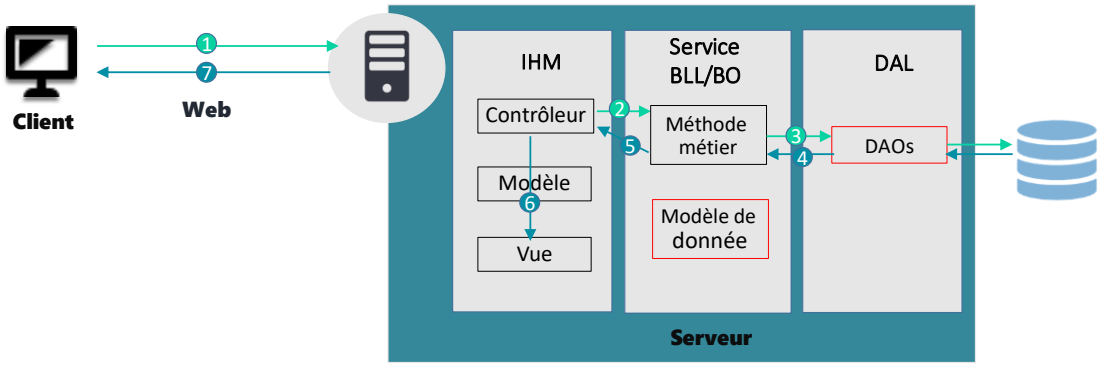
Le framework Spring

Module 5 – Spring Data JPA

Objectifs

- La couche d'accès aux données
- Notion d'Object Relational Mapping
- Définir une entité
- Définir une association
  - One to one
  - One to many
  - Many to one
  - Many to many
  - Unidirectionnelle ou bidirectionnelle
- Utiliser le Repository
- JPQL



L'application est orientée objet **!=** La technologie de BD est relationnelle

3

La couche d'accès aux données (DAL, Data Access Layer) est la couche pour permettre de persister les objets.

C'est dans cette partie que vous allez gérer tout ce qui concerne l'accès aux données.

Il y a, dans cette couche, plusieurs parties bien distinctes :

- La connexion à la base de données
- Les requêtes sur la base de données

    - Et des design pattern à mettre en place : DAO (Data Access Object), CRUD (Create Read Update Delete), Factory (Fabrique d'instance)

- Le mapping des données de la base relationnelle vers les objets Java

    - Pour ce dernier point, nous allons voir la solution des ORM

- Un ORM permet de mettre en correspondance le modèle de données relationnel et le modèle objets
  - On parle alors d'entités
- JPA :
  - est le standard pour les ORM Java
  - apparu en 2006
  - est une spécification, elle nécessite de choisir une implémentation (Hibernate est celle utilisée par défaut dans Spring)

Un ORM permet de mettre en correspondance le modèle de données relationnel et le modèle objets

On parle alors d'entités

Cela permet :

- Amélioration de l'architecture logicielle
- Génération automatique du code de requêtage SQL
- Abstraction de la base de données (travail sur les objets uniquement)
- Gestion des "incompatibilités" (héritage, associations…)

JPA :

est le standard pour les ORM Java

apparu en 2006

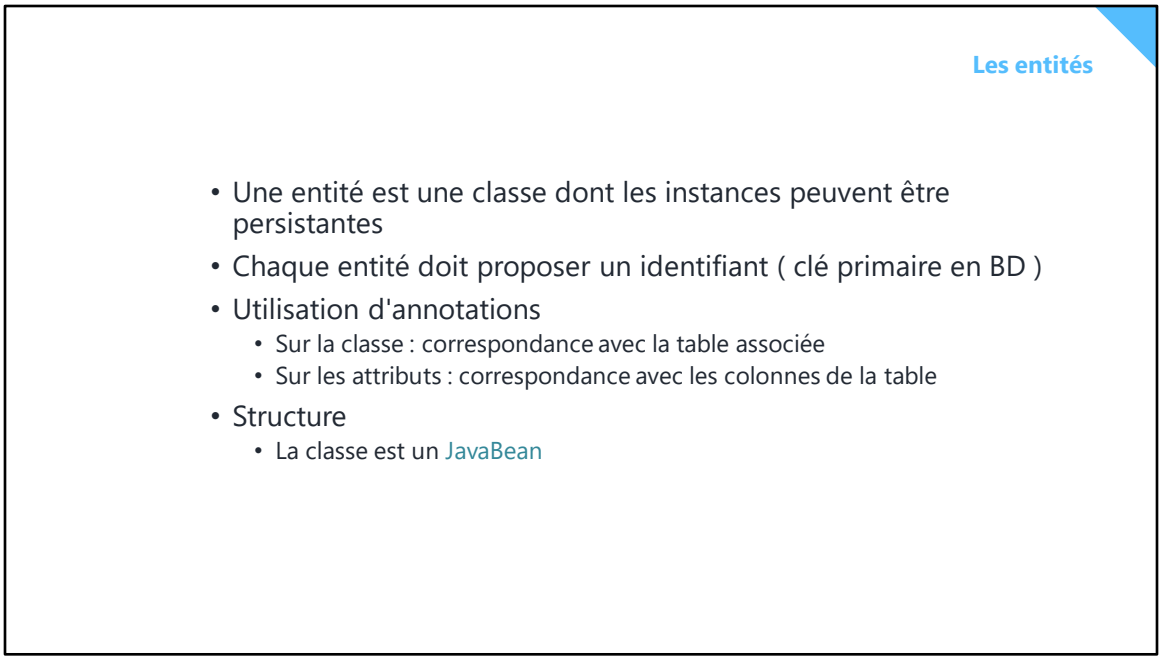
C'est une alternative possible : Spring JDBC

