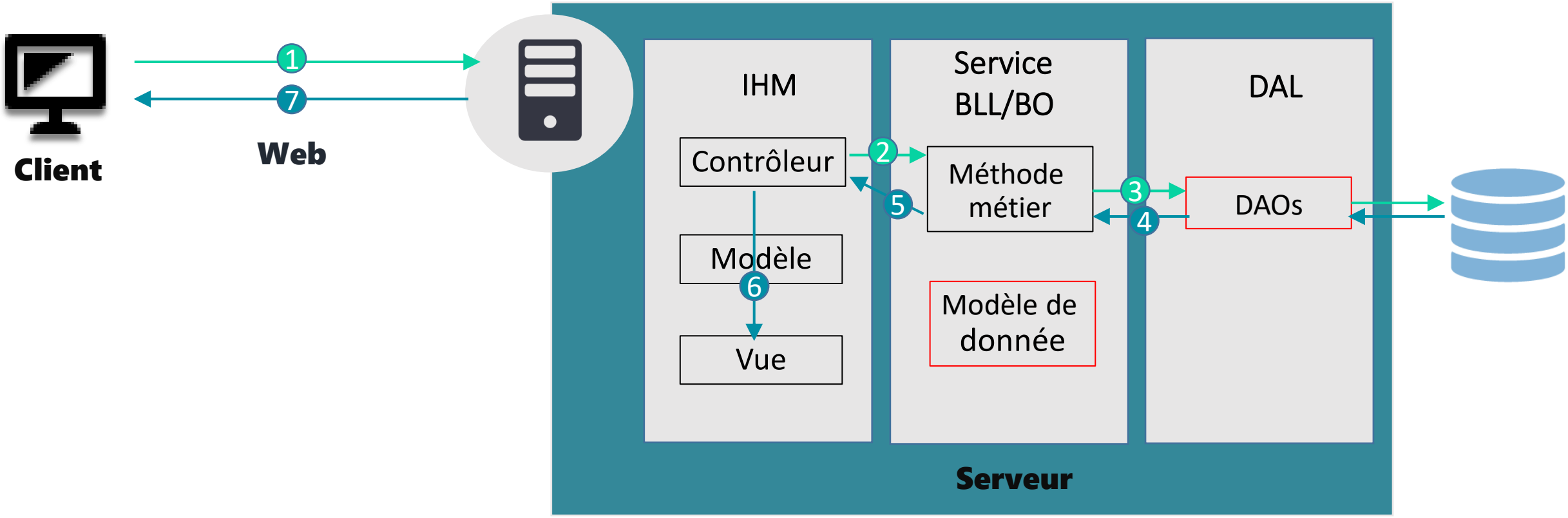
Le framework Spring

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

La couche d'accès aux données (DAL)



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

- 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)

