

Spring Approfondissement

Persistance avec Spring Data JPA

Christophe Fontaine
cfontaine@dawan.fr

18/07/2023

Objectifs



- Implémenter une couche de persistance performante avec Spring Data JPA

Durée : 2 jours

Pré-requis : Maîtriser la programmation orientée objet en Java
Maîtrise les bases de Spring Framework
(inversion de contrôle et injection des dépendances)

Bibliographie

- **Java Spring**
Le socle technique des applications Java EE

Hervé Le Morvan

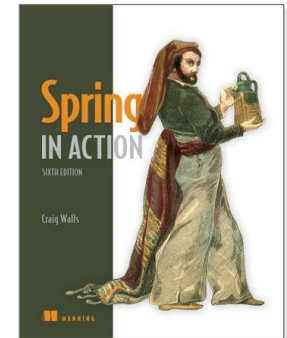
Éditions ENI - 4^{ème} édition - août 2022



- **Spring in Action**

Craig Walls

Manning - 6nd edition - Janvier 2022



- **Spring Reference Documentation**

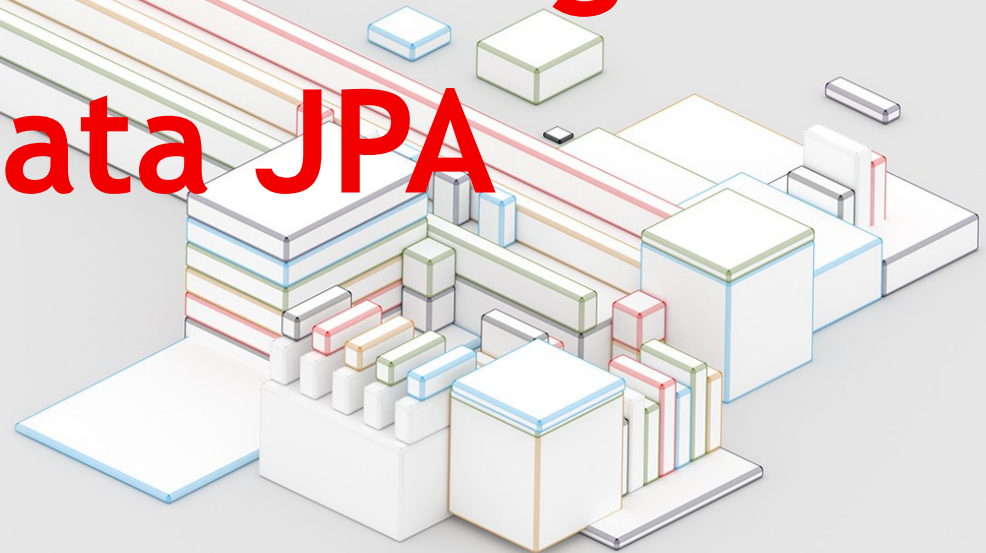
- Spring Framework (<https://docs.spring.io/spring-framework/docs/current/reference/html/>)
- Spring Boot (<https://docs.spring.io/spring-boot/docs/current/reference/html/index.html>)
- Spring Data JPA (<https://docs.spring.io/spring-data/jpa/docs/current/reference/html>)

Plan



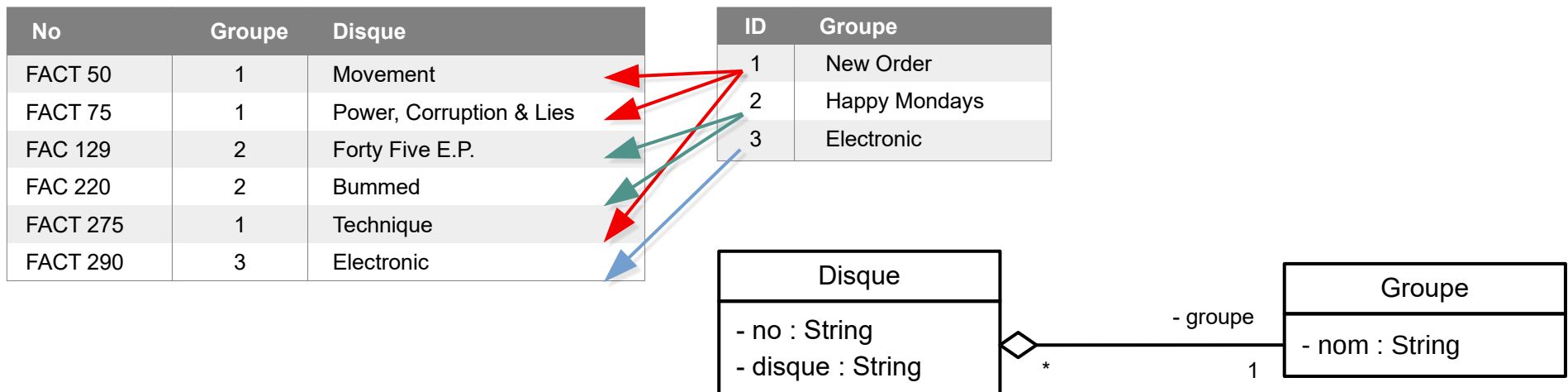
- Configurer un projet Spring Boot pour intégrer Spring Data JPA
- Réaliser le mapping des entités et des opérations
- Écrire des requêtes JPQL ou SQL
- Maîtriser des concepts avancées

Configurer un projet Spring Boot pour intégrer Spring Data JPA



Correspondance des modèles " Relationnel - Objet "

- Le modèle objet propose plus de fonctionnalités :
 - L'héritage
 - Le polymorphisme
- Les relations entre deux entités sont différentes
- De nombreux types de données sont différents
- Les objets ne possèdent pas d'identifiant unique contrairement au modèle relationnel



Mapping relationnel-objet



Concept permettant de connecter un modèle objet à un modèle relationnel

Couche qui va interagir entre l'application et la base de données

- **Pourquoi utiliser ce concept?**
 - Pas besoin de connaître l'ensemble des tables et des champs de la base de données
 - Faire abstraction de toute la partie SQL d'une application

Mapping relationnel-objet

Développeur

Objet

Personne

id

nom

prenom

ORM

Système

SGBD

personnes

Id

Nom

Prenom

...