C'est une spécification, elle nécessite de choisir une implémentation (Hibernate est celle utilisée par défaut dans Spring)

- La base de données doit être installée et fonctionnelle
- La dépendance Spring Data JPA (ORM) doit être installée sur le projet
- La dépendance vers le pilote de base de données doit être installée sur le projet
- Les informations d'accès à la base de données doivent être paramétrées dans le projet

La base de données doit être installée et fonctionnelle
La dépendance Spring Data JPA (ORM) doit être installée sur le projet
La dépendance vers le pilote de base de données doit être installée sur le projet
Les informations d'accès à la base de données doivent être paramétrées dans le projet

- Chaine de connexion :
  - Machine (Adresse IP + no de port)
  - Nom de la base de données
- Utilisateur de base de données



Une entité est une classe dont les instances peuvent être persistantes
Chaque entité doit proposer un identifiant (clé primaire en BD)
Utilisation d'annotations
    Sur la classe : correspondance avec la table associée
    Sur les attributs : correspondance avec les colonnes de la table
Structure
    La classe est un JavaBean

Dans les projets Java EE, les entités sont les BO (Business Object)

Définir une entité avec @Entity et @Id

- Annotations obligatoires

```
import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class Course {
  @Id
  private Long id;

  private String title;
…
```

Table générée :

COURSE		
Nom	**Type**	**Contraintes**
id	bigint	PK
title	varchar(255)	

Définir une entité avec @Entity et @Id

- Annotations facultatives :
  - @Table
  - @GeneratedValue
  - @Column
  - @Transient
  - @Basic

@Table : définir le nom de la table en base de données

@GeneratedValue : Définir une clé primaire auto-générée par la base de données

L'attribut strategy permet de préciser comment la clé doit être générée

@Column : permet de définir

le nom de la colonne (attribut name),
la longueur maximum (attribut length)
si la colonne en BD accepte les valeurs nulles ou non (attribut nullable)
si un index unique doit être créé (attribut unique)

@Transient : indique que l'attribut ne sera pas

mappé(et donc non persisté) dans la table

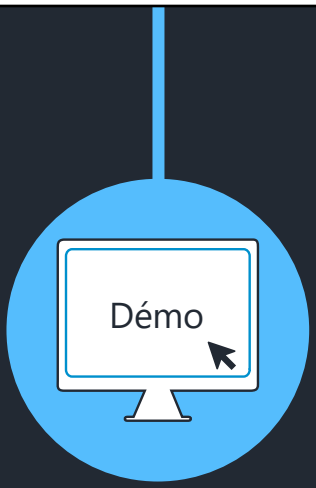
@Basic : annotation par défaut pour un attribut

Peut permettre d'utiliser les attributs fetch (LAZY/EAGER)

