MODULE Spring
Spring Data

HUMAN booster
VOTRE SOLUTION COMPETENCE

Spring Data

Plan du module Spring Data

- Introduction
- Concepts de base
- La gestion des transactions
- JPA avec Spring
- JPA avec Spring Boot
- Minimiser le poids de Spring Boot

@ Patrick Duc 2019

Spring Data

Introduction

@ Patrick Duc 2019

Introduction

Spring Data concerne la **persistence** des données.

Qu’entend-t’on par « persistence » ?

@ Patrick Duc 2019

Introduction

La persistence est le fait que certaines informations manipulées dans un programme puissent être enregistrées de manière **durable** (i.e. même après arrêt du programme) de façon à pouvoir être récupérées à un moment ultérieur.

Les manières d'enregistrer les informations sont multiples : dans un simple fichier (XML, texte, JSON, binaire...), dans une base de données relationnelle, dans une base de données NoSQL, ...

@ Patrick Duc 2019

Introduction

Le projet **Spring Data** fournit des services permettant de faciliter la persistence des données.

Ces services permettent de réaliser des **DAO** (Data Access Object) de manière très simple.

Et ce, de manière relativement uniforme pour différents systèmes de gestion de la persistence (JDBC en direct, JPA, Hibernate, MySQL en direct, PostgreSQL, GemFire, MongoDB, Neo4j, mémoire, etc.).

@ Patrick Duc 2019

Introduction

Spring permet de simplifier sur plusieurs plans le développement de DAO réellement indépendants de la technologie de persistance utilisée :

- la gestion des ressources du système est simplifiée
- les erreurs propres à la technologie de persistance sont wrappées dans des exceptions Spring, sans perte d'information
- les tests sont facilités de par la possibilité de modifier la configuration simplement

@ Patrick Duc 2019

Introduction

- la gestion des transactions est intégrée.

@ Patrick Duc 2019

Spring Data

Concepts de base

@ Patrick Duc 2019

Concepts de base

Pour simplifier et uniformiser la gestion technique de la persistence, Spring propose une abstraction de repository : l'interface `org.springframework.data.repository.Repository`.

Cette interface est générique : elle prend en paramètre le type de donnée à persister (entité, au sens JPA) et le type des identifiants de ce type de donnée.

Elle ne comporte pas d'opérations, ces dernières seront portées par des interfaces dérivées de celle-ci.

@ Patrick Duc 2019

Concepts de base

Exemple : supposons qu'on veuille persister des objets du type `Utilisateur`, qui sont identifiés dans une base de données par un numéro de type `Long`.

L'interface correspondante serait définie comme une dérivation de `Repository` :

```
public interface MonRepository<Utilisateur, Long>
    extends Repository { … }
```

On parle pour `MonRepository` d'un "**domain repository**".

@ Patrick Duc 2019

Concepts de base

De la même manière, on parle de “**domain classes**” pour les classes persistées (entités en JPA).

En pratique, un domain repository doit dériver d’une interface elle-même dérivée de `Repository`.

La première et la plus simple est `CrudRepository`.

Elle fournit un ensemble d’opérations standard de persistence : sauvegarde, listage, recheche par identifiant, suppression, test d’existence par identifiant, comptage d’éléments, vidage complet, etc.

@ Patrick Duc 2019

Concepts de base

Il existe aussi `PagingAndSortingRepository`, qui dérive de `CrudRepository`.

Son but est de faciliter la recherche de données persistées selon certains critères de **tri** et en retournant un nombre d'occurrences configurable (notion de **page** de recherche).

Elle fournit des opérations liées à ces services.

```
Page<T> findAll(Pageable pageable);

Iterable<T> findAll(Sort sort);
```

@ Patrick Duc 2019

Concepts de base

Il existe une interface dédiée à JPA :

`org.springframework.data.jpa.repository.JpaRepository`

qui dérive de `PagingAndSortingRepository`.

Elle fournit quelques opérations supplementaires liées à JPA.

D’autres interfaces spécifiques d’autres technologies de persistence existent aussi bien sûr.

@ Patrick Duc 2019

Concepts de base

Spring fournit des “Spring Data modules” qui permettent l’intégration de différentes technologies de persistence de données (JPA, Hibernate, JDBC, MySQL, MongoDB, etc.).

Il est possible d’utiliser plusieurs Spring Data modules dans la même application.

L’information à Spring du Spring Data module à utiliser peut être donnée **explicitement** dans le code, par **fichier XML**, ou par **annotation**.