Configurer 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

- 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

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;
…
```

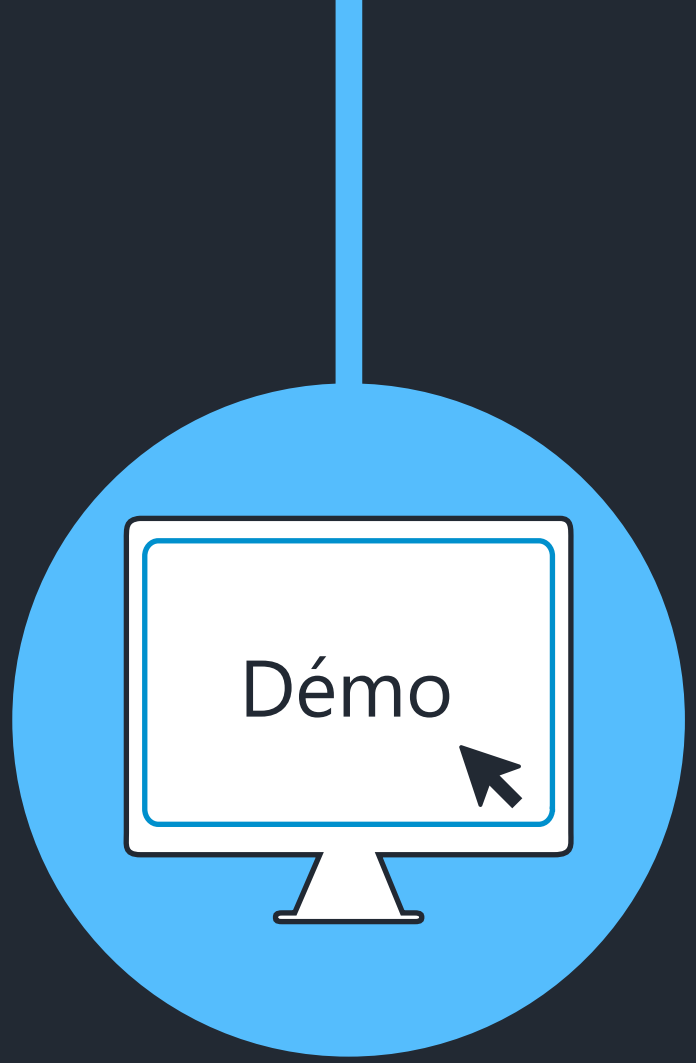
Table générée :

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

Définir une entité avec @Entity et @Id

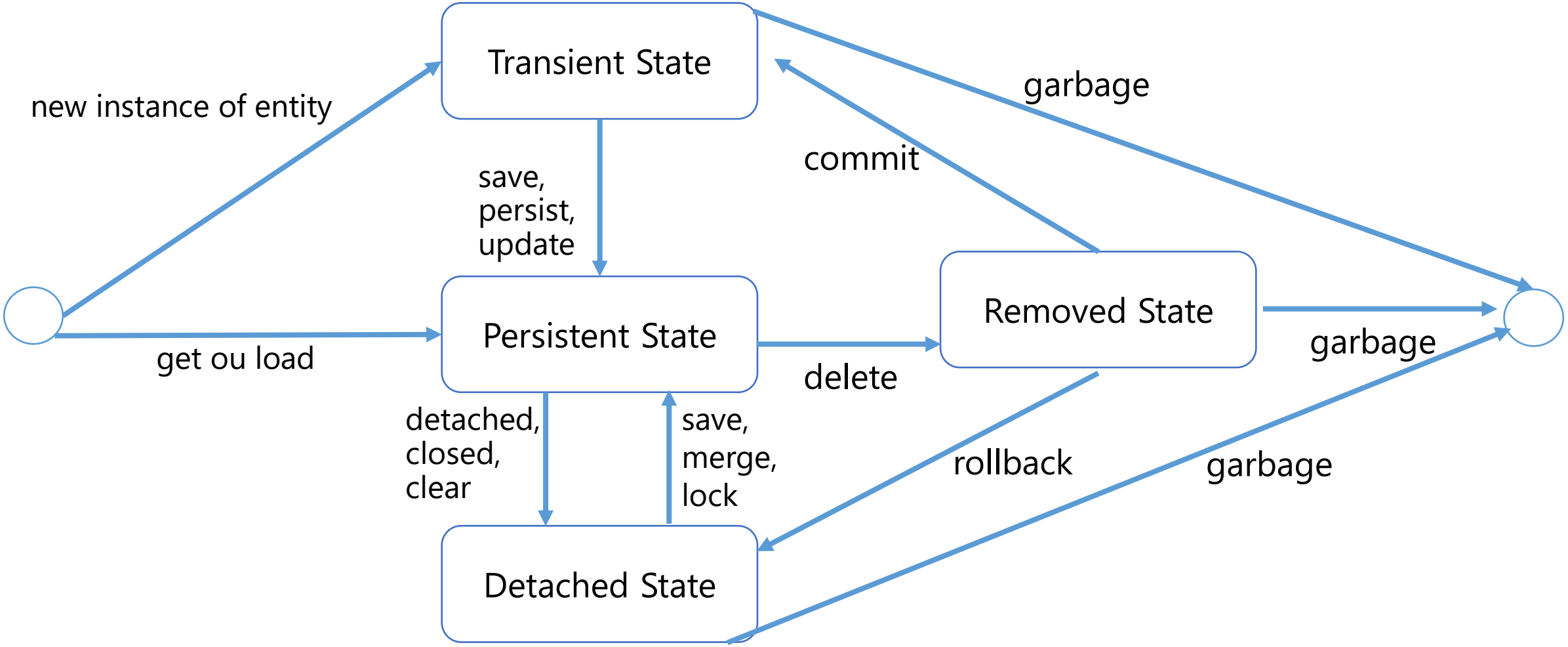
- Annotations facultatives :
  - @Table
  - @GeneratedValue
  - @Column
  - @Transient
  - @Basic

Installation de MySQL Server

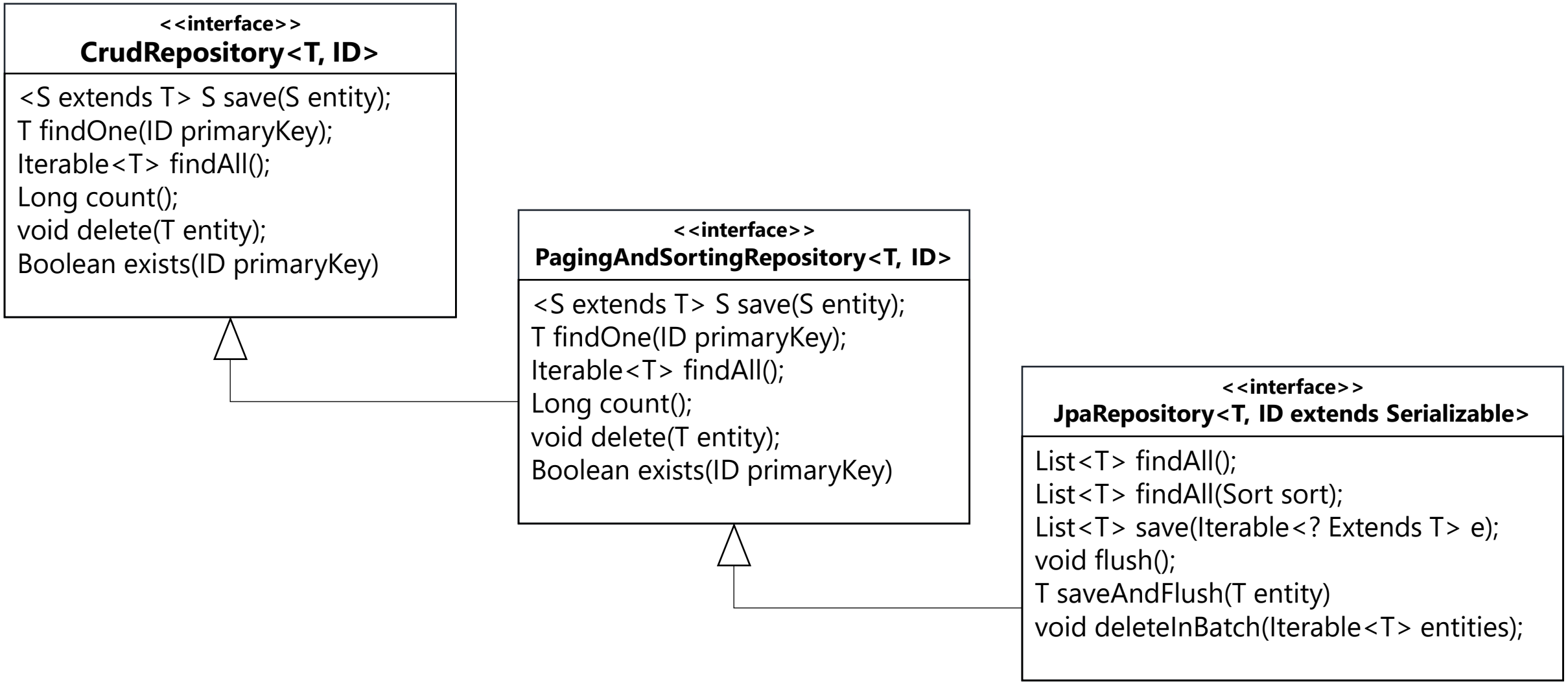


Définir une entité

Cycle de vie des entités



- Spring fournit des interfaces pour créer automatiquement les DAO:



