



## Module 1

Introduction à Java et à Spring Boot

Seance 1 Introduction à Java

Seance 2 Notions avancées en Java

Seance 3 Introduction à Spring Framework

Seance 4 Développement d'applications Web avec Spring Boot

Seance 5 Modèles et persistance

Toute application d'entreprise effectue des opérations de base de données en stockant et en récupérant de grandes quantités de données.

Malgré toutes les technologies disponibles pour la gestion du stockage, les développeurs d'applications ont généralement du mal à effectuer efficacement les opérations de base de données.

Pour les développeurs Java, en utilisant **JPA**, diminuent considérablement la charge d'interaction avec la base de données.

**JPA** constitue un pont entre nos **objets du domaine** (programme Java) et les **modèles relationnels** (programme de base de données).

Ainsi donc nous allons utiliser un ORM Java (**O**bject-**R**elational **M**apping)

## A retenir:

**JPA** : JAVA Persistence API c'est un standard Java issue du groupe de travail d'experts JSR 220 (Java Specification Requests) on dira que c'est une interface de programmation Java.

**ORM**: Object-Relational Mapping; est un type de programme informatique qui se situe entre un programme applicatif et une base de données relationnelle afin de simplifier l'interaction Objet Java et BDR

**EclipseLink** : Est l'implémentation standard de l'interface JPA fournie par Oracle avec le serveur GlassFish

**Hibernate** : Tout comme EclipseLink est une implémentations de l'interface JPA on dira donc que c'est un framework ORM. Il est fournie par Red-Hat

Quelques annotations utiles dans la déclaration d'une classe dit Entité JPA

- `@java.persistence.Entity`

Elle est placée au dessus de la classe à fin qu'elle soit persistante

- `@javax.persistence.Id`

Elle est placée au dessus du champ identifiant obligatoire pour que la classe soit persistante

- `@GeneratedValue`

Elle indique que la clé sera automatiquement générée par le **SGBD**  
l'annotation peut avoir un attribut strategy qui indique comment la clé sera générée

Spring data est un projet du framework spring :

- il permet d'écrire plus simplement le code d'accès aux données
- En évitant d'écrire la plupart du code répétitif des DAO
- Il offre une couche d'abstraction commune à de multiples sources de données (JPA, MongoDB, Redis, etc..)
- Mais il est assez flexible pour adresser les spécificités de chacune

# Spring Data : avantages



- Possibiliter de ne pas créer d'implémentation pour les couches DAO
- Il permet d'écrire plus simplement le code d'accès aux données
- Spring data génère automatiquement l'implémentation des couches DAO
- Pour que spring data puisse gérer de ces différentes opérations :
  - Il faut respecter le formalisme de création d'interfaces DAO héritant de **Repository<T>**

## Interface JpaRepository<T,ID extends Serializable>

### All Superinterfaces:

CrudRepository<T,ID>, PagingAndSortingRepository<T,ID>, QueryByExampleExecutor<T>, Repository<T,ID>

### All Known Implementing Classes:

QueryDslJpaRepository, SimpleJpaRepository

---

@NoRepositoryBean

```
public interface JpaRepository<T,ID extends Serializable>  
extends PagingAndSortingRepository<T,ID>, QueryByExampleExecutor<T>
```

JPA specific extension of Repository.

### Author:

Oliver Gierke, Christoph Strobl, Mark Paluch

# Spring Data JPA

All Methods	Instance Methods	Abstract Methods
Modifier and Type	Method and Description	
void	<b>deleteAllInBatch()</b>	Deletes all entities in a batch call.
void	<b>deleteInBatch(Iterable&lt;T&gt; entities)</b>	Deletes the given entities in a batch which means it will create a single <b>Query</b> .
<b>List&lt;T&gt;</b>	<b>findAll()</b>	
<S extends T> <b>List&lt;S&gt;</b>	<b>findAll(Example&lt;S&gt; example)</b>	
<S extends T> <b>List&lt;S&gt;</b>	<b>findAll(Example&lt;S&gt; example, Sort sort)</b>	
<b>List&lt;T&gt;</b>	<b>findAll(Iterable&lt;ID&gt; ids)</b>	
<b>List&lt;T&gt;</b>	<b>findAll(Sort sort)</b>	
void	<b>flush()</b>	Flushes all pending changes to the database.
<b>T</b>	<b>getOne(ID id)</b>	Returns a reference to the entity with the given identifier.
<S extends T> <b>List&lt;S&gt;</b>	<b>save(Iterable&lt;S&gt; entities)</b>	
<S extends T> <b>S</b>	<b>saveAndFlush(S entity)</b>	Saves an entity and flushes changes instantly.



