Spring Data JPA (avancé)

Arnaud Cogoluègnes

Zenika

November 7, 2013

Plan

Plan

Pourquoi JPQL avec les repositories CRUD

- Quand les méthodes CRUD ne suffisent pas
- Quand les méthodes par convention de nommage ne suffisent pas
- Quand une implémentation custom ne se justifie pas
  - Aucun traitement nécessaire...
  - ... juste exécution de la requête et récupération du résultat

@Query

```
public interface ContactRepository extends Repository<Contact,Long> {

  @Query("from Contact c where c.lastname = ?1")
  List<Contact> findByLastname(String lastname);

}
```

Binging des paramètres avec @Param

- Par défaut : paramètres JPQL/Java bindés selon leur index
- @Param permet de binder par nom

```
public interface ContactRepository extends Repository<Contact,Long> {

  @Query(
    "from Contact c "+
    "where c.firstname = :firstname and c.lastname = :lastname"
  )
  Contact findByFirstnameAndLastname(
     @Param("firstname") String firstname,
     @Param("lastname") String lastname);

}
```

Requête nommée

- Directement sur l'entité
- Convention de nommage dans l'interface

```
@Entity
@NamedQuery(
  name = "Contact.findByLastname",
  query = "from Contact c where c.lastname = ?1"
)
public class Contact { (...) }

public interface ContactRepository extends Repository<Contact,Long> {

  List<Contact> findByLastname(String lastname);

}
```

Requête de modification avec @Modifying

```
public interface ContactRepository extends Repository<Contact,Long> {

  @Modifying
  @Query("update Contact c set c.lastname = ?1 where c.lastname = ?2")
  @Transactional
  void setNewLastname(String newLastname,String oldLastname);

}
```

- EntityManager vidé après l'exécution
  - Car des entités peuvent être désynchronisées
  - Attribut clearAutomatically disponible pour choisir
- Ne pas oublier @Transactional !

Plan

Principes

- Spring Data JPA peut trier et paginer des données
- Il suffit de rajouter des paramètres dans les méthodes
- Rajoute de la flexibilité aux méthodes
  - Pas d'instruction de tri en dur dans les requêtes JPQL

Tri avec Sort

- Ajouter un paramètre de type Sort à la méthode

```
public interface ContactRepository extends Repository<Contact,Long> {

  List<Contact> findByLastname(String lastname, Sort sort);

}
```

Tri avec Sort

- Lors de l'appel, spécifier
  - Un ordre avec l'énumération `Direction`, `ASC` ou `DESC`
  - Une ou des propriétés

```
repo.findByLastname("Dalton",new Sort(Direction.DESC,"firstname"));
repo.findByLastname("Dalton",new Sort("firstname")); // ASC par défaut
```

Pagination avec Pageable

- Ajouter un paramètre de type Pageable à la méthode

```
public interface ContactRepository extends Repository<Contact,Long> {

  List<Contact> findByLastname(String lastname, Pageable pageable);

}
```

Pagination avec Pageable

- Utiliser PageRequest comme implémentation de Pageable
- Nécessité de passer un Sort pour des pages cohérentes

```
PageRequest pageable = new PageRequest(0, 10,new Sort("firstname"));
List<Contact> contacts = repo.findByLastname("Dalton",pageable);
```

Récupérer une Page

- Retourner une Page<T>
- Donne des informations sur les données et leur pagination
- Implique une requête count à chaque appel

```
public interface ContactRepository extends Repository<Contact,Long> {

  Page<Contact> findByLastname(String lastname, Pageable pageable);

}
```

Exploiter la Page

- Informations disponibles :
  - Les enregistrements !
  - Numéro de la page
  - Nombre d'éléments total
  - Première page ou dernière page
  - etc.

```
PageRequest pageable = new PageRequest(0, 10,new Sort("firstname"));
Page<Contact> page = repo.findByLastname("Dalton",pageable);
List<Contact> contacts = page.getContent();
if(page.isFirstPage()) { (...) }
```