@ Patrick Duc 2019

Concepts de base

Exemple de configuration par fichier XML.

```
<beans … xmlns:jpa=”http://www.springframework.org/schema/data/jpa” xsi:schemaLocation=”http://www.springframework.org/schema/data/jpa https://www.springframework.org/schema/data/jpa/spring-jpa.xsd”>

    <jpa:repositories base-package=”mon.package.de.repository”/>

</beans>
```

@ Patrick Duc 2019

Concepts de base

```
interface LivreRepository extends Repository<Livre, Long> { … }

@Entity // entité JPA ==> repository JPA

Class Livre { … }

interface UsagerRepository extends Repository<Usager, Long> { … }

@Document // objet MongoDB ==> repository MongoDB

Class Usager { … }
```

@ Patrick Duc 2019

Concepts de base

Spring définit l'annotation `@Repository` qui permet d'indiquer que la classe ainsi annotée doit être placée dans le contexte applicatif, et pourra ainsi se voir injectée dans un autre bean.

En effet, pour rappel, un `@Repository` est un `@Component`...

@ Patrick Duc 2019

Concepts de base

Exemple.

```
@Repository

interface UserRepository extends
JpaRepository <User, Integer> { … }
```

@ Patrick Duc 2019

Concepts de base

Autre concept : les méthodes-requêtes.

En général, un système de persistence donne la possibilité d'effectuer des requêtes vers la base de données.

Exemple : les requêtes SQL.

Spring permet d'associer de manière très simple une requête paramétrable à une méthode portée par une interface dérivée de `Repository`.

@ Patrick Duc 2019

Concepts de base

En se basant sur le **nom de la méthode**, Spring peut découvrir quelle requête lui correspond et peut **implémenter** cette méthode **automatiquement**.

Spring se base pour cela sur des conventions de nommage des méthodes :

- Débute par `findBy`, `queryBy`, `readBy`, `countBy` ou `getBy`
- Contient ensuite un ou des nom(s) de propriété
- Les noms de propriété peuvent être articulés par `And` ou `Or`

@ Patrick Duc 2019

Concepts de base

Possibilités additionnelles :

- Peut contenir beaucoup d'autres opérateurs de comparaison, comme `Between`, `LessThan`, `GreaterThan`, `Like`, etc.
- Peut contenir le mot-clé `Distinct` dans la partie préfixe du nom, pour s'assurer que le résultat ne contient pas deux fois la même valeur
- Le résultat peut être trié en suffixant le nom de la méthode par `OrderByXxxAsc` ou `OrderByXxxDesc` (`Xxx` étant le nom d'une propriété)
- La casse de la valeur des propriétés de type String peut être négligée au moyen de `IgnoreCase` et `AllIgnoreCase`

@ Patrick Duc 2019

Concepts de base

Exemple.

Etant donné la classe suivante :

```
class User {
    String firstName;
    String lastName;
    Address address;
    ...
}
```

@ Patrick Duc 2019

Concepts de base

Les méthodes suivantes sont des méthode-requêtes valide :

```
List<User> findByLastname(String lastName);

List<User> findByFirstnameAndLastname(String
firstName, String lastName);

List<User>
findDistinctUserByLastnameOrderByFirstnameDesc(String
lastname);

List<User>
findUserDistinctByLastnameOrFirstnameAllIgnoreCaseOrd
erByLastnameAsc(String lastName, String firstName);
```

@ Patrick Duc 2019

Concepts de base

Il est possible de parcourir des propriétés imbriquées. Exemple :

```
Class Address {
    String zipCode;
    String city;
    …
}
```

```
List<User> findByAddressZipCode(String zipCode);
```

==> recherche par `(objet).address.zipCode`

@ Patrick Duc 2019

Concepts de base

Question : comment Spring découpe t'il une expression complexe du type `AddressZipCode` ?

1) recherche d'une propriété `addressZipCode` dans la classe concernée

2) si pas trouvée : découpage en une "tête" et une "queue" selon la règle "camel-case" à partir de la droite

==> production de `AddressZip` et de `Code`

==> recherche d'une propriété `AddressZip` et application des phases 1 et 2 de l'algorithme sur la queue jusqu'à trouver une arborescence valide

3) si pas trouvée ==> levée d'une exception

@ Patrick Duc 2019

Concepts de base

Exemple de `AddressZipCode`.