# Spring Data JPA : requêtes nommées

Keyword	Sample	JPQL snippet
And	<code>findByLastnameAndFirstname</code>	<code>... where x.lastname = ?1 and x.firstname = ?2</code>
Or	<code>findByLastnameOrFirstname</code>	<code>... where x.lastname = ?1 or x.firstname = ?2</code>
Is, Equals	<code>findByFirstname</code> , <code>findByFirstnameIs</code> , <code>findByFirstnameEquals</code>	<code>... where x.firstname = ?1</code>
Between	<code>findByStartDateBetween</code>	<code>... where x.startDate between ?1 and ?2</code>
LessThan	<code>findByAgeLessThan</code>	<code>... where x.age &lt; ?1</code>
LessThanEqual	<code>findByAgeLessThanEqual</code>	<code>... where x.age &lt;= ?1</code>
GreaterThan	<code>findByAgeGreaterThan</code>	<code>... where x.age &gt; ?1</code>
GreaterThanEqual	<code>findByAgeGreaterThanEqual</code>	<code>... where x.age &gt;= ?1</code>
After	<code>findByStartDateAfter</code>	<code>... where x.startDate &gt; ?1</code>
Before	<code>findByStartDateBefore</code>	<code>... where x.startDate &lt; ?1</code>

# Intégration de Spring JPA

## Cas MySQL



Pour une application utilisant Spring Boot, Il vous suffit d'ajouter une dépendance dans votre projet au module **spring-boot-starter-thymeleaf** et **mysql-connector-j**.


Si vous utilisez Maven, il vous faut ajouter dans votre fichier pom.xml :

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

```
<dependency>  
  <groupId>com.mysql</groupId>  
  <artifactId>mysql-connector-j</artifactId>  
  <scope>runtime</scope>  
</dependency>
```

# Configuration de la base de données MySQL

- **Application.properties**

```
5 spring.jpa.hibernate.ddl-auto = none
6  spring.datasource.url=jdbc:mysql://[address]:[port]/[BD]
7 spring.datasource.username=[username]
8 spring.datasource.password=[password]
9 spring.datasource.driver-class-name = com.mysql.jdbc.Driver
```

- **Application.yml**

```
1 server:
2   port: 8081
3
4 spring:
5   application:
6      name: Monetab v1.4
7   jpa:
8     hibernate.ddl-auto: none
9    datasource:
10     url: jdbc:mysql://[address]:[port]/[BD]
11     username: [ username ]
12     password: [ password ]
13     driver-class-name: com.mysql.jdbc.Driver
```

# spring.jpa.hibernate.ddl

- **DDL** (**D**ata **D**efinition **L**anguage)

Il est utilisé pour définir et gérer la structure des bases de données

- **spring.jpa.generate-ddl**

active et désactive la fonctionnalité et est indépendant du fournisseur

- **spring.jpa.hibernate.ddl-auto**

IL est une fonctionnalité Hibernate qui permet d'initialiser la base de données avec Hibernate

- **spring.jpa.hibernate.ddl-auto** (enum)

none, validate, update, create-drop

- **spring.datasource.url**

URL JDBC de la base de données

- **spring.datasource.username**

Nom d'utilisateur de connexion à la base de données.

- **spring.datasource.password**

Mot de passe de connexion à la base de données.

# Spring Data : méthodes requêtes

En plus des interfaces que fournies :

- Spring data permet de faire de la dérivation de requête via les noms de méthodes :
  - Le nom d'une méthode sera traduite en une instruction d'interrogation de la source de données
  - Spring va se baser sur les mots clefs présents dans le nom de la méthode pour faire la génération.

```
4 public interface FormateurRepository extends CrudRepository {  
5  
6     // recherche par "nom"  
7     Formateur findByNom(String nom);  
8  
9     // ici, par "nom" ou "prenom"  
10    Formateur findByNomOrPrenom(String nom, String prenom);  
11  
12 }
```