Sort, Page et Query

- Query peut cohabiter avec Tri et Page

```
public interface ContactRepository extends Repository<Contact,Long> {

  @Query("from Contact c join fetch a.address where c.firstname = ?1")
  List<Contact> findByLastname(String lastname, Sort sort);

}
```

Page et Query

- Possibilité de préciser sa requête de count

```
public interface ContactRepository extends Repository<Contact,Long> {

  @Query(
    value="from Contact c join fetch a.address where c.firstname = ?1",
    countQuery="select count(*) from Contact c where c.firstname = ?1"
  )
  Page<Contact> findByLastname(String lastname, Pageable pageable);

}
```

Plan

Criteria dans JPA 2.0

- Spring JPA 2.0 propose une API Criteria
- Une alternative au JPQL
- Type-safe si le méta-modèle est généré
  - Pas le cas dans les exemples suivants
- Type-safe mais complexe et verbeuse...

Une requête Criteria sur Contact

- Initialisation de la requête

```
CriteriaBuilder builder = em.getCriteriaBuilder();
CriteriaQuery<Contact> query = builder.createQuery(Contact.class);
Root<Contact> root = query.from(Contact.class);
```

Une requête Criteria sur Contact

- Création d'un prédicat
- L'utilisation du méta-modèle peut rendre le prédicat type-safe

```
Predicate outlaws = builder.equal(root.get("lastname"),"Dalton");
```

Une requête Criteria sur Contact

- Exécution de la requête

```
List<Contact> cs = em.createQuery(query.select(root)).getResultList();
```

Combiner les prédicats

```
Predicate p1 = (...);
Predicate p2 = (...);
query.where(builder.and(p1,p2));
```

Une requête Criteria sur Contact

```
CriteriaBuilder builder = em.getCriteriaBuilder();
CriteriaQuery<Contact> query = builder.createQuery(Contact.class);
Root<Contact> root = query.from(Contact.class);

Predicate outlaws = builder.equal(root.get("lastname"),"Dalton");
Predicate p2 = (...);
query.where(builder.and(outlaws,p2));
List<Contact> cs = em.createQuery(query.select(root)).getResultList();
```

Specification de Spring Data JPA

- `Specification` permet
  - la réutilisation des `Predicates`
  - la combinaison des `Predicates`
- Il faut toujours écrire les `Predicates` !
- Elimine une partie du code “boilerplate”
- Pattern Domain Driven Design
  - `http://domaindrivendesign.org/node/87`

Une classe centralise les Specifications

```
package com.zenika.repository;

import org.springframework.data.jpa.domain.Specification;
import com.zenika.model.Contact;

public abstract class ContactSpecs {

  public static Specification<Contact> outlaws() {
    (...)
  }

  public static Specification<Contact> inMidThirties() {
    (...)
  }

}
```

Le repository utilise les Specifications

```
List<Contact> contacts = repo.findAll(outlaws());
```

- Voyons comment faire !

Implémenter une Specification

```
public abstract class ContactSpecs {

  public static Specification<Contact> outlaws() {
    return new Specification<Contact>() {
      @Override
      public Predicate toPredicate(Root<Contact> root,
                CriteriaQuery<?> query, CriteriaBuilder cb) {
        return cb.equal(root.get("lastname"), "Dalton");
      }
    };
  }
}
```

Rendre le repository Specification-friendly

- Etendre l'interface JpaSpecificationExecutor<T>
- Propose les méthodes à base de Specifications

```
public interface ContactRepository
    extends Repository<Contact,Long>,
    JpaSpecificationExecutor<Contact> {


}
```

Utiliser les Specifications

```
import static org.springframework.data.jpa.domain.Specifications.*;
import static com.zenika.repository.ContactSpecs.*;
(...)
// avec import statique de ContactSpecs
List<Contact> contacts = repo.findAll(outlaws());
// avec import statique de Specifications
List<Contact> contacts = repo.findAll(
  where(outlaws()).and(inMidThirties())
);
```