A. Recherche d'une propriété `addressZipCode` dans la classe `User`

==> propriété non trouvée, on poursuit avec la règle 2

B. Découpage en `AddressZip` et `Code`, on poursuit avec la règle 1

C. Recherche d'une propriété `addressZip` dans la classe `User`

==> propriété non trouvée, on poursuit avec la règle 2

D. Décalage à gauche, découpage en `address` et `zipCode`, on poursuit avec la règle 1

@ Patrick Duc 2019

Concepts de base

E. Recherche d'une propriété `address` dans la classe `User`

==> propriété trouvée, on sélectionne `Address` comme classe courante et on poursuit avec la règle 1

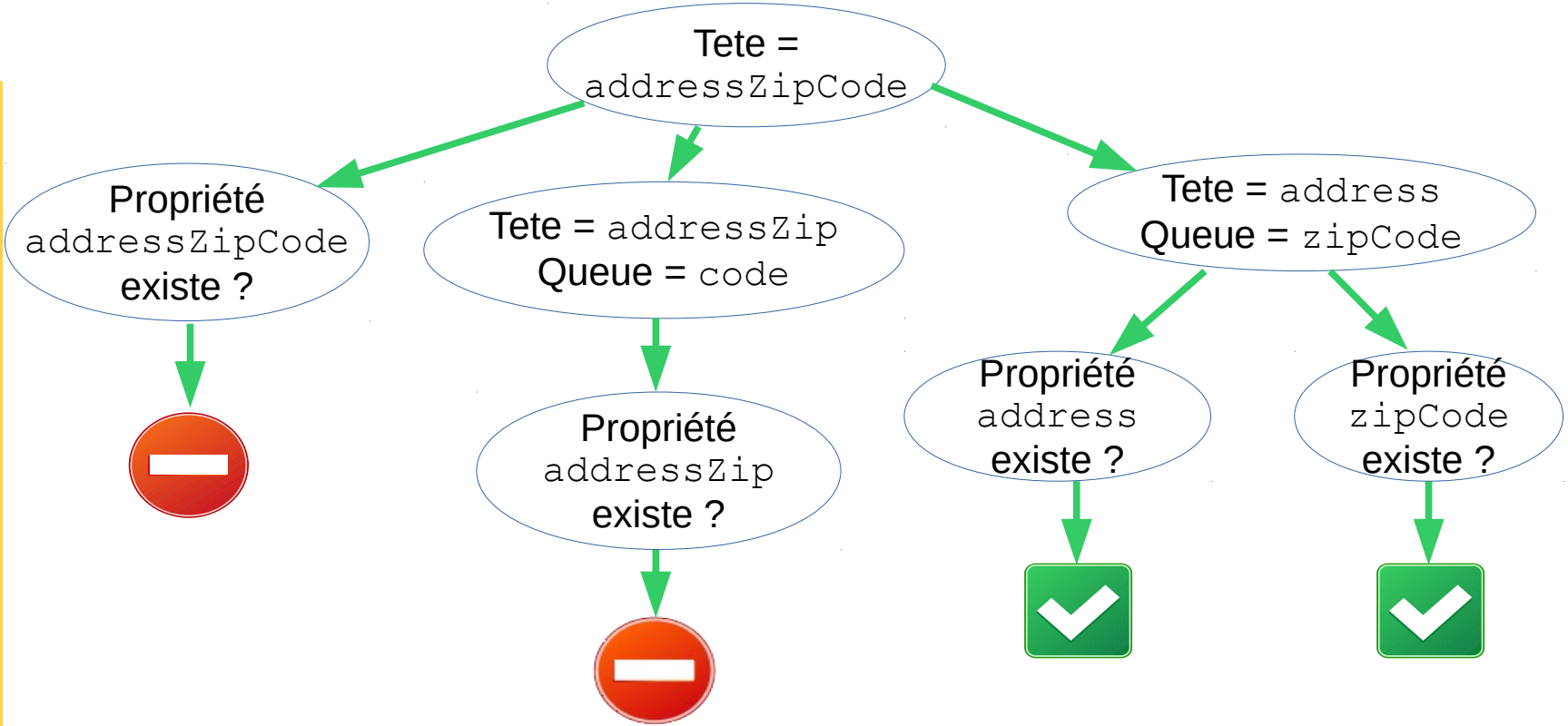
F. Recherche d'une propriété `zipCode` dans la classe `Address`

==> propriété trouvée, sortie avec succès

La requête sera donc de la forme `(objet).address.zipCode`

@ Patrick Duc 2019

Concepts de base



@ Patrick Duc 2019

Concepts de base

Il existe une autre stratégie pour déterminer quelle requête doit être exécutée : rechercher si une requête correspondante n'a pas déjà été déclarée.

Par exemple, en JPA, cela peut se faire par utilisation d'une annotation `@NamedQuery` ou `@NamedNativeQuery` au niveau de la classe, ou d'une annotation `@Query` au niveau d'une méthode du repository.

Exemple pages suivantes.