Et optional : l'attribut peut être nul ou non

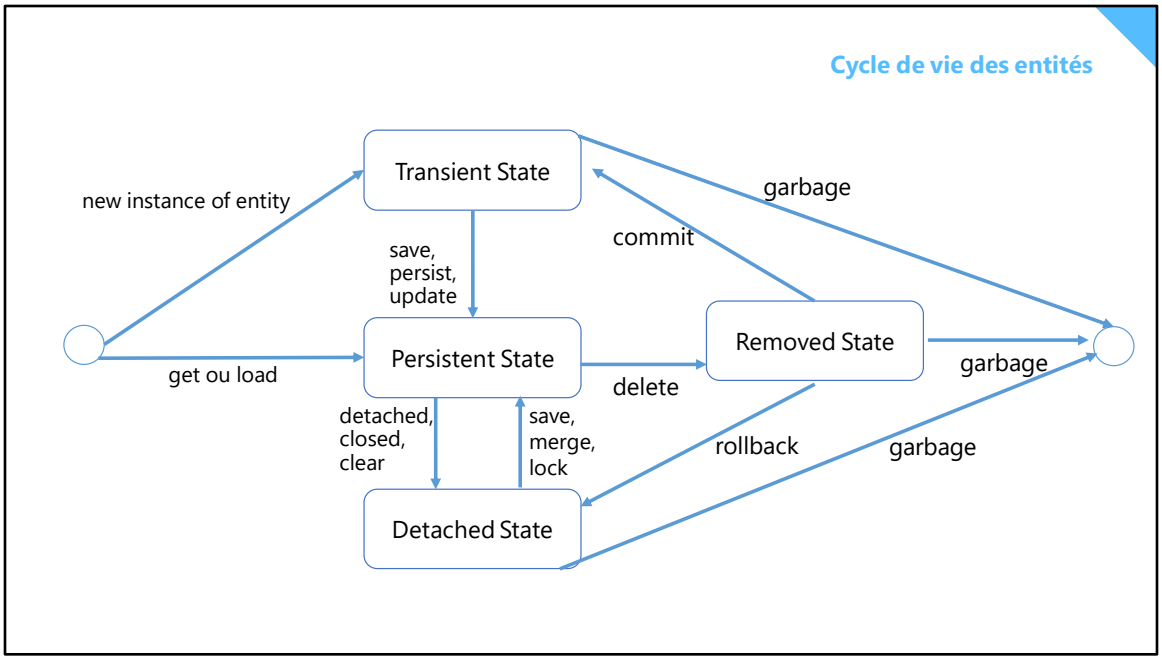
Installation de MySQL Server

- Installons notre serveur de base de données



Définir une entité

- Mettons en place les starter de Spring Boot
- Le driver de la base de données
- La configuration de connexion à la base de données
- Et une entité pour commencer



Les ORM doivent gérer les entités dans un contexte pour pouvoir les manipuler à leur guise.
Dans JPA , Il est possible

- de créer un nouvel objet d'une entité
- et le stocker dans la base de données,
- soit récupérer les données existantes d'une entité à partir de la base de données.

Ces entités sont liées au cycle de vie et chaque objet de l'entité passe par les différentes étapes du cycle de vie.
Il y a principalement quatre états du cycle :
1.État transitoire (Transient)
2.État persistant (Persistent)
3.État détaché (Detached)
4.État supprimé (Removed)

État 1 : État transitoire (Transient)

- Lorsque nous instancions un objet à l'aide de l'opérateur new, l'objet est dans l'état (New).

- Cet objet n'est connecté à aucune session de l'ORM.
- Comme il n'est connecté à aucune session, cet état n'est donc connecté à aucune table de base de données.
- Ainsi, si il y a des modifications aux données de l'instance, la table de la base de données n'est pas modifiée.
- Les objets transitoires sont indépendants de l'ORM et existent dans la **mémoire de tas** .

Dans l'exécution : voici comment un objet peut être dans cet état :

- Lorsque les objets sont générés par une application mais ne sont connectés à aucune session.
- Les objets sont générés par une session fermée.

État 2 : État persistant (Persistent)
Une fois l'entité associée à la session de l'ORM
Il existe deux manières de passer un objet en état persistant :

- Enregistre l'entité dans la base de données.
- Charger une entité depuis la base de données.

Dans cet état. chaque entité représente une ligne dans la table de base de données. Par conséquent, si nous apportons des modifications aux données, l'ORM détectera ces modifications et apportera des modifications à la table de la base de données.

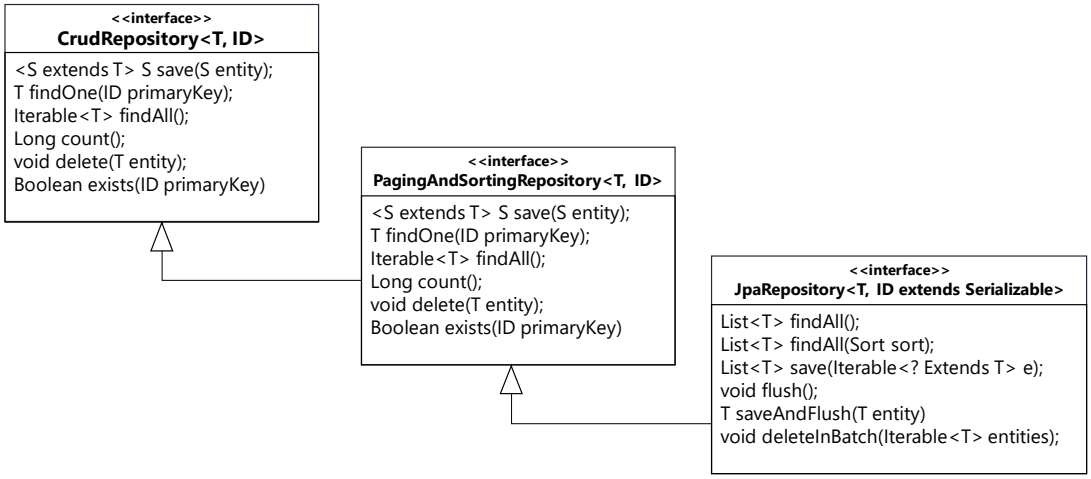
État 3 : État détaché (Detached)
Pour convertir un objet de l'état persistant à l'état détaché, nous devons soit fermer la session, soit vider son cache. Dans ce cas, les modifications apportées aux données n'affecteront pas la base de données. Chaque fois que nécessaire, l'objet détaché peut être reconnecté à une nouvelle session de l'ORM

État 4 : État supprimé (Removed):
Dans le cycle de vie, c'est le dernier état. L'objet est supprimé de la base de données. Dans cet état, si une modification est apportée aux données, cela n'affectera pas la base de données.

Il est important d'avoir ces états en tête quand nous développons avec des ORM.