...

...

Mapping relationnel-objet



- **Avantages :**
 - Gain de temps au niveau du développement d'une application
 - Abstraction de toute la partie SQL
 - La portabilité de l'application d'un point de vue SGBD
- **Inconvénients :**
 - L'optimisation des frameworks/outils proposés
 - La difficulté à maîtriser les frameworks/outils

JPA

- Une API (Java Persistence API)
- Des implémentations :



- Permet de définir le mapping entre des objets Java et des tables en base de données
- Remplace les appels à la base de données via JDBC

Configurer une connexion



- **Dépendances Maven**

- Dépendance pour initialiser JPA

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

- Dépendance pour charger le driver de base de données

```
<dependency>  
  <groupId>mysql</groupId>  
  <artifactId>mysql-connector-java</artifactId>  
</dependency>
```

Configurer une connexion

Pour configurer une connexion il faut définir les propriété suivante :

Propriétés	Description
<code>spring.datasource.url</code>	URL JDBC de la base de données
<code>spring.datasource.username</code>	Nom d'utilisateur de connexion de la base de données
<code>spring.datasource.password</code>	Mot de passe de connexion de la base de données
<code>spring.datasource.driver-class-name</code>	Nom complet du pilote JDBC
<code>spring.jpa.properties.hibernate.dialect</code>	permet à Hibernate de générer du SQL optimisé pour une base de données particulière (liste)
<code>spring.jpa.show-sql</code>	Afficher les requête SQL (par défaut : false)
<code>spring.jpa.hibernate.ddl-auto</code>	Mode DDL: create, update, create-drop (par défaut pour une base de donnée intégrée), none (par défaut)

Configurer une connexion

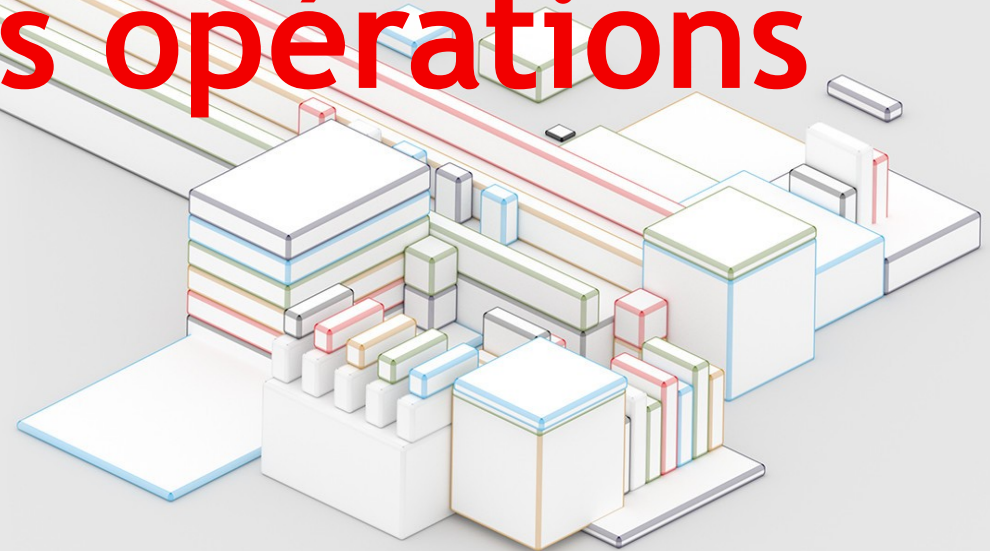


- Exemple de fichier **application.properties**

```
spring.datasource.url=jdbc:mysql://127.0.0.1:3306/formation
spring.datasource.username=root
spring.datasource.password=mot de passe
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.jpa.properties.hibernate.dialect
                                =org.hibernate.dialect.MySQL8Dialect
spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto=create
```

- Ces valeurs sont en fait les valeurs de configuration d'hibernate (persistence.xml)

Réaliser le mapping des entités et des opérations



Entité



- Avec JPA, Une entité est une classe dont les instances peuvent être persisté en base de données
- Une entité est une classe Java standard qui doit :
 - être identifiée comme une entité avec l'annotation **@Entity**
 - avoir un attribut qui joue le rôle d'identifiant annoté avec **@Id** (représente la clé primaire de la table)
 - avoir un **constructeur sans argument**
 - Implémenter l'interface **Serializable**
 - ne doit pas être **final**
 - aucune de ses méthodes ne peut être **final**

Attributs persistants

- Par défaut, **tous les attributs** d'une entité sont persistants
- Les attribut qui ne sont pas persister sont ceux :
 - qui ont pour annotation **@Transient**
 - dont la variable de l'attribut est **transient**
 - qui sont **final** et/ou **static**

```
@Entity
public class NoPersist{
    @Transient
    int attr1;
    transient int attr2;
    static int attr3;
    final int attr4=0;
}
```


Table associée à l'entité

- Par défaut, le nom de la table associée est le nom de la classe de l'entité
- On peut le modifier avec `@Table(name = "nom_table")`

```
@Entity
@Table(name="personnes")
public class Personne implements Serializable {
    @Id
    private long id;
    private String firstName;
    private String name;

    public Personne() {
    }
    // Getters / Setters
}
```