@ Patrick Duc 2019

Concepts de base

1. Exemple de requête définie au niveau d'une entité

```
@Entity

@NamedQuery(name = "User.findByEmailAddress",

 query = "select u from User u where u.emailAddress
= ?1")

public class User {

    ...

}
```

@ Patrick Duc 2019

Concepts de base

```
public interface UserRepository extends
JpaRepository<User, Long> {

  List<User> findByLastname(String lastname);

  User findByEmailAddress(String emailAddress);

}
```

@ Patrick Duc 2019

Concepts de base

2. Exemple de requête définie au niveau d'un repository

```
public interface UserRepository extends
JpaRepository<User, Long> {

  @Query("select u from User u where u.emailAddress
= ?1")

  User findByEmailAddress(String emailAddress);

}
```

@ Patrick Duc 2019

Concepts de base

```
public interface UserRepository extends
JpaRepository<User, Long> {

  List<User> findByLastname(String lastname);

  User findByEmailAddress(String emailAddress);

}
```

@ Patrick Duc 2019

Concepts de base

Il existe donc 2 stratégies de recherche des requêtes :

- Recherche parmi les requêtes déclarées
- Création d’une requête d’après le nom de la méthode

Le choix de cette stratégie peut être défini par l’utilisateur, soit (comme d’habitude) par fichier XML, soit par annotation.

@ Patrick Duc 2019

Concepts de base

Première possibilité: on utilise l'attribut `query-lookup-strategy` de la balise `<repositories>`.

NB. Cette balise porte aussi l'attribut `base-package`, qui permet d'indiquer le package à partir duquel les repositories doivent être recherchés.

@ Patrick Duc 2019

Concepts de base

Deuxième possibilité : on utilise l'attribut `queryLookupStrategy` de l'annotation `@Enable{store}Repositories`.

Cette annotation, placée sur une classe de configuration Java, permet d'indiquer un package à partir duquel la recherche de repositories doit être effectuée.

`{store}` indique le type de Spring Data module auquel les repositories recherchés doivent appartenir.

@ Patrick Duc 2019

Concepts de base

Si une exception est levée par une technologie de persistence utilisée (JDBC, JPA, …), elle est englobée dans une exception Spring `DataAccessException`.
L'intérêt est que le code de gestion des erreurs devient lui aussi **indépendant** des technologies sous-jacentes utilisées.

La ou les exception(s) originelle(s) est(sont) appelée(s) des exception(s) imbriquée(s), ou « nested exception(s) ».

@ Patrick Duc 2019

Concepts de base

Une `DataAccessException` met à disposition plusieurs méthodes pour obtenir des détails sur la cause originelle du problème :

- `getCause()` retourne l'exception à l'origine de la `DataAccessException`
- `getMostSpecificCause()` retourne l'exception la plus interne, la toute première – qui peut être celle-ci
- `getMessage()` retourne le message associé à l'exception, y compris le message de la ou des exception(s) interne(s)

@ Patrick Duc 2019

Concepts de base

```
{
    ...
}
catch (DataAccessException e) {
    logger.error("Data access exception recue : " + e);
    logger.error("Message : " + e.getMessage());
    logger.error("Exception d'origine : " + e.getCause());
    logger.error("Exception racine : " + e.getMostSpecificCause());
    throw(e);
}
```

@ Patrick Duc 2019

Spring Data

La gestion des transactions

@ Patrick Duc 2019

La gestion des transactions

Spring propose un modèle de transaction compatible des différents modèles de transaction sous-jacents :

- JDBC
- JTA
- Hibernate
- JPA
- ...

@ Patrick Duc 2019

La gestion des transactions

Ce modèle de transaction est supporté par une API et une annotation.

On ne présentera ici que l’annotation, qui est suffisante pour la grosse majorité des cas d’utilisation.

@ Patrick Duc 2019

Les annotations

`@Transactional`

Cette annotation permet de préciser les caractéristiques transactionnelles de cette ou ces méthode(s) :

- La propagation de la transaction
- L'isolation de la transaction
- Le time-out (si supporté par le gestionnaire de transactions)
- Le caractère read-only ou pas de la transaction.

@ Patrick Duc 2019

Propagation des transactions

Si une méthode s'exécutant dans le cadre d'une transaction lève une `RuntimeException`, alors la transaction sera invalidée (rollback).

Si l'exception levée est de type checked, alors par défaut la transaction ne sera pas invalidée.

Ce comportement peut cependant être modifié au travers d'attributs de l'annotation.

Le gestionnaire de transactions à utiliser peut aussi être choisi.

@ Patrick Duc 2019

Propagation des transactions

Si une classe est marquée `@Transactional` et si une de ses méthodes l'est aussi, les caractéristiques transactionnelles définies au niveau de la méthode s'imposeront à celles définies au niveau de la classe.