Spring Repository

- Spring fournit des interfaces pour créer automatiquement les DAO:

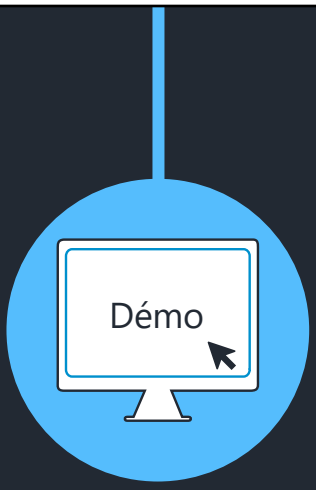


Pour gérer la manipulation des entités en base de données. Il faut créer normalement une couche DAL. Spring Data est une abstraction à JPA.

- L'idée est de ne pas coder toutes les méthodes standard du CRUD (Create Read Update Delete)
- Il fournit aussi des méthodes sur le tri et la pagination
- Et toutes les méthodes liées aux ensemble : select *; insert , delete

Il n'y a aucune implémentation à créer, Spring Data JPA les contient.

Nous verrons plus tard, qu'il est même possible de déclarer des méthodes pour des recherches complexes de cette manière.



Repository : les bases

- Créons notre premier DAO avec Spring Data JPA
- Et voyons les contraintes du cycle de vie.

Les clés primaires composites (méthode 1)

- Utilisation des annotations @Id et @IdClass
  - @Id sur les attributs composant la clé composite
  - Création d'une classe décrivant la clé primaire
    - Attributs, getters et setters de la clé primaire identiques à la classe principale
    - Implémentation de Serializable
    - Constructeur sans paramètre
  - Ajout de l'annotation @IdClass(nomClassePK.class) sur la classe principale

Les clés primaires composites (méthode 2)

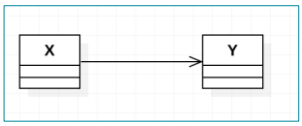
- Utilisation des annotations @EmbeddedId et @Embeddable
  - Création d'une classe décrivant la clé primaire
    - Attributs, getters et setters composant la clé primaire
    - Implémentation de Serializable
    - Constructeur sans paramètre
    - Annoté avec @Embeddable
  - Déclaration d'un attribut instance de classe embarquée
    - @EmbeddedId sur l'attribut composant la clé composite

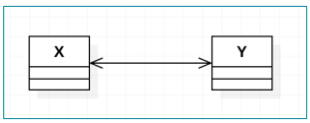
Attention :
Modification de la structure de la classe Entité

Nous allons créer un projet de démonstrations séparé pour mettre en avant l'ensemble des annotations des entités.

Les relations d'associations

- Les relations
  - Unidirectionnelle : le bean X possède une référence vers le bean Y
  - Bidirectionnelle : le bean X possède une référence vers le bean Y et réciproquement





Les relations

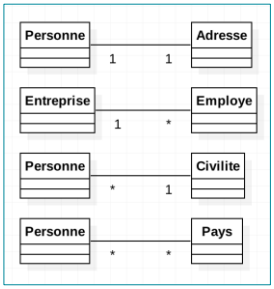
Unidirectionnelle : le bean X possède une référence vers le bean Y

Bidirectionnelle : le bean X possède une référence vers le bean Y et réciproquement

Le plus souvent dans les projets, nous privilégions les relations unidirectionnelles car plus simples et suffisantes

Cardinalités des associations

- Cardinalité
  - Indique combien d'instances vont intervenir de chaque côté d'une relation
  - One to one (1:1)
  - One to Many (1:N)
  - Many to One (N:1)
  - Many to Many (M:N)



Cardinalité

Indique combien d'instances vont intervenir de chaque côté d'une relation

One to one (1:1)

Une personne a une adresse

L'adresse n'appartient qu'à une personne

One to Many (1:N)

Une entreprise a des employés

Un employé travaille dans une seule entreprise

Many to One (N:1)

Une personne a une civilité

Une civilité est partagée par plusieurs personnes

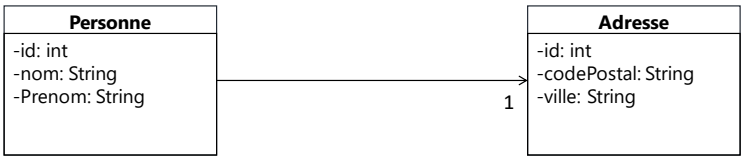
Many to Many (M:N)

Des personnes ont visité des pays

Chaque pays peut avoir été visité par plusieurs personnes

Association 1-1 unidirectionnelle

- Création de deux classes annotées avec @Entity
  - Chaque classe sera mappée avec sa propre table
- @OneToOne positionnée sur l'attribut d'association
  - Paramètres possibles : Cascade, orphanRemoval
  - @Basic(fetch = LAZY ou EAGER)
  - @JoinColumn



@OneToOne positionnée sur l'attribut associé

Paramètres possibles :

- Cascade → pour déterminer si l'enregistrement, la suppression ou la mise à jour de l'un ; impactera l'autre.
- orphanRemoval → si l'un des 2 est détruit l'autre le sera automatiquement

@Basic(fetch = LAZY ou EAGER)

→ LAZY : remonté que de l'id dans l'association. L'objet sera remonté complétement si il est appelé explicitement dans le code

→ EAGER : quand l'objet principal est remonté; celui associé l'est aussi. Ce n'est pas critique dans le cas d'une association 1-1 mais dans le cas d'une association 1-n vous finissez pas remonter la base en une fois.