Propriétés de la colonne

- Par défaut, une colonne de la table aura le nom de l'attribut correspondant
- L'annotation **@Column** permet pour définir plus précisément la colonne, avec les attributs suivant :
 - **name**: pour définir le nom de la colonne
 - **unique**: pour définir si le champs doit être unique (par défaut à **false**)
 - **nullable**: pour définir si le champ peut être null (par défaut à **true**)
 - **length**: pour les chaînes de caractères, permet de définir la longueur (par défaut 255)

```
@Column(name="family_name", length=50, nullable=false)  
private String nom ;
```

Énumération

- Une énumération est stockée par défaut sous forme numérique (0,1,... n)
- L'annotation `@Enumerated` permet de définir comment l'énumération sera stockée
 - `EnumType.ORDINAL` stockée sous forme numérique
 - `EnumType.STRING` le nom de l'énumération est stocké

```
public enum Civilite{  
    MADemoiselle, MADame, MONSIEUR  
}  
  
@Entity  
public class Personne{  
    @Enumerated(EnumType.STRING)  
    private Civilite civ; // Stocke MADemoiselle,  
                           MADame ou MONSIEUR  
    // ...                dans la base de données
```

Large OBject

- L'annotation **@Lob** indique que l'attribut de l'entité est un type de données de longueur variable pour stocker des objets volumineux (Large OBject)
- Le type de données peut être un :
 - **CLOB** (Character Large Object) pour stocker du texte
↳ String ou un tableau de char
 - **BLOB** (Binary Large Object) pour stocker des données binaires (images, audio ...)
↳ un tableau d'octet

```
@Entity
public class User {
    @Lob
    private String texte;    // → CLOB

    @Lob
    private byte[] photo;    // → BLOB
}
```

Classe Intégrable

- Une classe intégrable va stocker ses données dans la table de l'entité mère ce qui va créer des colonnes supplémentaires
- La classe intégrable est annotée avec **@Embeddable**
- L'attribut de l'objet dans la classe mère doit utiliser l'annotation **@Embedded**

```
@Embeddable
public class PersonneDetail{
    private LocalDate birthday;
}
```

```
@Entity
public class Personne{
    @Id
    private long id;
    private String prenom;
    private String nom;
    @Embedded
    private PersonneDetail detail;
    //...
}
```

Table personne

id	prenom	nom	birthday

Utilisation multiple d'une classe intégrable



- Une classe entité peut référencer plusieurs instances d'une même classe intégrable
mais le nom de colonnes dans la table associé à ne peuvent être les mêmes pour chacune instance de la classe intégrable
- Pour renommer les colonnes d'un attribut annoté avec **@Embedded**, on utilise l'annotation **@AttributeOverrides** à l'intérieur de cette annotation le nouveau nom de chaque colonnes est définie avec l'annotation **@AttributeOverride**

```
@Embedded
@AttributeOverrides({
    @AttributeOverride(
        name= "nom_colonne",
        column= @Column(name="nouveau_nom_colonne")),
    @AttributeOverride( ... )
})
```

Utilisation multiple d'une classe intégrable

```
@Embeddable  
public class Adresse {  
    private String rue;  
    private String ville;  
    private int codePostal;  
    ...  
}
```

```
@Entity  
public class Employe {  
    @Embedded  
    private Adresse adresse;  
  
    @Embedded  
    @AttributeOverrides({  
        @AttributeOverride(  
            name="ville",  
            column=  
                @Column(name="ville_travail")),  
        @AttributeOverride( ... )  
    })  
    // Adresse du travail  
    private Adresse adresseTravail;  
    // ...  
}
```

Clé primaire

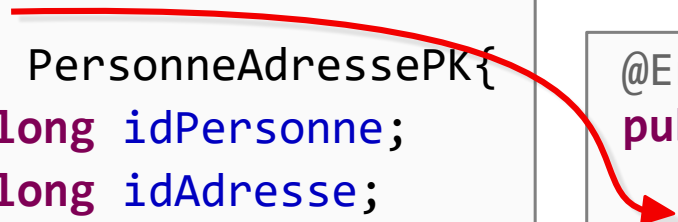


- Une entité doit avoir un attribut qui correspond à la clé primaire dans la table associée
- **Clé primaire simple**
Une entité a un attribut unique qui sert de clé primaire
Il est annoté avec **@Id**
- **Clé primaire composée**
Une clé primaire peut être composée de plusieurs colonnes
Pour mapper une clé primaire composée :
 - On crée une classe intégrable **@Embeddable** qui ne contient que les variables d'instances de la clé primaire
 - On intègre cette classe dans l'entité principale avec l'annotation **@EmbeddedId**

Clé primaire

```
@Embeddable
public class PersonneAdressePK {
    private long idPersonne;
    private long idAdresse;
    public PersonneAdressePK() {
    }
    // ...
}
```

```
@Entity
public class PersonneAdresse {
    @EmbeddedId
    protected PersonneAdressePK pAddrPK;
    //...
}
```



- Pour une clé primaire, on peut utiliser les types suivant :
 - **Type primitif** : byte, int, short, long et char
 - **Classes wrapper** : Byte, Integer, Short, Long et Character
 - String
 - java.math.BigInteger
 - java.util.Date et java.sql.Date

Génération automatique de clé primaire



- L'annotation **@GeneratedValue** indique que la clé primaire est générée automatiquement lors de l'insertion en Bdd
- Elle doit être utilisée en complément de l'annotation **@id**
- Elle a 2 attributs :
 - **generator** : contient le nom du générateur à utiliser
 - **strategy** : permet de spécifier le mode de génération de la clé primaire
- **GenerationType.AUTO** (par défaut)
La génération est gérée par l'implémentation de l'ORM Hibernate crée une séquence unique via la table **hibernate_sequence**

Génération automatique de clé primaire



- **GenerationType.IDENTITY**

La génération se fait à partir d'une propriété entity propre au système de gestion de bdd

