotNul*	Opérateurs sur valeurs *null*
• *Not*	Différent de
• *In, NotIn*	Est contenu, ou pas, dans la liste de valeurs
• *True, False*	Opérateurs sur valeurs booléennes
• *IgnoreCase, AllIgnore* *StartingWith, EndingWith, Containing*	Opérateurs sur chaines de caractères

  Remarques :

  - Si le découpage entre relation et propriété ne fonctionne pas correctement, par exemple parce que des noms sont mélangés entre objets et valeurs littérales, le Framework met à disposition un opérateur de séparation : le caractère '_', à inclure dans le nom de la méthode afin de les distinguer correctement.
  - Les méthodes peuvent évidemment ajouter les paramètres *Pageable* et *Sort*.
  - On peut spécifier ses propres *Query* JPQL directement dans le corps de l'interface, en redéfinissant l'implémentation standard par une @nnotation *@Query(« select... »).*

    On peut utiliser des *NamedParameters* (infos à reporter dans la signature).

    Pour des *Query* SQL il faut mentionner le paramètre « *nativeQuery=true ».*

    On peut aussi utiliser des expressions SpEL (#{...}) dans les *Query*.

- Spring Data fournit de surcroît des « *Helpers Classes* » pour la constructions de *Criteria JPA*: les *Specifications*

  Pour les utiliser il faut d'abord câbler ce nouveau comportement sur notre interface principale, sa signature devient (Cf TP)

```
IRepositoryDAO<T, ID extends Serializable>
    extends IDAO<T>, JpaRepository<T, ID>,
            JpaSpecificationExecutor<T>
```

  Cette interface *JpaSpecificationExecutor* permet d'ajouter les méthodes suivantes à nos *Repositories* (instances de service DAO) :

  - *count(Specification<T>)*
  - *findAll(Specification<T>)*
  - *findAll(Specification<T>, Pageable)*
  - *findAll(Specification<T>, Sort)*
  - *findOne(Specification<T>)*

  Les Spécifications permettent, par l'instanciation d' « *Inner-Class* » et d'un mécanisme de *callback*, de manipuler plus simplement la construction de *Predicate JPA*, et d'avoir en sortie un code plus propre.

  ex TP : La classe *ProductByGenreSpecificationBuilder* fournit une méthode *static*, *buildProductByGenreSpec,* qui construit une instance de *Specification* en redéfinissant la méthode *toPredicate*.

  Elle prend en paramètre les éléments JPA (*Root, CriteriaQuery, CriteriaBuilder*) que l'on doit usuellement manipuler pour construire des *Predicate* (cette partie nous est évitée par l'emploi des *Specifications,* ce qui est très appréciable et évite d'avoir toute une partie de code redondant dans nos DAO, CF TP JPA : *ProductDAOJPA*.getList :

```
CriteriaBuilder criteriaBuilder   = getEntityManager().getCriteriaBuilder();
CriteriaQuery criteriaQuery       = criteriaBuilder.createQuery();
Root<Product> fromProduct         = criteriaQuery.from(Product.class);
```

  Ici la construction est externalisée dans un package/des classes de *SpecificationBuilder* dédiées, ce qui améliore le design, et la construction de notre *Predicate* de recherche se résume (en dehors de la structure d' « *Inner-Class »*) à :

```
return cb.equal(root.get(Product_.genre).get("nom"), genreName);
```

  Remarque : on utilise bien ici pour construire le *Predicate* la classe de Méta-donnée JPA *Product_,* qui détient les infos de structures de persistance, et permet de vérifier la cohérence de la requête (équation de *Type*).

- En terme de configuration (XML), l'intégration de Spring Data passe par la simple déclaration de l'élément suivant (Cf TP):

```
<jpa:repositories base-package="fr.shortcircuit.service.dao.repository"
  factory-class="fr.shortcircuit.service.dao.util.CustomRepositoryFactoryBean" />
```

  - Le préfixe « jpa » est arbitraire, il correspond au *NameSpace* déclaré en en-tête de fichier XML de context :

```
xmlns:jpa="http://www.springframework.org/schema/data/jpa"
```

  - La propriété « *base-package* » correspond à la définition du nom complet de package dans lequel se trouvent les interfaces de *Repository* qui doivent être ultérieurement scannées par Spring Data.

    On peut compléter cette définition en excluant des packages par l'emploi additionnel du nœud « *exclude-filter* ».

  - La propriété « *factory-class* », (Design Pattern « *Factory* », famille « Création »), très importante correspond à une classe en charge de réaliser l'instanciation des *Repository*.

    Elle permettra notamment de définir une classe racine qui servira comme classe mère d'héritage pour les instances de Repository qui seront proxyfiées.

    Cf Tp : méthode *getTargetRepository* qui renvoie, pour chaque appel de création d'un Repository, une instance d'*AbstractRepositoryDAO*, dans laquelle on a logé des méthodes que l'on veut disponibles pour l'ensemble des instances de *Repository* de l'application.
    (Là encore, les *Generics* sont largement utilisés)

2 Contenu du TP

- Il met en place le Framework Spring Data en migrant le code du TP précédent dédié à JPA, et notamment les points suivants :

    - La nouvelle configuration de Contexte Spring.
    - Les interfaces de *Repository* avec des exemples de finders
    - Un exemple d'utilisation de *Specification.*
    - Le package *fr.shortcircuit.service.dao.util*, dans lequel on retrouve les éléments principaux de haut niveau : *AbstractRepositoryDAO, CustomRepository, IDAO, IRepositoryDAO*
    - Les classes de Meta-Information du Model JPA
    - La configuration Maven, permettant de builder le projet, et de le lancer sur une Target Tomcat7