@JoinColumn → quand le nom de la colonne de jointure veut être choisi.

Association 1-1 bidirectionnelle

- Création de deux classes annotées avec @Entity
  - Chaque classe sera mappée avec sa propre table
- Déclaration dans les 2 classes d'un attribut de l'autre classe.
  - Annoté avec @OneToOne
  - Pour ne pas faire boucler l'ORM utilisation de l'attribut mappedBy sur l'un des 2.
- Gestion dans le code de la bidirectionnalité.

Création de deux classes annotées avec @Entity

Chaque classe sera mappée avec sa propre table

Déclaration dans les 2 classes d'un attribut de l'autre classe.

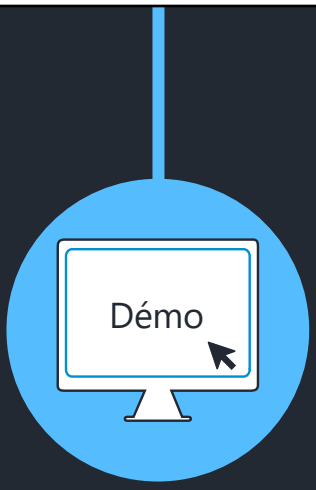
Annoté avec @OneToOne

Pour ne pas faire boucler l'ORM utilisation de l'attribut mappedBy sur l'un des 2.

Gestion dans le code de la bidirectionnalité.

- Quand en Java, nous devons gérer une association bidirectionnelle. Il faut ajouter le code pour celle-ci :
- Ajouter dans la classe Personne dans le mutateur d'adresse, la mise à jour du lien bidirectionnelle et appeler cette méthode dans son constructeur avec

paramètres.



L'association 1-1

- Regardons par la démonstration la différence entre 1-1 unidirectionnelle et bidirectionnelle

Association 1-N unidirectionnelle

- Création de deux classes annotées avec @Entity
  - Chaque classe sera mappée avec sa propre table
  - Création d'une colonne de jointure
- @OneToMany positionné sur l'attribut d'annotation
  - Paramètres possibles : Cascade, orphanRemoval
  - @Basic(fetch = LAZY ou EAGER)
  - @JoinColumn

La logique est très proche de l'association 1-1

Création de deux classes annotées avec @Entity

Personne

Adresse

Chaque classe sera mappée avec sa propre table

Création d'une colonne de jointure

Déclaration d'un attribut Adresse dans la classe Personne

Annoté avec @OneToMany

Paramètres possibles : Cascade, orphanRemoval

@Basic(fetch = LAZY ou EAGER) → la notion de Lazy et Eager a plus d'intérêt dans ce cas.

→ devons nous remonter l'ensemble des

adresses (tous les attributs) en même temps que la personne ?

→ cela dépend, s'ils sont affichés en même : oui ➔ EAGER

→ sinon, il vaudrait mieux éviter

➔ LAZY

@JoinColumn

Association 1-N bidirectionnelle

- Création de deux classes annotées avec @Entity
  - Chaque classe sera mappée avec sa propre table
  - Création d'une colonne de jointure
- Déclaration d'un attribut List<Adresse> dans la classe Personne
  - Annoté avec @OneToMany(mappedBy="...")
- Déclaration d'un attribut Personne dans la classe Adresse
  - Annoté avec @ManyToOne
- Gestion dans le code de la bidirectionnalité.

Création de deux classes annotées avec @Entity

Chaque classe sera mappée avec sa propre table

Création d'une colonne de jointure

Déclaration d'un attribut List<Adresse> dans la classe Personne

Annoté avec @OneToMany(mappedBy="...")

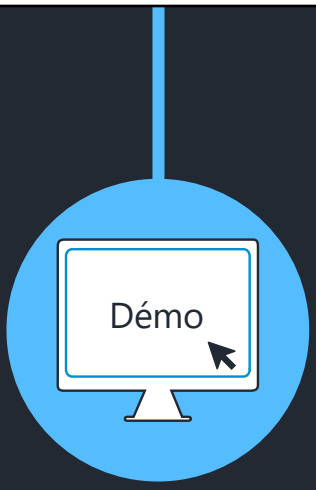
Déclaration d'un attribut Personne dans la classe Adresse

Annoté avec @ManyToOne

Gestion dans le code de la bidirectionnalité.

La logique est très proche de la version 1-1 bidirectionnelle.

La différence principale est le fait que l'annotation est différente selon qu'on est côté 1→ n ou n →1



L'association 1-N

- Regardons par la démonstration la différence entre 1-n unidirectionnelle et bidirectionnelle

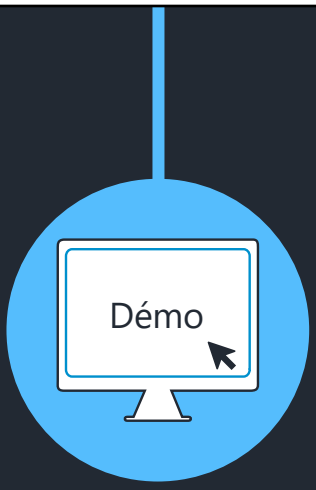
Association N-1 unidirectionnelle

- Création de deux classes annotées avec @Entity
  - Chaque classe sera mappée avec sa propre table
  - Une colonne de jointure dans la table Personne
- Déclaration d'un attribut Civilite dans la classe Personne
  - Annoté avec @ManyToOne
    - Paramètres possibles : cascade, fetch, optional

Dans ce cas, nous voulons gérer une association Unidirectionnelle côté N.
On veut que plusieurs personnes puissent avoir la même civilité.