- **GenerationType.TABLE**

La génération s'effectue en utilisant une table pour assurer l'unicité. Hibernate crée une table **hibernate_sequence** qui stocke les noms et les valeurs des séquences à utiliser avec l'annotation **@TableGenerator**

```
@Entity
@TableGenerator(name="personneGen")
public class Personne{
    @Id
    @GeneratedValue(strategy = GenerationType.TABLE
                    ,generator = "personneGen")
    private Integer id;
}
```

Génération automatique de clé primaire



- **GenerationType.SEQUENCE**

La génération se fait par une séquence définie par le système de gestion de bdd

À utiliser avec l'annotation **@SequenceGenerator**

```
@Entity
@SequenceGenerator(name="seqGen")
public class Personne{
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE,
                    generator = "seqGen")
    private Integer id;
}
```

Héritage

- L'annotation **@Inheritance** est placé sur la classe parent
- Elle a un attribut **strategy** qui permet de définir la manière d'organiser l'héritage : **SINGLE_TABLE**, **TABLE_PER_CLASS** et **JOINED**
- La différence entre ces 3 stratégie se situe au niveau de l'optimisation du stockage et des performances

Stratégie	Avantages	Inconvénients
SINGLE_TABLE	Aucune jointure → très performant	Organisation des données non optimale
TABLE_PER_CLASS	Performant en insertion	Polymorphisme lourd à gérer
JOINED	Intégration des données proche du modèle objet	Utilisation intensive des jointures → baisse des performances

Héritage : SINGLE_TABLE

- La classe parent et les classes enfants sont dans une seule et même table
- L'annotation **@DiscriminatorColumn**, va ajouter à la table une colonne appelée "**Discriminator**" qui définit le type de la classe enregistrée
- Pour chaque classe, on définit une valeur à placer dans cette colonne avec l'annotation **@DiscriminatorValue**

```
@Entity
@Inheritance (strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn( name="compte_discriminator", length=15
                      discriminatorType=DiscriminatorType.STRING)
@DiscriminatorValue("COMPTE")
public abstract class Compte implements Serializable {...}
```

```
@Entity
@DiscriminatorValue("COMPTE_EPARGNE")
public class CompteEpargne extends Compte implements Serializable {...}
```

Héritage : TABLE_PER_CLASS



- Chaque entité a sa propre table indépendante

```
@Entity
@Inheritance (strategy=InheritanceType.TABLE_PER_CLASS)
public abstract class Compte implements Serializable{
    //...
}

-----

@Entity
public class CompteEpargne extends Compte
                             implements Serializable {
    //...
}
```

- Avec la stratégie **TABLE_PER_CLASS**, on ne peut pas utiliser la stratégie **Identity** pour la génération automatique de clé primaire (**@GeneratedValue**)

Héritage : JOINED

- Chaque Entity a sa propre table
- La relation d'héritage est représentée par une jointure

```
@Entity
@Inheritance (strategy=InheritanceType.JOINED)
public abstract class Compte implements Serializable {
    //...
}
```

```
@Entity
public class CompteEpargne extends Compte
implements Serializable {
    //...
}
```


Classe mère persistante

- Une entité peut aussi avoir une classe mère dont l'état est persistant, sans que cette classe mère ne soit une entité
- La classe mère est annotée avec **@MappedSuperclass**

```
@MappedSuperclass
public abstract class Base {
    @Id @GeneratedValue
    private Long Id;
    // ... }
```

- Aucune table ne correspondra à cette classe mère
L'état de la classe mère sera rendu persistant dans les tables associées à ses classes entités filles

```
@Entity
public class User extends Base{
    // ... }
```

Gestion de la concurrence



- La gestion de la concurrence est essentielle dans le cas de longues transactions
- Hibernate possède plusieurs modèles de concurrence :
 - **None** : la transaction concurrentielle est déléguée au SGBD → Elle peut échouer
 - **Optimistic (Versioned)** : si on détecte un changement dans l'entité, nous ne pouvons pas la mettre à jour
@Version(Numeric, Timestamp, DB Timestamp)
↳ On utilise une colonne explicite Version (meilleure stratégie)
 - **Pessimistic** : utilisation des LockMode spécifiques à chaque SGBD

Relations entre Entity

- **Relations 1,1 → @OneToOne**

Chaque instance d'entité est liée à une seule instance d'une autre entité

Un employé a une seule place de parking et une place de parking ne correspond qu'à un seul employé

- **Relations n,1 / 1,n → @ManyToOne / @OneToMany**

Plusieurs instances d'une entité peuvent être liées à une seule instance de l'autre entité

Une commande ne concerne qu'un seul client et un client peut avoir plusieurs commandes

- **Relations n,m → @ManyToMany**

Les instances d'entité peuvent être liées à plusieurs instances les unes des autres

Un client peut commander plusieurs articles et un article peut être commandé par plusieurs clients

Relations unidirectionnelles

- Une seule entité possède une variable d'instance qui fait référence à l'autre entité
- Elle n'a que le côté propriétaire d'une relation

```
@Entity
public class Employe {
    @Id
    private long id;
    private String name;
    @OneToOne
    private PlaceParking placeParking;
}
```

```
@Entity
public class PlaceParking {
    @Id
    private long id;
    private char etage;
    private int numero;
}
```

En Bdd

employes	
PK	id
	nom
FK	place_parking_id

places_parking	
PK	id
	etage
	numero

- Le coté qui contient la clé étrangère correspond au coté propriétaire (Employe) de la relation