@ Patrick Duc 2019

Propagation des transactions

Spring propose différentes possibilités pour la propagation des transactions :

- `REQUIRED` (valeur par défaut)

  ⇒ l'exécution se fait dans la transaction courante si elle existe, sinon une nouvelle transaction est créée

@ Patrick Duc 2019

Propagation des transactions

```
@Transactional(propagation=Propagation.REQUIRED)
public void operationComplexe(Donnee donnee) {
    dao.FaisQuelqueChose(donnee);
    try {
       beanInterne.faisAutreChose();
    } catch(RuntimeException e) {
       // la transaction en cours est invalidée
    }
}
```

@ Patrick Duc 2019

Propagation des transactions

```
@Transactional(propagation=Propagation.REQUIRED)

public void faisAutreChose() {

    throw new RuntimeException("La transaction est
    invalidée !");

}
```

La transaction utilisée pour l'exécution de `faisAutreChose()` sera la même que celle créée pour l'exécution de `operationComplexe()`. Cette transaction sera donc invalidée.

@ Patrick Duc 2019

Propagation des transactions

- `REQUIRES_NEW`

  ⇒ l’exécution implique la création d’une nouvelle transaction

  ⇒ par conséquent, si cette transaction est invalidée, alors l’éventuelle transaction englobante ne sera pas invalidée

@ Patrick Duc 2019

Propagation des transactions

```
@Transactional(propagation=Propagation.REQUIRED)
public void operationComplexe(Donnee donnee) {
    dao.FaisQuelqueChose(donnee);
    try {
       beanInterne.faisAutreChose();
    } catch(RuntimeException e) {
       // la transaction en cours n'est pas invalidée
    }
}
```

@ Patrick Duc 2019

Propagation des transactions

```
@Transactional(propagation=Propagation.REQUIRES_NEW)

public void faisAutreChose() {

    throw new RuntimeException("La transaction est
    invalidée !");

}
```

Une nouvelle transaction sera créée pour l'exécution de `faisAutreChose()`. Celle créée pour l'exécution de `operationComplexe()` ne sera donc pas invalidée.

@ Patrick Duc 2019

Propagation des transactions

- `NESTED`

  ⇒ comme pour une méthode annotée `REQUIRED`, l'exécution implique l'utilisation d'une transaction existante, ou la création d'une nouvelle transaction si ce n'est pas le cas

  ⇒ cependant, un point de sauvegarde est établi à chaque entrée dans une méthode. Par suite, le rollback de la transaction ne concernera **que ce qui a été réalisé dans la méthode de bas niveau**

@ Patrick Duc 2019

Propagation des transactions

- `NESTED` (suite)

  ⇒ ce type d'organisation n'est supporté que par les transactions gérées par Spring JDBC (classe `org.springfdramework.jdbc.core.support.JdbcDaoSupport`)

- `NEVER`

  ⇒ stipule qu'aucune transaction ne doit exister à l'entrée dans la méthode

  ⇒ si ce n'est pas le cas, alors une exception est levée

@ Patrick Duc 2019

Propagation des transactions

- `NOT_SUPPORTED`

  ⇒ la méthode s'exécute en dehors de toute transaction ; si une transaction est en cours, alors la transaction est mise en pause

- `SUPPORTS`

  ⇒ si une transaction est en cours à l'entrée dans la méthode, alors l'exécution se fait dans le cadre de cette transaction ; sinon, l'exécution se fera hors transaction.

@ Patrick Duc 2019

Propagation des transactions

- `MANDATORY`

  ⇒ stipule qu'une transaction doit exister à l'entrée dans la méthode

  ⇒ si ce n'est pas le cas, alors une exception est levée

@ Patrick Duc 2019

Isolation des transactions

Les accès à une base de données peuvent se faire de manière concurrente.

Si les transactions ne sont pas isolées les unes des autres, cela peut soulever certains problèmes :

- « lecture sale » : une transaction lit des données écrites par une transaction concurrente non validée
- « lecture non reproductible » : une transaction relit des données lues précédemment et constate qu'elles ont été modifiées par une transaction concurrente

@ Patrick Duc 2019

Isolation des transactions

- « lecture fantôme » : une transaction ré-exécute une requête renvoyant un ensemble de lignes et constate que l’ensemble de lignes a changé du fait d’une autre transaction récemment validée.

Spring propose 5 niveaux d’isolation pour choisir la visibilité d’une transaction sur les effets d’autres transactions.