Fonctionnellement et techniquement, c'est très similaire au 1-N unidirectionnel.
La différence, c'est que si une personne est supprimée, il ne faut pas que la civilité soit supprimée.

- Création de deux classes annotées avec @Entity
  - Personne
  - Civilite
  - Chaque classe sera mappée avec sa propre table
  - Pas de table de jointure (une colonne de jointure dans la table Personne)
- Déclaration d'un attribut Civilite dans la classe Personne
  - Annoté avec @ManyToOne
    - Paramètres possibles : cascade, fetch, optional



L'association N-1 unidirectionnelle

Mettons en place l'association Personne – Civilite dans une demonstration.

Association M-N unidirectionnelle

- Création de deux classes annotées avec @Entity
  - Chaque classe sera mappée avec sa propre table
  - Création d'une table de jointure
- Déclaration d'un attribut de type List<Pays> dans la classe Personne
  - Annoté avec @ManyToMany
    - Paramètres possibles : orphanRemoval
  - @Basic(fetch = LAZY ou EAGER)

Création de deux classes annotées avec @Entity

Personne

Pays

Chaque classe sera mappée avec sa propre table

Création d'une table de jointure

Déclaration d'un attribut de type List<Pays> dans la classe Personne

Annoté avec @ManyToMany

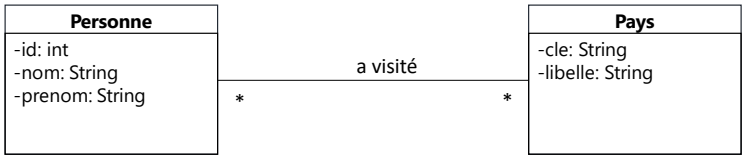
Paramètres possibles : orphanRemoval

@Basic(fetch = LAZY ou EAGER)

@JoinColumn

Association M-N bidirectionnelle

- Création de deux classes annotées avec @Entity
  - Chaque classe sera mappée avec sa propre table
  - Création d'une table de jointure
- Déclaration d'un attribut de type List<Pays> dans la classe Personne
  - Annoté avec @ManyToMany
- Déclaration d'un attribut de type List<Personne> dans la classe Pays
  - Annoté avec @ManyToMany(mappedBy="paysVisites«)



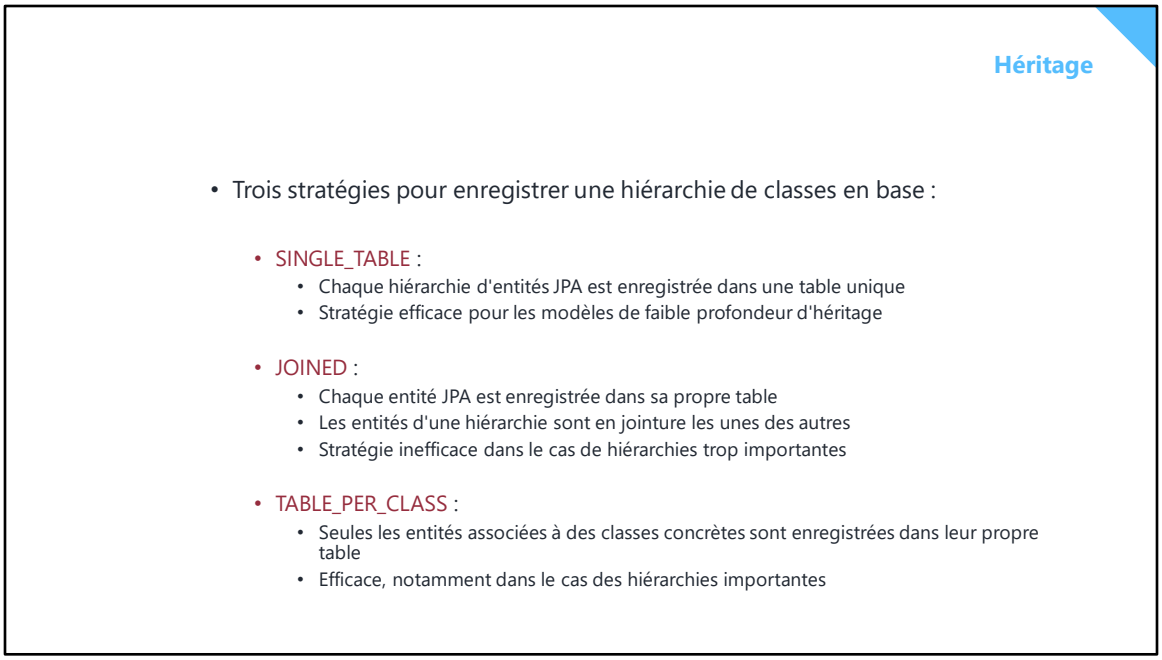
Le principe en bidirectionnel est similaire aux associations bidirectionnelles précédentes

Choisir un côté qui sera mappé sur l'autre; pour éviter que l'ORM ne boucle.