Relations bidirectionnelles



- Chaque entité a une variable d'instance qui fait référence à l'autre entité
- Elles doivent suivre les règles suivantes :
 - Le côté inverse doit faire référence à son côté propriétaire en utilisant l'attribut **mappedBy** de l'annotation **@OneToOne**, **@OneToMany** ou **@ManyToMany**
 - L'attribut **mappedBy** désigne la **variable d'instance** de l'entité qui est le propriétaire de la relation
 - Le côté **@ManyToOne** est toujours le côté propriétaire de la relation (pas de **mappedBy**)
 - Pour les relations **@ManyToMany**, chaque côté peut être le côté propriétaire

Relations bidirectionnelles

```
@Entity
public class Employe {
    @Id
    private long id;
    private String name;
    @OneToOne
    private PlaceParking placeParking;
}
```

```
@Entity
public class PlaceParking {
    @Id
    private long id;
    private char etage;
    private int numero;
    @OneToOne(mappedBy="placeParking")
    private Employe employe;
}
```

En Bdd

employes	
PK	id
	nom
FK	place_parking_id

places_parking	
PK	id
	etage
	numero

- **Requêtes et direction des relations**

Le sens d'une relation détermine si une requête (JPQL ...) peut naviguer d'une entité à une autre

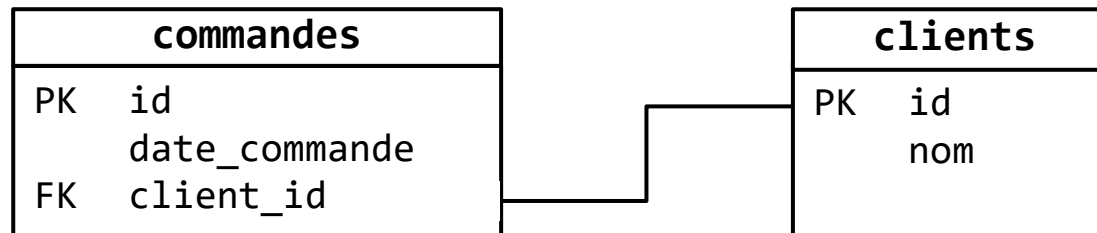
Relations entre Entité

@ManyToOne / @OneToMany

```
@Entity
public class Commande {
    @Id
    private long id;
    @ManyToOne
    private Client client;
    //...
}
```

```
@Entity
public class Client {
    @Id
    private long id;
    private String nom;
    @OneToMany(mappedBy = "client")
    private List<Commande> commandes;
    //...
}
```

En Bdd



- Le côté qui contient la clé étrangère correspond au côté propriétaire (Commande) de la relation

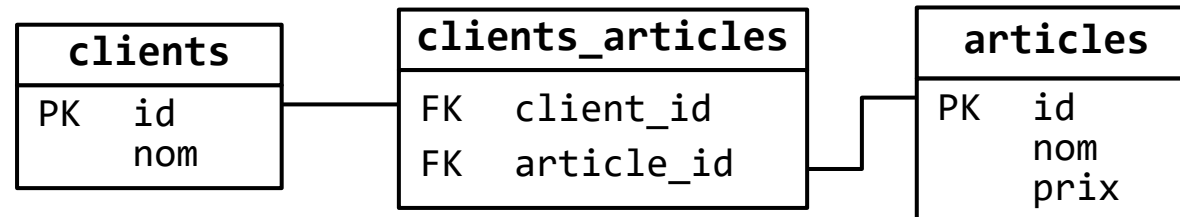
Relations entre Entité

@ManyToMany

```
@Entity
public class Client {
    @Id
    private long id;
    private String nom;
    @ManyToMany
    private List<Article> articles;
    //...
}
```

```
@Entity
public class Article {
    @Id
    private long id;
    private String nom;
    private double prix;
    @ManyToMany(mappedBy = "articles")
    private List<Client> clients;
    //...
}
```

En Bdd



- Chaque côté peut être le côté propriétaire de la relation (celui qui n'a pas d'attribut **mappedBy**)

Renommer la colonne ou la table de jointure



- Avec **@OneToOne** et **@ManyToOne**, on peut renommer la colonne de jointure avec l'annotation **@JoinColumn**

```
@JoinColumn(name="nom_colonne_jointure")
```

```
@ManyToOne
@JoinColumn(name="id_marque")
private Marque marque;
```

- Avec **@ManyToMany**, l'annotation **@JoinTable** permet de renommer la table de jointure et ses colonnes

```
@JoinTable(name="nom_table_jointure",
    joinColumns = @JoinColumn(name="nom_colonne"),
    inverseJoinColumns = @JoinColumn(name="nom_colonne"))
```

```
@ManyToMany
@JoinTable(name="article2fournisseur",
    joinColumns = @JoinColumn(name="fk_article"),
    inverseJoinColumns = @JoinColumn(name="fk_fournisseur"))
private List<Fournisseur> fournisseurs=new ArrayList<>();
```

Stratégies de chargement des relations



- **Chargement tardif : LAZY**

Les entités en relation ne sont chargées qu'au moment de l'accès

↳ par défaut pour **@OneToMany** et **@ManyToMany**

- **avantages**

- temps de chargement initial beaucoup plus court
 - moins de consommation de mémoire

- **désavantages**

- peut avoir un impact sur les performances lors de moments indésirables
 - risque d'exception **LazyInitializationException**

```
@OneToMany(mappedBy="customer", fetch=FetchType.EAGER)
```