@ Patrick Duc 2019

Isolation des transactions

Ces niveaux d'isolation sont définis au moyen du paramètre `isolation` de l'annotation `@Transactional` :

- `ISOLATION_DEFAULT` : le niveau d'isolation est celui par défaut du gestionnaire de bases de donnée
- `READ_UNCOMMITTED` : le niveau le plus bas ; les 3 problèmes précédents sont possibles
- `READ_COMMITTED` : l'isolation est supérieure : les lectures sales sont impossibles ; les autres problèmes demeurent

@ Patrick Duc 2019

Isolation des transactions

- `REPEATABLE_READ` : le 3ème niveau d'isolation, car il empêche les lectures sales et les lectures non répétables (la lecture d'une ligne retournera toujours le même contenu) ; par contre, la lecture d'un ensemble de lignes peut retourner un résultat différent
- `SERIALIZABLE` : le niveau le plus élevé : les 3 problèmes précédents sont empêchés. Le SGBDR émule l'exécution en série des transactions, ce qui peut **diminuer fortement** les performances. De plus, la validation d'une transaction peut être refusée.

@ Patrick Duc 2019

Concepts de base

On s’intéresse pour la suite à **Spring Data JPA**, qui concerne la persistence via JPA (Java Persistence API).

JPA est une spécification du modèle **ORM** (Object-Relationship Model) pour Java : JSR-338.

On utilisera Spring Boot, qui facilite grandement la configuration de JPA.

@ Patrick Duc 2019

Concepts de base

Vous êtes censés connaître JPA. Par conséquent, on ne décrira pas JPA mais plutôt comment indiquer à Spring Data les informations JPA.

A moins qu'un rappel de JPA soit nécessaire ?

@ Patrick Duc 2019

Spring Data

JPA avec Spring

@ Patrick Duc 2019

JPA avec Spring

Dans JPA, les DAO sont appelés des **entités**.

Les entités sont des POJO annotés. Leur annotation permet à Spring de les reconnaître et de les mapper sur des objets d'un SGBDR (tables et colonnes principalement).

Les entités sont gérées par un **entity manager**. Cet entity manager est en particulier chargé d'assurer la cohérence entre les entités « **managed** » et leur représentation dans le SGBDR.

@ Patrick Duc 2019

JPA avec Spring

Les entités gérées par un entity manager forment un « **persistence context** ».

Un **entity manager** est créé par une fabrique d'entity manager (**entity manager factory**).

Cette fabrique correspond à une « **persistence unit** », par laquelle on définit les caractéristiques de la connexion vers un SGBDR, ou **datasource** : URL de la base de données, identifiant et mot de passe du compte utilisé, driver JDBC utilisé...

@ Patrick Duc 2019

JPA avec Spring

L’outil qui gère les persistence units, donc qui en pratique fait le lien entre les objets Java et les objets du monde des bases de données relationnelles est un « **persistence provider** ».

Exemples de persistence providers :

- Hibernate (le plus utilisé)
- EclipseLink (implémentation de référence)
- Apache Open JPA
- ...

@ Patrick Duc 2019

JPA avec Spring

Il existe 3 manières de configurer l'entity manager factory avec Spring :

- en définissant un `LocalEntityManagerFactoryBean` qui scannera le fichier XML JPA définissant les paramètres de définition de la persistence (`persistence.xml`) :

```
<beans>

   <bean id="myEmf"
class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">

       <property name="persistenceUnitName" value="myPersistenceUnit"/>

   </bean>

</beans>
```

@ Patrick Duc 2019

JPA avec Spring

- en obtenant un `EntityManagerFactory` de JNDI, qui se base sur le même fichier XML `persistence.xml` ou bien sur les entrées persistence-unit-ref du descripteur de déploiement JEE (`web.xml` par exemple) :

```
<beans>

   <jee:jndi-lookup id="myEmf" jndi-name="persistence/myPersistenceUnit">

</beans>
```

@ Patrick Duc 2019

JPA avec Spring

- en définissant un `LocalContainerEntityManagerFactoryBean` qui permettra de choisir précisément les informations de persistence, en se basant sur le fichier XML de définition de la persistence (`persistence.xml`) et sur d'autres sources :

```
<beans>

   <bean id="myEmf"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">

        <property name="dataSource" value="maDataSource"/>

        … etc…

   </bean>

</beans>
```

@ Patrick Duc 2019

JPA avec Spring

Un `LocalContainerEntityManagerFactoryBean` permet de configurer plus finement les paramètres de persistence (data source en particulier, type de gestion des transactions, etc.).

La récupération d'un entity manager factory par JNDI est le moyen standard pour obtenir un tel objet dans un environnement JEE (application s'exécutant sur un serveur JEE).

@ Patrick Duc 2019

JPA avec Spring

Spring permet d'utiliser JPA **sans spécificité Spring**, au travers de l'injection d'un entity manager factory ou d'un entity manager.