Plan

Rappel : transaction avec Spring

- Spring gère les transactions déclarativement
- Réglage par XML ou par annotation
- Spring propose aussi une API et un `TransactionTemplate`
  - Rarement utilisé dans les applications
  - Pratique quand la démarcation déclarative est insuffisante

Rappel : transaction avec Spring

- Configuration du `PlatformTransactionManager`
- Activation de la configuration par annotation

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

<tx:annotation-driven />
```

Rappel : transaction avec Spring

- D'autres implémentations de `PlatformTransactionManager`
  - Pour Hibernate, JDBC, JTA, etc.
- `transactionManager` est le nom habituel du bean
- Bien ajouter le namespace `tx`

Rappel : transaction avec Spring

- Annoter les classes avec `@Transactional`
  - Possibilité d'annoter les interfaces ou les classes parentes

```
@Transactional
public class ContactServiceImpl implements ContactService {

  public void delete(Contact ... contacts) { ... }

}
```

Rappel : transaction avec Spring

```
contactService.delete(contact1,contact2);
```

- L'appel de méthode est intercepté
- Une transaction est démarrée
- Le code applicatif est appelé
- La transaction est committée
- La méthode retourne
- Si une `RuntimeException` est lancée, la transaction est annulée

Spring Data JPA et les transactions

- Les méthodes des repositories sont transactionnelles
- Les méthodes de sélection sont en lecture seule
  - `findOne`, `findAll`, etc.
- Les méthodes de modification ne sont pas en lecture seule
  - `save`, `delete`, etc.
- Les méthodes ajoutées sont en lecture seule !
  - `@Query`, convention, implémentations custom
- Pour les méthodes ajoutées et modifiant des données, apposer
  - `@Modifying`
  - `@Transactional`

SimpleJpaRepository

- Implémentation utilisée par défaut
- Jetons un coup d'oeil...

```
@org.springframework.stereotype.Repository
@Transactional(readOnly = true)
public class SimpleJpaRepository<T, ID extends Serializable>
        implements JpaRepository<T, ID>, JpaSpecificationExecutor<T> {

  public T findOne(ID id) { (...) }

  @Transactional
  public void deleteAll() { (...) }

  (...)

}
```

Plan

Proxies dans Spring

- Principe très utilisé dans Spring
- Un mécanisme pour implémenter la programmation orientée aspect
- Permet d'ajouter du comportement de façon transparente
- Ex. : begin/commit de transaction, sécurité, logging

Principe d'un proxy (AOP)

- Le proxy implémente la même interface que l'objet cible
- Tous les appels de méthodes sont interceptés
- Du code est exécuté autour de ces méthodes

Proxy JDK

- Spring utilise les proxies JDK
- L'API proxies JDK est simple mais bas niveau
- Spring l'enrichit, notamment via AspectJ
- Ex. : notion de poincut, configuration par annotations, etc.

Créer un proxy JDK

```
ContactRepository repo = (ContactRepository) Proxy.newProxyInstance(
  getClass().getClassLoader(),
  new Class<?>[]{ContactRepository.class},
  invocationHandler
);
```

- Qu'est-ce que l'`InvocationHandler` ?

InvocationHandler

- L'InvocationHandler implémente le comportement

```
InvocationHandler invocationHandler = new InvocationHandler() {
  public Object invoke(Object proxy, Method method, Object[] args)
                                    throws Throwable {
    // comportement avant
    // appelle de la méthode
    Object res = method.invoke(cible,args);
    // comportement après
    return res;
  }
};
ContactRepository repo = (ContactRepository) Proxy.newProxyInstance(..);
repo.findAll();
```

Ajouter une transaction de façon transparente