Stratégies de chargement des relations



- **Chargement immédiat : EAGER**

Les entités en relation sont chargées dès le load de l'objet

↳ par défaut pour `@OneToOne` et `@ManyToOne`

- **avantages**

- aucun impact sur les performances lié à l'initialisation retardée

- **désavantages**

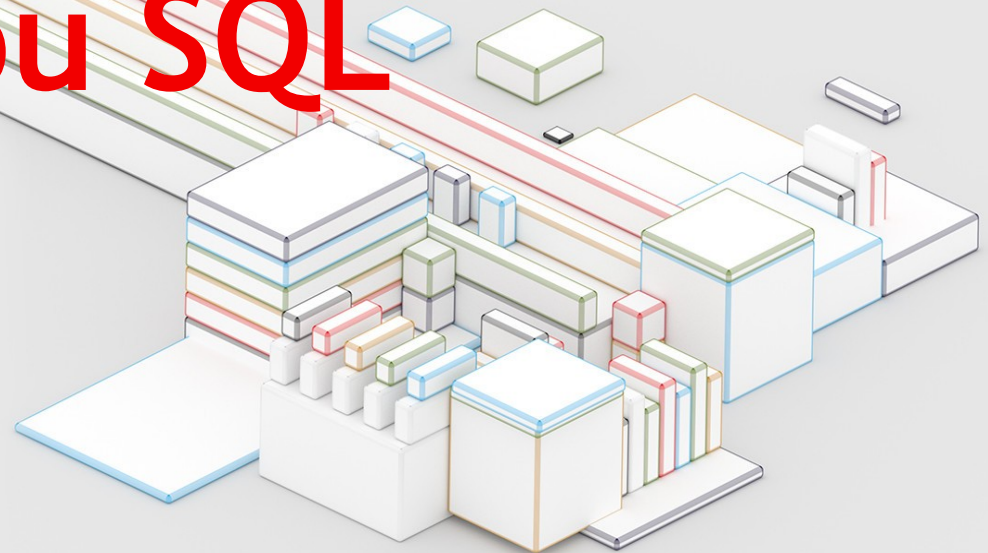
- temps de chargement initial long
 - charge trop de données inutiles, peut avoir un impact sur les performances

Traitement en cascade



- Les annotations **@OneToOne**, **@OneToMany**, **@ManyToOne** et **@ManyToMany** possèdent l'attribut cascade
- Une opération appliquée à une entité est propagée aux relations de celle-ci :
par exemple, lorsqu'un utilisateur est supprimé, son compte l'est également
- 4 Types : **PERSIST** , **MERGE** , **REMOVE** , **REFRESH**
- **CascadeType.ALL** : cumule les 4

Écrire des requêtes JPQL ou SQL



Repository



- Pour implémenter des requêtes sur les données de la base, on va créer une classes d'accès aux données, annoté avec **@Repository**
- Le framework Spring Data JPA permet d'offrir une surcouche de JPA avec un ensemble de DAO déjà prêts
- L'interface central de spring Data est **Repository<T, ID>**
 - **T** correspond à la classe de l'entité
 - **ID** correspond à la classe de l'Id de l'entité
- C'est un interface marqueur qui sert à définir les données avec lesquels on va travailler

CrudRepository, JpaRepository



- L'interface **CrudRepository** fournit à la classe entité des fonctionnalités **CRUD** (**C**reate, **R**ead, **U**psert et **D**eleter) hérite de l'interface Repository)

```
public interface CrudRepository<T, ID> extends Repository<T, ID>
{
    <S extends T> S save(S entity);    // Sauvegarde l'entité
    Optional<T> findById(ID primaryKey); // Retourne l'entité
                                         en fonction de son ID
    Iterable<T> findAll(); // Retourne toutes les entités
    long count();          // Retourne le nombre d'entités
    void delete(T entity); // Supprime l'entité
    boolean existsById(ID primaryKey); // test si l'entité qui
                                         a pour id ID existe

    // ...
}
```

- Il existe aussi des interfaces dépendant de la technologie utilisée (ex: **JpaRepository**, **MongoRepository** ...) qui hérite de **CrudRepository**

Définir les méthodes de la requête



- Dans l'interface repository, on peut créer les requêtes de deux façons :
 - En dérivant la requête depuis le nom de la méthode directement
 - En utilisant une méthode annotée avec **@Query** qui contient la requête écrite manuellement (en JPQL ou en SQL natif)

```
@Repository
public interface PersonRepository extends JpaRepository<Person, Long> {
    List<Person> findByEmailAddressAndLastname(EmailAddress
                                                emailAddress, String lastname);
}

@Repository
public interface QuizRepository extends JpaRepository<Quiz, Long> {
    @Query(value = "FROM Quiz qz WHERE qz.id=:quizId")
    Quiz findQuizById(@Param("quizId") long quizId);
}
```


Création de requêtes avec le nom des méthodes



L'analyse des noms de méthodes est divisée en

- **Sujet**(find...By, exists...By, count...By, delete...By)
Il peut contenir d'autre expression, tout le texte entre find et by est considéré comme descriptif
Sauf pour les mots clés qui limite le résultat (Distinct, Top/First)
Le premier By agit comme un délimiteur pour indiquer le début du prédicat
- **Prédicat**, de façon général :
 - On peut définir des conditions sur les propriétés d'entité et les concaténer avec **AND** et **OR** et aussi les opérateurs comme **Between**, **LessThan**, **GreaterThan** et **Like**

Création de requêtes avec le nom des méthodes



LIKE permet d'effectuer une recherche sur un modèle particulier

% et _ sont 2 jokers :

- % représente : 0, 1 ou plusieurs caractères
- _ représente : un caractère

– On peut utiliser **IgnoreCase** pour

- des propriétés individuelles
(ex: `findByLastnameIgnoreCase`)
- toutes les propriétés d'un type
(ex: `findByLastnameAndFirstnameAllIgnoreCase`)