Ainsi, on peut utiliser les annotations JPA `@PersistenceUnit` et `@PersistenceContext` pour indiquer les entity manager factories et entity managers à utiliser dans l'application.

@ Patrick Duc 2019

JPA avec Spring

Exemples de mise en oeuvre de `@PersistenceUnit` et `@PersistenceContext` pages suivantes.

@ Patrick Duc 2019

JPA avec Spring

1. Cas d'utilisation d'un entity manager factory

```
public class ProductDaoImpl implements ProductDao {

   private EntityManagerFactory emf;

   @PersistenceUnit

   public void setEntityManagerFactory(EntityManagerFactory emf) {

      this.emf = emf;

   }
```

@ Patrick Duc 2019

JPA avec Spring

```
  public Collection loadProductsByCategory(String category) {

      try (EntityManager em = this.emf.createEntityManager()) {

         Query query = em.createQuery("from Product as p where
p.category = ?1");

         query.setParameter(1, category);

         return query.getResultList();

      }

   }

}
```

@ Patrick Duc 2019

JPA avec Spring

Cette classe, si définie comme un bean, peut se voir injecter un entity manager factory par défaut :

```
<beans>

    <context:annotation-config/>

    <bean id="myProductDao" class="product.ProductDaoImpl"/>

</beans>
```

@ Patrick Duc 2019

JPA avec Spring

2. Cas d'utilisation directe d'un entity manager par défaut

```
public class ProductDaoImpl implements ProductDao {

   @PersistenceContext

   private EntityManager em;

   public Collection loadProductsByCategory(String category) {

     Query query = em.createQuery("from Product as p where p.category = ?1");

     query.setParameter(1, category);

     return query.getResultList();

   }

}
```

@ Patrick Duc 2019

JPA avec Spring

Cependant, le module Spring Data JPA permet de masquer les concepts d'entity manager et d'entity manager factory au travers de l'interface `Repository`.

D'ailleurs… à quel concept JPA correspond un repository ?

@ Patrick Duc 2019

JPA avec Spring

Une implémentation de l'interface `Repository` correspond à un contexte de persistence (alias entity manager), puisque l'interface porte les méthodes correspondant à des opérations de lecture / sauvegarde / recherche / suppression / etc.

Tout comme l'entity manager est l'interface par laquelle sont réalisées les opérations correspondantes.

@ Patrick Duc 2019

JPA avec Spring

L’un des apports de Spring est qu’il n’est pas nécessaire d’implémenter l’interface repository que l’on souhaite définir pour nos entités !

Ainsi, l’interface suivante sera implémentée automatiquement par Spring lors de l’exécution.

```
public interface MonRepository<Utilisateur, Long>
    extends JpaRepository { … }
```

@ Patrick Duc 2019

JPA avec Spring

Et comme on l’a vu, Spring permet de générer également automatiquement les requêtes sous-jacentes aux méthodes-requêtes définies par le développeur, ou bien d’associer des requêtes JPQL existantes à des méthodes-requêtes du repository.

@ Patrick Duc 2019

Spring Data

JPA avec Spring Boot

@ Patrick Duc 2019

JPA avec Spring Boot

Spring Boot utilise par défaut Hibernate comme persistence provider.

Une application Spring Boot portant l'annotation `@SpringBootApplication`, la fonction **d'auto-scan** du package de la classe principale est automatiquement activée.

Et comme une `@Entity` JPA est un bean, les entités sont automatiquement détectées.

@ Patrick Duc 2019

JPA avec Spring Boot

Même chose pour un `@Repository` : étant donné que c'est un `@Component`, une telle classe sera automatiquement détectée et placée dans le contexte applicatif.

D'où la possibilité d'injecter ce repository là où il sera nécessaire.

@ Patrick Duc 2019

JPA avec Spring Boot

NB. L'auto-scan lié à une `@SpringBootApplication` ne fonctionne par défaut que sur le package contenant l'application et sur ses sous-packages.

Que faire si des repositories et des entités se trouvent dans un autre package ?

Spring propose deux annotations pour résoudre ce problème.

`@EnableJpaRepositories` permet d'indiquer une liste de packages à scanner pour y rechercher des repositories.

@ Patrick Duc 2019

JPA avec Spring Boot

Et `@EntityScan` permet d'indiquer une liste de packages à scanner pour y rechercher des entités.

```
@SpringBootApplication

@EntityScan(basePackages =
{"mon.package.de.repositories"})

@EnableJpaRepositories(basePackages =
{"mon.package.d.entites"})

public class SpringBootDataJpaApplication { … }
```

@ Patrick Duc 2019

JPA avec Spring Boot

De plus, une `@SpringBootApplication` est automatiquement une classe de configuration Java (Java-based configuration).