```
InvocationHandler handler = new InvocationHandler() {
  @Override
  public Object invoke(Object proxy, Method method, Object[] args)
                    throws Throwable {
    TransactionStatus status = tm.getTransaction(
      new DefaultTransactionDefinition()
    );
    try {
      Object res = method.invoke(targetRepo, args);
      tm.commit(status);
      return res;
    } catch(Exception e) {
      tm.rollback(status);
      throw e;
    }
  }
};
```

Proxies dans Spring Data JPA

- Principe légèrement différent des proxies AOP
- Il n'y pas de cible
- Le comportement de la méthode est implémenté à la volée
- Ex. de critères d'implémentation : le nom de la méthode

Proxies dans Spring Data JPA

- Le proxy “route” les appels :
  - Implémentation par défaut pour les méthodes CRUD
  - Implémentation custom pour les méthodes de l’interface custom
  - Implémentation suivant l’annotation apposée
  - Implémentation dynamique pour les méthodes basées sur les conventions
  - etc.

Plan

Querydsl

- API pour la construction de requêtes
- Type-safe si le méta-modèle est généré
  - C'est le cas dans les exemples suivants
- Support pour JPA, Hibernate, JDO, SQL, MongoDB, etc.
- Projet Open Source (`http://www.querydsl.com/`)

Une requête Querydsl sur Contact

```
JPAQuery query = new JPAQuery(entityManager);
QContact contact = QContact.contact; // méta-modèle

List<Contact> = query.from(contact)
  .where(
    contact.lastname.eq("Dalton").and(contact.age.between(30, 40))
  )
  .list(contact);
```

Querydsl vs. Criteria JPA 2.0

- Tous les deux type-safe grâce à la génération du méta-modèle
- Querydsl a fait le choix d'une API courante ("fluent API")
- Tous deux nécessitent un temps d'apprentissage

Predicate

- Querydsl définit une interface `Predicate`
  - Ne pas confondre avec celle de JPA 2.0
- `BooleanExpression` est l'implémentation la plus courante

```
BooleanExpression predicate = QContact.contact.lastname.eq("Dalton");
```

Spring Data JPA et Querydsl

```
package org.springframework.data.querydsl;

public interface QueryDslPredicateExecutor<T> {

  T findOne(Predicate predicate);

  Iterable<T> findAll(Predicate predicate);

  Iterable<T> findAll(Predicate predicate, OrderSpecifier<?>...orders);

  Page<T> findAll(Predicate predicate, Pageable pageable);

  long count(Predicate predicate);
}
```

Rendre le repository `Predicate`-friendly

- Etendre l'interface `QueryDslPredicateExecutor<T>`

```
public interface ContactRepository
    extends Repository<Contact,Long>,
    QueryDslPredicateExecutor<Contact> {

}
```

Pattern spécification pour les Predicates

- Pour la réutilisation/combinaison des Predicates
- Moins justifié qu'avec JPA 2.0
  - Predicates plus simples à construire

```
public abstract class ContactSpecs {

  public static BooleanExpression outlaws() {
    return QContact.contact.lastname.eq("Dalton");
  }

  public static BooleanExpression inMidThirties() { (...) }

}
```

Utiliser les Predicates

```
import static com.zenika.repository.ContactSpecs.*;
(...)
// 1 Predicate
int count = repo.count(outlaws());
// Combinaison de Predicates (utilisation de BooleanExpression)
int count = repo.count(outlaws().and(inMidThirties()));
```

Plan

Problématique

- Savoir qui a créé/modifié une entité
- Savoir quand l'entité a été créée/modifiée
- Spring Data JPA propose d'auditer les entités automatiquement

Déclarer le listener dans /META-INF/orm.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings>

  <persistence-unit-metadata>
    <persistence-unit-defaults>
      <entity-listeners>
        <entity-listener
          class="o.s.data.jpa.domain.support.AuditingEntityListener" />
      </entity-listeners>
    </persistence-unit-defaults>
  </persistence-unit-metadata>

</entity-mappings>
```

Avoir une classe “auditor”

- Généralement le `User` de l'application

```
@Entity
public class User {

  @Id
  @GeneratedValue(strategy = GenerationType.SEQUENCE)
  private Long id;

  private String login;

  (...)
}
```