– On peut ordonner avec **OrderBy** et l'ordre de trie avec **Asc** ou **Desc**

Création de requêtes avec le nom des méthodes



```
@Repository
interface PersonRepository extends Repository<Person, Long> {

    List<Person> findByEmailAddressAndLastname(EmailAddress emailAddress,
                                              String lastname);

    // Enables the distinct flag for the query
    List<Person> findDistinctPeopleByLastnameOrFirstname(String lastname,
                                                         String firstname);
    List<Person> findPeopleDistinctByLastnameOrFirstname(String lastname,
                                                         String firstname);

    // Enabling ignoring case for an individual property
    List<Person> findByLastnameIgnoreCase(String lastname);
    // Enabling ignoring case for all suitable properties
    List<Person> findByLastnameAndFirstnameAllIgnoreCase(String lastname,
                                                         String firstname);

    // Enabling static ORDER BY for a query
    List<Person> findByLastnameOrderByFirstnameAsc(String lastname);
    List<Person> findByLastnameOrderByFirstnameDesc(String lastname);
}
```

Mot clef requête repository



Mot-clés	Exemples
Distinct	<code>findDistinctByLastnameAndFirstname</code>
And	<code>findByLastnameAndFirstname</code>
Or	<code>findByLastnameOrFirstname</code>
Is, Equals	<code>findByFirstnameIs, findByFirstnameEquals</code>
Between	<code>findByStartDateBetween</code>
LessThan	<code>findByAgeLessThan</code>
LessThanEqual	<code>findByAgeLessThanEqual</code>
GreaterThan	<code>findByAgeGreaterThan</code>
GreaterThanEqual	<code>findByAgeGreaterThanEqual</code>
After	<code>findByStartDateAfter</code>
Before	<code>findByStartDateBefore</code>
IsNull, Null	<code>findByAge(Is)Null</code>
IsNotNull, NotNull	<code>findByAge(Is)NotNull</code>
Like	<code>findByFirstnameLike</code>

Mot clef requête repository



Mot-clés	Exemples
NotLike	<code>findByFirstnameNotLike</code>
StartingWith	<code>findByFirstnameStartingWith</code>
EndingWith	<code>findByFirstnameEndingWith</code>
Containing	<code>findByFirstnameContaining</code>
OrderBy	<code>findByAgeOrderByLastnameDesc</code>
Not	<code>findByLastnameNot</code>
In	<code>findByAgeIn(Collection ages)</code>
NotIn	<code>findByAgeNotIn(Collection ages)</code>
True	<code>findByActiveTrue()</code>
False	<code>findByActiveFalse()</code>
IgnoreCase	<code>findByFirstnameIgnoreCase()</code>

- Langage spécifique à JPA
- Il permet de manipuler les propriétés objets dans les requêtes
- Il n'est pas dépendant des noms de tables, des noms de colonnes et des particularité de chaque base de données
- Les classes qui peuvent être désignées par leur nom dans une requête sont :
 - les entités
 - les classes intégrables

```
SELECT e FROM Employe e WHERE e.departement.nom = 'Direction'
```

Requête JPQL



- Une requête JPQL décompose en 5 parties :
 - Le type d'opération : **SELECT**, **UPDATE**, **DELETE**
 - Le périmètre de la requête : **FROM** et **les jointures**
 - Les restrictions: **WHERE**
 - Le regroupement : **GROUP BY** et **HAVING**
 - Le tri : **ORDER BY**
- Les opérations s'exécutent sur **des entités**
- Le texte des requêtes utilise les alias de classe :
select e from Employe as e
- Les attributs des classes doivent être préfixés par les alias **e.nom**

Paramètres des requêtes



- Un paramètre peut être désigné par
 - son numéro **?numéro** (paramètre de position)
 - son nom **:nomDuParamètre** (paramètre nommés)
- Les paramètres sont numérotés à partir de 1
- Un paramètre peut être utilisé plusieurs fois dans une requête

```
@Query("Select p From Pays p WHERE p.nom= :nom OR p.langueList.nom= :nom")  
List<Pays> paysByNomAndLangue(@Param("nom")String nom) ;
```

```
@Query("Select p From Pays p WHERE p.nom= ?1 OR p.langueList.nom= ?1")  
List<Pays> paysByNomAndLangue(String nom) ;
```


Expression de chemin

- Les requêtes peuvent contenir des expressions de chemin pour naviguer entre les entités en suivant les relations (**@OneToOne**, **@OneToMany**, ...)
- Une navigation peut être chaînée à une navigation précédente à la condition que la navigation précédente ne donne qu'une seule entité (**OneToOne** ou **ManyToOne**)
- Si une navigation aboutit à plusieurs entités, on peut utiliser la clause **join** pour obtenir ces entités

e est un alias pour Employe

e.departement → le département d'un employé

e.projets → la collection de projets auxquels participe un employé

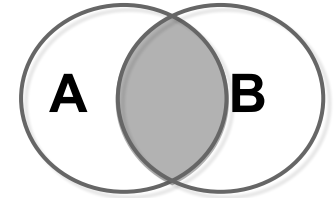
select e **from** Employe e **where** e.departement.nom = 'Qualité'

e.projets.nom n'est pas autorisé → e.projets est une collection

Les jointures

- **Jointure interne (INNER JOIN)**

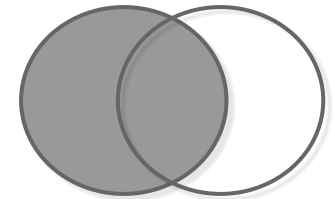
JOIN correspond à une **INNER JOIN** en SQL



```
SELECT DISTINCT e FROM Employe e JOIN e.projets p
SELECT DISTINCT e FROM Employe e JOIN e.projets p
WHERE p.manager= 'Mohamed'
```

- **Jointures externes (LEFT OUTER JOIN)**

LEFT JOIN correspond à une
LEFT OUTER JOIN en SQL



permet de lister tous les résultats de la table de gauche
même s'il n'y a pas de correspondance dans la table de droite

```
SELECT DISTINCT e FROM Employe e LEFT JOIN e.projets p
```