La configuration d'une data source peut du coup être simplement par code Java, comme montré sur l'exemple page suivante.

@ Patrick Duc 2019

JPA avec Spring Boot

```
@Bean
public DataSource dataSource() {
    DriverManagerDataSource dataSource = new
    DriverManagerDataSource();
    dataSource.setDriverClassName("com.mysql.cj
    .jdbc.driver");
    dataSource.setUsername("user1");
    dataSource.setPassword('password");
```

@ Patrick Duc 2019

JPA avec Spring Boot

```
    dataSource.setUrl(”jdbc:mysql://
    localhost:3306/myDb?
    createDatabaseIfNotExist=true”);

    return dataSource;

}
```

@ Patrick Duc 2019

JPA avec Spring Boot

Bien entendu, la définition d'une datasource peut aussi être faite par fichier de propriétés :

```
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.datasource.username=mysqluser

spring.datasource.password=mysqlpass

spring.datasource.url=jdbc:mysql://localhost:3306/myDb?
createDatabaseIfNotExist=true
```

@ Patrick Duc 2019

JPA avec Spring Boot

Ceci, associé au fait qu'une `@SpringBootApplication` active automatiquement **l'auto-configuration** (recherche automatique de beans avec injection de dépendances, et intégration de composants externes détectés en fonction des classes et JAR trouvés dans le classpath), fait que si le Spring Data module JPA est trouvé dans le classpath et qu'une datasource est disponible dans le contexte applicatif, alors cette datasource sera automatiquement associée aux repositories.

@ Patrick Duc 2019

JPA avec Spring Boot

Mise en pratique :

- Exercice 01 : une application Spring MVC + Spring Data JPA

@ Patrick Duc 2019

Spring Data

Minimiser le poids de Spring Boot

@ Patrick Duc 2019

Minimiser le poids de Spring Boot

Comme on peut le voir en consultant les dépendances d'un projet Spring Boot, cet environnement « tire » beaucoup de choses, qui ne sont pas obligatoirement nécessaires.

Dans un projet opérationnel, il faut toujours chercher à minimiser les dépendances.

Il est donc recommandé de remplacer les dépendances Spring Boot nommées `starter-*` par celles effectivement utilisées, comme montré sur l'exemple suivant.

@ Patrick Duc 2019

Minimiser le poids de Spring Boot

Exemple avec Spring Boot JPA

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

@ Patrick Duc 2019

Minimiser le poids de Spring Boot

```
<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-jpa</artifactId>
</dependency>
```

@ Patrick Duc 2019

Minimiser le poids de Spring Boot

Autre recommandation : exclure les fonctions d'auto-configuration inutilisées.

Ceci peut se faire en remplaçant l'annotation `@SpringBootApplication` par les 3 annotations `@Configuration`, `@EnableAutoConfiguration` et `@ComponentScan` et en utilisant l'attribut `exclude` de la deuxième annotation pour éliminer les services d'auto-configuration inutilisés.

@ Patrick Duc 2019

Minimiser le poids de Spring Boot

Exemple : pas d’auto-configuration de la datasource

```
@Configuration
@EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.class)}
@ComponentScan
public class MySpringBootApplication { … }
```

@ Patrick Duc 2019

Minimiser le poids de Spring Boot

Autre recommandation : désactiver tous les services inutilisés au travers des directives de configuration de Spring Boot dans le fichier `application.properties`.

Exemple page suivante.

@ Patrick Duc 2019

Minimiser le poids de Spring Boot

```
spring.main.web-environment=false
spring.main.banner-mode=off
spring.jmx.enabled=false
server.jsp-servlet.registered=false
spring.freemarker.enabled=false
spring.groovy.template.enabled=false
spring.http.multipart.enabled=false
spring.mobile.sitepreference.enabled=false
spring.session.jdbc.initializer.enabled=false
spring.thymeleaf.cache=false
...
```

@ Patrick Duc 2019

Minimiser le poids de Spring Boot

Source :

https://stackoverflow.com/questions/39241851/developing-spring-boot-application-with-lower-footprint

@ Patrick Duc 2019

Minimiser le poids de Spring Boot

Côté mémoire, choisir des paramètres de mémoire JVM adaptés à Spring peut améliorer les choses (voir l'article `https://spring.io/blog/2015/12/10/spring-boot-memory-performance`).

Mémoire maximale utilisable par une JVM : paramètre `Xmx`

Taille de la pile d'exécution : paramètre `Xss`

```
-Xmx32m -Xss256k
```

@ Patrick Duc 2019