

# Mapping basique

## **Introduction**

Annotations des propriétés

Annotations pour s'adapter au schéma

# Approches du mapping

---

❖ Le mapping peut s'effectuer de 3 façons :

- Annotations JPA/Hibernate :  
Le plus simple et le plus confortable
- Descripteur XML JPA 2 : *orm.xml*  
Vision centralisée mais peu pratique
- Descripteur Hibernate Natif:  
*\*.hbm.xml*  
Projets legacy

# Tyologie des annotations

---

- ❖ Les annotations peuvent être classifiées en différentes catégories :
  - **Logiques** :  
Modèle métier : les entités, leurs attributs et leurs associations
  - **Validation** :  
Contraintes de validation sur le modèle métier
  - **Physique** :  
Adaptation du modèle métier à un modèle physique  
Noms des tables, colonnes, clé étrangères
- ❖ En plus des annotations de *javax.persistence.\**, Hibernate définit des annotations spécifiques dans *org.hibernate.annotations.\**

# Mapping basique

Introduction

**Annotations des propriétés**

Annotation pour s'adapter au schéma

# Classe entité

---

- L'annotation **@Entity** est suffisante sur une classe POJO
- Une classe Entité est généralement sérialisable
- Une classe Entité doit également avoir l'annotation **@Id** qui indique sa clé primaire

# Clé primaire

---

La clé primaire doit être un champ ou une propriété du bean entité, annotée avec ***@javax.persistence.Id***

- l'emplacement de l'annotation *@Id* est déterminant car il indique de type d'accès (*Field* ou *Property*) au bean par la couche de persistance
- L'annotation peut alors se placer devant l'attribut ou devant la méthode *getter*
- les autres annotations doivent utiliser le même type d'accès que *@Id*

La clé primaire doit être d'un type primitif de java ou d'une classe enveloppe (ex : Long) , du type *java.lang.String*, ou d'une classe composée de ces types (clé composite)

# Génération de clé

---

L'annotation **@GeneratedValue** permet de générer automatiquement la clé par le service de persistance:

```
@Id
```

```
@GeneratedValue(strategy=GenerationType.AUTO)
```

```
public int getId() {  
    return id;  
}
```

# Stratégie de génération

---

La méthode utilisée par la couche de persistance pour générer la clé est précisée via l'attribut ***strategy*** de l'annotation *@GeneratedValue*.

Les valeurs possibles sont:

- ***GenerationType.IDENTITY***: utilise une colonne *identity* de la base pour obtenir la clé (DB2, MySQL, MSSQL, Sybase, Hypersonic)
- ***GenerationType.TABLE***: nécessite une table générée par l'annotation *@TableGenerator*
- ***GenerationType.SEQUENCE***: s'appuie sur une séquence de la base de données (Oracle, Postgres, ...)
- ***GenerationType.AUTO***: la plus simple, par défaut. S'adapte au capacité de la base



# ID Naturels

---

Les **identifiants naturels** représentent des identifiants uniques qui ont une signification dans le monde réel.

- Même si un identifiant naturel ne constitue pas une bonne clé primaire, il est toujours utile d'en informer Hibernate.
- Hibernate fournit une API dédiée et efficace pour charger une entité par son identifiant naturel, tout comme elle le propose pour le chargement par son identifiant (PK).

# Example

---

```
@Entity(name = "Book")
public static class Book {

    @Id
    private Long id;

    private String title;

    private String author;

    @NaturalId
    private String isbn;

    //Getters and setters are omitted for brevity
}
```

# Identifiants composites

---

- Les identifiants composites sont à éviter sauf pour mapper une base de données existante.
- 3 syntaxes permettent de spécifier des identifiants composites :
  - Utilise un composant **@EmbeddedId**
  - Définir **plusieurs propriétés avec @Id** (Mécanisme spécifique Hibernate)
  - Définir plusieurs propriétés avec **@Id** et déclarer une classe externe comme **@IdClass**

# Exemple *@EmbeddedId*

---

```
@Entity
class User {
    @EmbeddedId
    @AttributeOverride(name="firstName",
        column=@Column(name="fld_firstname"))
    UserId id;
    Integer age;
}

@Embeddable
class UserId implements Serializable {
    String firstName;
    String lastName;
}
```

# Exemple multiples @Id

---

```
@Entity
class Customer implements Serializable {
@Id @OneToOne
    User user;
@Id
    String customerNumber;
    boolean preferredCustomer;
    //implements equals and hashCode
}
@Entity
class User {
    @EmbeddedId UserId id;
    Integer age;
}
@Embeddable
class UserId implements Serializable {
    String firstName;
    String lastName;
    //implements equals and hashCode
}
```

# Exemple multiples *@Id* et *@IdClass*

---

*Advanced*

```
@Entity
@IdClass(CustomerId.class)
class Customer implements Serializable {
    @Id @OneToOne
    User user;
    @Id String customerNumber;
    boolean preferredCustomer;
}
class CustomerId implements Serializable {
    UserId user;
    String customerNumber;
}
@Entity
class User {
    @EmbeddedId UserId id;
    Integer age;
}
@Embeddable
class UserId implements Serializable {
    String firstName;
    String lastName;
}
```

# Propriétés

---

- Avec les annotations, toute propriété de la classe entité et par défaut persistante  $\Leftrightarrow$  à l'annotation **@Basic**.
- Avec *hbm.xml*, il faut spécifier les propriétés persistantes
- Pour ne pas rendre persistant une propriété, l'annoter avec **@Transient**
- @Basic permet de préciser :
  - **fetch** : La stratégie de fetching. Si lazy, la classe doit être instrumentée au build.
  - **optional** : true/false (Mais plutôt utilisé @NotNull)

# @Temporal

---

L'annotation **@Temporal** s'applique aux champs ou propriétés persistants de type *java.util.Date* ou *java.util.Calendar* et depuis JPA 2.2, *java.time.LocalDate*, *java.time.LocalDateTime*, *java.time.LocalDateTime*, *java.time.OffsetTime* and *java.time.OffsetDateTime*.