Paging and Sorting



- **Pageable** → permet d'ajouter dynamiquement une pagination à la requête
- **Page** → contient le nombre d'élément et le nombre de page disponible
- **Slice** → un slice ne sait seulement si un autre Slice à la suite

```
Page<User> findByLastname(String lastname, Pageable pageable);  
Slice<User> findByLastname(String lastname, Pageable pageable);
```

- Les options de tri sont également gérées via l'instance **Pageable**, Si on a besoin que d'un tri, on ajoute un paramètre **Sort** à la méthode

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

Paging and Sorting



- **Sort** et **Pageable** doivent être différent de null si l'on ne veut pas
 - de trie: **Sort.unsorted()**
 - de pagination: **Pageable.unpaged()**
- à la place de **Sort** et **Slice**, on peut retourner aussi une liste

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

Limiter les résultats de la requête



- On peut limiter les résultat d'une requête en utilisant **first** ou **top**(identique)
- On peut aussi ajouter un valeur numérique pour spécifier le nombre maximum de résultat retourné
sinon par défaut le nombre d'élément retourné est par défaut égal à 1

```
User findFirstByOrderByLastnameAsc();
User findTopByOrderByAgeDesc();
Page<User> queryFirst10ByLastname(String lastname, Pageable pageable);
Slice<User> findTop3ByLastname(String lastname, Pageable pageable);
List<User> findFirst10ByLastname(String lastname, Sort sort);
List<User> findTop10ByLastname(String lastname, Pageable pageable);
```

Procédure stockée



- Une procédure stockée est une série d'instructions SQL désignée par un nom qui est enregistré dans la base de donnée

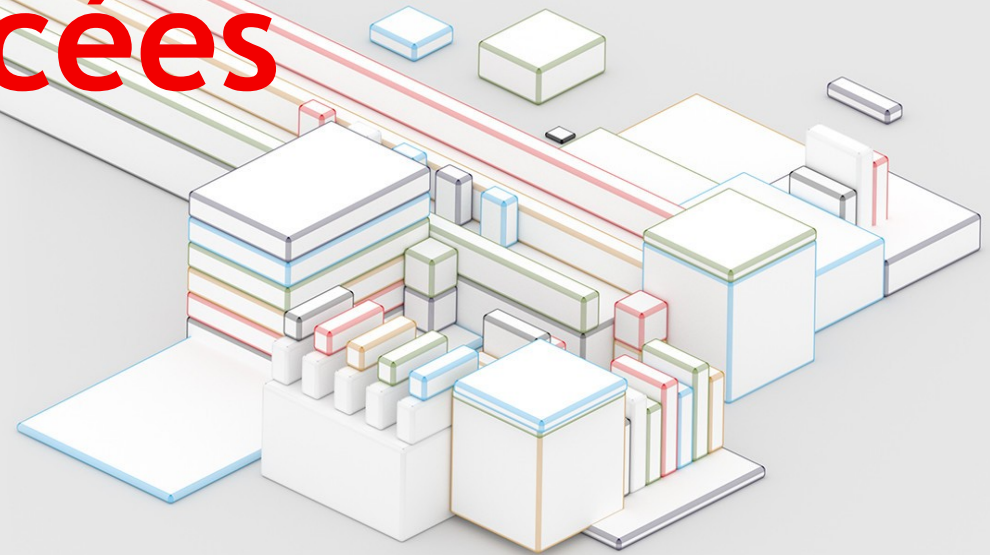
Avantage → Réduire le trafic réseau entre l'application et la SGBD

```
CREATE PROCEDURE doubler( IN val_in INT, OUT val_out INT)
BEGIN
    SET val_out = val_in *2 ;
END
```

- Pour utiliser une procédure stockée, on utilise l'annotation **@Procedure**

```
@Procedure("doubler")    // appel explicite
Integer explicitlyNamedDoubler(Integer arg);
@Procedure                // appel implicite
Integer doubler(@Param("arg") Integer arg);
```

Maîtriser des concepts avancés



Mise en place d'une solution d'audit de tables



- Activation de l'audit avec Spring Data Jpa
↳ **@EnableJpaAuditing**

```
@Configuration
@EnableJpaAuditing
public class AuditConfiguration {

}
```

- Ajout du Listener d'entité JPA de Spring Data sur une entité ou une classe annoté avec @MappedSuperClass
↳ **@EntityListeners(AuditingEntityListener.class)**
- On peut maintenant capturer les informations d'audit par le Listener lors de la persistance et de la mise à jour de l'entité

Mise en place d'une solution d'audit de tables



- La variable d'instance annotée avec :
 - **@CreatedDate** → pour suivre la dates de création
 - **@LastModifiedDate** → pour suivre la date de la dernière modification

```
@CreatedDate
@Column(uptable = false, nullable=false)
private LocalDateTime created;

@LastModifiedDate
@Column(nullable=false)
private LocalDateTime modified;
```

- Et si on utilise Spring Security, on peut suivre l'identité
 - **@CreatedBy** → du créateur de l'entité
 - **@LastModifiedBy** → de celui qui a modifié en dernier l'entité

Mise en place d'une solution d'audit de tables



- En implémentant l'interface `AuditorAware<T>`, on peut personnaliser les valeurs définies pour les attributs annotés avec `@CreatedBy` et `@LastModifiedBy`

```
public class AuditorAwareImpl implements AuditorAware<String> {  
    @Override  
    public Optional<String> getCurrentAuditor() {  
        // ...  
    }  
}
```

- Et créer un Bean dans la classe de configuration

```
@Configuration  
@EnableJpaAuditing(auditorAwareRef = "auditorProvider")  
public class AuditConfiguration {  
    @Bean  
    public AuditorAware<String> auditorProvider(){  
        return new AuditorAwareImpl();  
    }  
}
```



Plus d'informations sur <http://www.dawan.fr>

**Contactez notre service commercial au
09.72.37.73.73 (prix d'un appel local)**