- Il est aussi possible d'ajouter des paramètres : Cascade, orphanRemoval
- Et des contraintes sur le chargement @Basic(fetch = LAZY ou EAGER)

Nous allons voir la mise en place de l'association M-N

TP – Filmothèque - Partie 07



JPA et Héritage

Comme il existe plusieurs façon de représenter un héritage de données dans le modèle relationnel, l'annotation @Inheritance dispose de l'attribut strategy pour préciser la stratégie utilisée dans le modèle de données.

Cette stratégie est une énumération du type InheritanceType et accepte les valeurs :

- SINGLE_TABLE : l'héritage est représenté par une seule table en base de données

- JOINED : l'héritage est représenté par une jointure entre la table de l'entité parente et la table de l'entité enfant
- TABLE_PER_CLASS : l'héritage est représenté par une table par entité

SINGLE_TABLE :

Cette stratégie permet de représenter en base de données un héritage avec une seule table. Une colonne contiendra un identifiant pour déterminer le type réel de la classe : on parle de la colonne *discriminante*. La table doit contenir toutes les colonnes nécessaires pour stocker les informations de la super classe et de l'ensemble des classes filles.

Son inconvénient est qu'il n'est pas possible d'ajouter des contraintes de type NOT NULL sur les colonnes représentant les propriétés des classes filles.

JOINED :

Cette stratégie permet de représenter en base de données un héritage avec une table par entité. Pour les classes filles, JPA réalisera une jointure entre la table

représentant l'entité et la table représentant l'entité de la super classe. L'implémentation est très proche de celle d'un héritage avec une seule table (seule la stratégie change) mais le schéma de la base de données est très différent.

C'est celle la plus communément utilisée, elle fonctionne aussi bien sur une base de données existante que sur une à générer.

TABLE_PER_CLASS :

Cette stratégie permet de représenter en base de données un héritage avec une table par entité. Les attributs hérités sont répétés dans chaque table. Ainsi, la notion d'héritage n'est pas exprimée dans le modèle relationnel de données.

La stratégie TABLE_PER_CLASS est plus complexe à mettre en place pour la gestion des clés primaires.

Si on prend un héritage entre la classe abstraite Vehicule, les classes filles Voiture et Camion. Pour JPA, les objets de type Voiture et Camion sont également des objets de type Vehicule. À ce titre, il ne peut pas exister deux objets avec la même clé primaire. Mais, comme ces objets sont représentés en base de données par deux

tables différentes, une colonne de type AUTO_INCREMENT en MySQL ne suffit pas à garantir qu'il n'existe pas une voiture ayant la même clé qu'un camion.

Avec, cette stratégie, il n'est pas possible d'utiliser l'annotation @GeneratedValue avec la valeur IDENTITY, On peut, par exemple, utiliser une table servant à générer une séquence de clés.

Héritage – SINGLE_TABLE

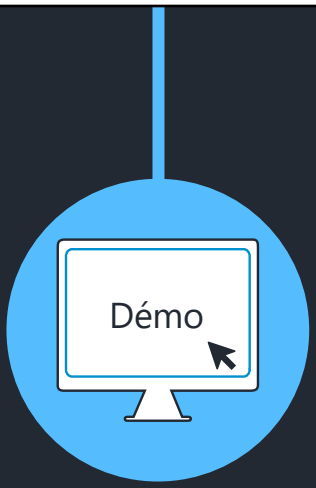
- Toute la hiérarchie de classes est enregistrée dans une seule table
  - @Entity sur chaque classe
  - @Inheritance(strategy=InheritanceType.SINGLE_TABLE) sur la classe mère
- Autant de colonnes que de champs persistants différents
- Utilisation d'une colonne supplémentaire discriminante
  - @DiscriminatorColumn(name="TYPE_ENTITE") sur la classe mère
  - @DiscriminatorValue("...") sur chacune des classes de la hiérarchie

Héritage –TABLE_PER_CLASS

- Autant de tables qu'il y a de classes concrètes annotées @Entity dans la hiérarchie
  - @Entity sur chaque classe
  - @Inheritance(strategy=InheritanceType.TABLE_PER_CLASS) sur la classe mère
- Chaque table possède
  - sa propre clé primaire
  - les colonnes correspondant aux attributs issus de l'héritage
  - ses propres attributs
- Pas de colonne discriminante

Héritage –JOINED

- Autant de tables qu'il y a de classes annotées @Entity dans la hiérarchie
  - @Entity sur chaque classe
  - @Inheritance(strategy=InheritanceType.JOINED) sur la classe mère
- Chaque table possède
  - Ses propres champs
- Les tables "filles" possèdent
  - Leurs propres champs
  - Une colonne référence la table mère
- Possibilité de définir une colonne discriminante