- elle permet de préciser le type SQL de la colonne vers laquelle s'effectue le mapping: **DATE**, **TIME** ou **TIMESTAMP**
- l'attribut *TemporalType* de l'annotation *@Temporal* prend la valeur *TIMESTAMP* par défaut



# @Lob

---

L'annotation **@Lob** s'applique aux champs ou propriétés représentant des données binaires (*byte[]*, *Byte[]*, *Serializable*) ou de type caractère (*char[]*, *Character[]*, *String*) non limité en taille dans la base

- le mapping s'effectue alors sur une colonne de type SQL BLOB ou CLOB:
  - BLOB si le champ ou la propriété est *byte[]*, *Byte[]*, *Serializable*
  - CLOB si le champ ou la propriété est *char[]*, *Character[]*, *String*

# @Enumerated

---

L'annotation **@Enumerated** s'applique aux champs ou propriétés de type *enum*

- elle permet de préciser le type SQL de la colonne vers laquelle s'effectue le mapping: chaîne de caractères ou nombre
- l'attribut *EnumType* de l'annotation **@Enumerated** prend la valeur *ORDINAL* (par défaut) ou *STRING*

# Exemple

---

```
@Entity  
public class Personne implements Serializable{  
    @Id private int id;  
    private String nom;  
    private String prenom;
```

```
    @Temporal(TemporalType.DATE)  
    private Date naissance;
```

```
    @Lob private Byte[] photo;
```

```
    @Enumerated(EnumType.STRING)  
    private Genre sexe;
```

# Validation du modèle

---

Les colonnes des entités peuvent être annotés via les annotations de

- **javax.validation**

- @Min, @Max, @NotNull, @DecimalMax, @DecimalMin, @Pattern, @Size, ...*

- **org.hibernate.validator**

- @URL, @NotEmpty, @NotBlank, @Length, @Email*

Les entités sont validées avant la tentative de persistance

Si Hibernate génère le schéma, il peut ajouter des contraintes BD

# Example

---

```
@Entity
public class User {

    @NotNull(message = "Name cannot be null")
    private String name;

    @Size(min = 10, max = 200, message
        = "About Me must be between 10 and 200 characters")
    private String aboutMe;

    @Min(value = 18, message = "Age should not be less than 18")
    @Max(value = 150, message = "Age should not be greater than 150")
    private int age;

    @Email(message = "Email should be valid")
    private String email;

    // standard setters and getters
}
```

# Mapping basique

Introduction

Annotations des propriétés

**Annotation pour s'adapter au schéma**

# Adaptation au modèle physique

---

Si les noms des tables ou colonnes de la base de données ne correspondent pas aux noms utilisés du côté Java, il faut s'adapter au modèle physique via les annotations :

- **@Table** (au niveau de la classe) permet de changer le nom par défaut de la table
- **@Column** (au niveau de l'attribut ou de son getter) permet de changer le nom par défaut de la colonne

Il existe également des annotations permettant de contrôler les noms des clés étrangères

**@JoinColumn** ou des tables d'association

**@JoinTable**

# @Table

---

- L'annotation *@Table* permet de préciser :
  - ***name*** : Le nom de la table
  - ***catalog*** : Le catalogue
  - ***schema*** : Le schéma
  - ***uniqueConstraints*** : Les contraintes sur la table. (Effectif seulement si Hibernate génère la table)



# @Column

---

- L'annotation **@Column** permet de s'adapter au schéma physique :
  - **name** : Le nom de la colonne
  - **unique** (true/false): Contrainte sur la colonne
  - **nullable** (true/false): Colonne requise ou non
  - **insertable** (true/false): incluse dans les ordres INSERT
  - **updatable** (true/false): incluse dans les ordres UPDATE
  - **columnDefinition** (optional): surcharge le fragment DDL pour cette colonne (non portable)
  - **table** : Table cible (par défaut la table primaire)
  - **length** : Longueur de la colonne (défaut 255)
  - **precision** : La précision décimale (défaut 0)
  - **scale** (optional): L'échelle décimale (défaut 0)

# Exemple

---

```
@Entity
@Table(name="TBL_FLIGHT", schema="AIR_COMMAND",
uniqueConstraints=
@UniqueConstraint(name="flight_number",
columnNames={"comp_prefix", "flight_number"} ) )
public class Flight implements Serializable {
    @Column(name="comp_prefix")
    public String getCompagnyPrefix() {return companyPrefix;}

    @Column(name="flight_number")
    public String getNumber() { return number; }
}
```