Les entités à auditer doivent être Auditable

```
public interface Auditable<U, ID extends Serializable>
                 extends Persistable<ID> {

  U getCreatedBy();

  void setCreatedBy(final U createdBy);

  DateTime getCreatedDate();

  void setCreatedDate(final DateTime creationDate);

  U getLastModifiedBy();

  void setLastModifiedBy(final U lastModifiedBy);

  DateTime getLastModifiedDate();

  void setLastModifiedDate(final DateTime lastModifiedDate);
}
```

Rendre une entité Auditable

- Soit ajouter les champs soi-même

```
@Entity
public class Contact implements Auditable<User,Long> {
  (...)
  @ManyToOne
  private U createdBy;

  @Temporal(TemporalType.TIMESTAMP)
  private Date createdDate;

  @ManyToOne
  private U lastModifiedBy;

  @Temporal(TemporalType.TIMESTAMP)
  private Date lastModifiedDate;
  (...)
}
```

Rendre une entité `Auditable`

- Soit utiliser une classe fournie par Spring Data JPA

```
@Entity
public class Contact extends AbstractAuditable<User, Long> {

 (...)

}
```

Fournir l'utilisateur en cours

- Implémenter `AuditorAware`
- Utilise généralement le contexte de sécurité

```
public class SecurityAuditorAware implements AuditorAware<User> {

  @Override
  public User getCurrentAuditor() {
    return SecurityContext.getCurrentUser();
  }

}
```

Activer l'auditing

- Déclarer le bean `AuditorAware`
- L'enregistrer auprès de Spring Data JPA

```
<bean id="auditorAware"
      class="com.zenika.model.SecurityAuditorAware" />

<jpa:auditing auditor-aware-ref="auditorAware" />
```

Enjoy!

```
Contact contact = new Contact();
contact.setFirstname("Joe");
contact.setLastname("Dalton");
// l'utilisateur doit être positionné sur l'AuditorAware !
contactRepository.save(contact);
// lastModifiedBy, lastModifiedDate, etc. positionnés !
```

Plan

Plan

Où démarquer les transactions ?

- Transactions démarquées généralement sur la couche métier
- Une méthode métier = un cas d'utilisation
- Cas d'utilisation = opération atomique
- Ce n'est pas une règle absolue

Transaction au niveau de la couche métier

- Penser ACID...

```
@Service
@Transactional
public class ContactServiceImpl implements ContactService {

  @Autowired ContactRepository repo;

  public Contact create(Contact contact) {
    if(repo.exists(contact)) {
      throw new AlreadyExistingContactException();
    } else {
      repo.save(contact);
    }
    return contact;
  }

}
```

Transaction au niveau de la couche repository

- Chaque méthode de repository est transactionnelle
- Fonctionnel pour des applications purement CRUD
- Catastrophique pour d'autres applications...

```
// dangereux si appelé en-dehors d'une transaction !
for(Contact contact : contacts} {
  if(!repo.exists(contact)) {
    repo.save(contact);
  }
}
```

Plan

Quand toutes les couches dépendent de Spring Data

```
@Service
@Transactional
public class ContactServiceImpl implements ContactService {

  @Autowired ContactRepository repo;

  public Page getPage(Contact modele,Pageable pageable) {
    return repo.findByFirstnameAndLastname(
      modele.getFirstname(),modele.getLastname(),
      pageable
    );
  }

}
```

Qui passe à travers les couches ?

- Typiquement, l'API de tri et de pagination
  - `Sort`, `Page`, `Pageable`, `Pageable`
- Les API de criteria/DSL
  - JPA 2.0, Querydsl

C'est grave ?

- Les dépendances vers Spring Data peuvent être tolérées
  - package `org.springframework.data.domain`
  - Même tolérance que pour `DataAccessException`
- Critères à prendre en compte
  - testabilité de la couche métier avec des mock objects
  - trop de couplage avec une technologie
- Tout ce que nous avons vu ne présente pas un couplage fort