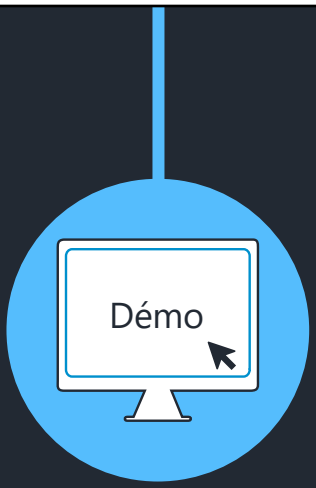


L'héritage

- Mettons en place ces 3 stratégies d'héritage

Gestion des collections de base

- Possibilité d'enregistrer une collection d'éléments simple (String, Date, Integer…) sans avoir besoin de créer une nouvelle classe Entity
- Utilisation de l'annotation @ElementCollection
- Possibilité de redéfinir le nom de la table de jointure ainsi que les colonnes
  - @CollectionTable (
    name = "…",
    joinColumns=@JoinColumn(name = "…", referencedColumnName = "…")
  - @Column(name="…")



Les collections de base

Voyons la mise en place d'une telle collection

Signature de méthodes = requêtes (Query Methods)

- Dans un Repository il est possible de définir des requêtes par le nom de méthodes et des mots clefs :

Mot clé	Traduction SQL
find…By, read…By ou get…By	SELECT
Distinct	DISTINCT
And	AND
Or	OR
OrderBy	ORDER BY
Asc, Desc	ASC, DESC
…	

Plus d'info sur : https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#repositories.query-methods.details

Cela permet de créer

- Des filtres complexes.
- Des recherches plus spécifiques

Exemple :

```
public interface PersonneRepository extends
JpaRepository<Personne, Long>{
  List<Personne> findByEmailAndNom(String email, String
prenom);
  List<Personne> findByNomAndPrenom(String nom, String prenom);
  List<Personne> findDistinctByNomOrPrenom(String nom, String
prenom);
}
```

Java Persistence Query Language (JPQL)

- Permet de requêter les entités
- Des requêtes portables
- Ressemble à du SQL mais adapté à l'univers objet

Java Persistence Query Language (JPQL)

- Syntaxe d'un SELECT :

SELECT *clause_select* FROM *clause_from* [WHERE *clause_where*] [GROUP BY *clause_group_by*] [HAVING *clause_having*] [ORDER BY *clause_order_by*]

- Syntaxe d'un UPDATE:

UPDATE *clause_update* [WHERE *clause_where*]

- Syntaxe d'un DELETE

DELETE *clause_delete* [WHERE *clause_where*]

Exemples

Select p FROM Personne p WHERE p.nom LIKE :var

La clause FROM fait référence aux entités et non aux tables.

Dans ce cas on peut aussi écrire : FROM Personne p WHERE p.nom LIKE :var

:var est appelé un paramètre nommé, il est aussi possible d'utiliser des paramètres numérotés (Select p FROM Personne WHERE p.nom LIKE ?1)

Select DISTINCT p FROM Personne WHERE p.civilite.cle = 'M'

Cette requête nécéssitera une jointure en SQL

UPDATE Personne p SET p.prenom = 'Pierre' WHERE p.dateNaissance < :date_fin

DELETE Personne p WHERE p.dateNaissance < :date_fin

JPQL et Repository

- Repository, JPQL
  - Déclarer une signature de méthode avec
  - en argument de la méthode; les paramètres de la requête
  - l'annotation @Query(« déclaration de la requête JPQL »)
- Il est possible de créer des requêtes en SQL (natif)
  - À utiliser pour des cas complexes (Vue, …)
  - Utiliser l'attribut nativeQuery = true sur @Query
- Repository et requêtes nommées
  - Déclarer sur les entités avec l'annotation @NamedQueries
  - lister les requêtes avec l'annotation @NamedQuery
  - Déclarer dans le Repository une méthode avec le même nom que la requête et les paramètres correspondant

Exemple de requête native :

```
public interface PersonneRepository extends
JpaRepository<Personne, Integer>{

  //JPQL
  @Query("select p from Personne p where p.nom = ?1")
  List<Personne> trouverPersonnesParNom(String nom);

  //Requete en SQL
  @Query(value="select * from Personne p where p.nom = ?1",
nativeQuery=true)
  List<Personne> trouverPersonnesParNomSQL(String nom);
}
```

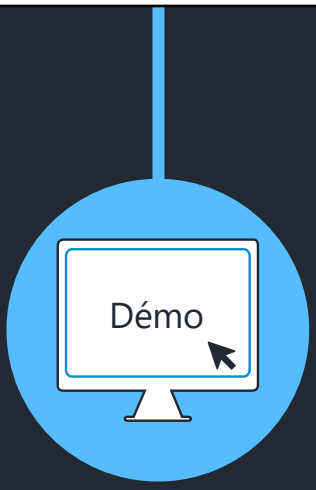
_____

Exemple requête nommée :

```
@Entity
@NamedQueries({
  @NamedQuery(name="Personne.findMessieurs",
```

```
              query="select p from Personne p where
p.civilite.cle = 'M'")
})
public class Personne { …}

public interface PersonneRepository extends
JpaRepository<Personne, Integer>{
  //Exemple NamedQuery
 List<Personne> findMessieurs();
}
```



Plus loin avec le Repository

- Faisons une passe sur les différentes possibilités supplémentaires de Repository

TP – Filmothèque - Partie 08