Repository : les bases

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)

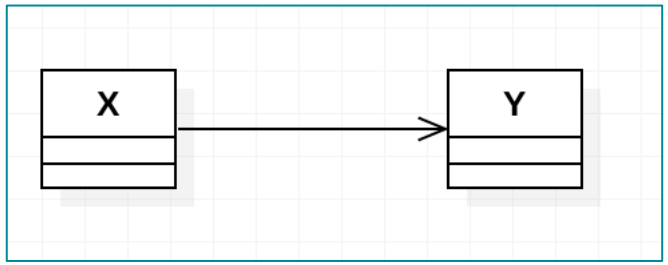
- Utilisation des annotations @EmbeddedId et @Embeddable
  - Création d'une classe décrivant la clé primaire
    - Attributs, getters et setters composant la clé primaire
    - Implémentation de Serializable
    - Constructeur sans paramètre
    - Annoté avec @Embeddable
  - Déclaration d'un attribut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

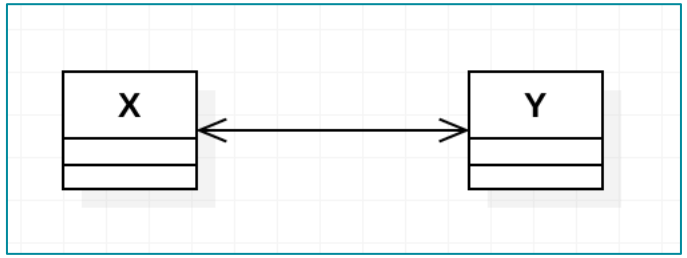
Attention :

Modification de la structure de la classe Entité

Les clés primaires composites

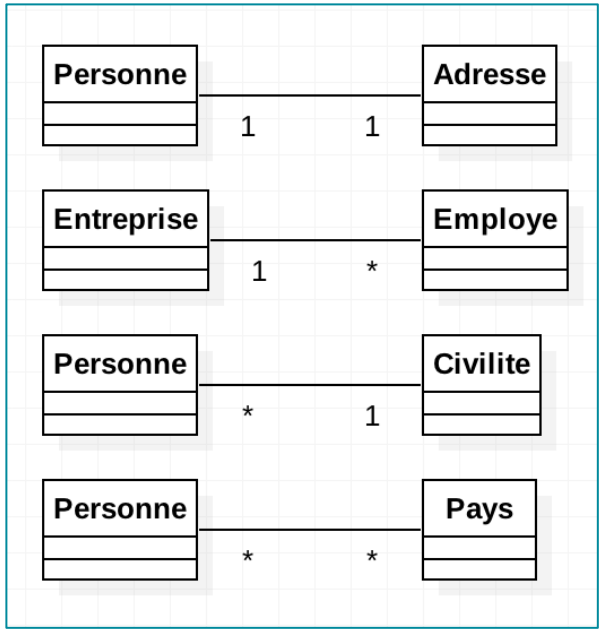
- Les relations
  - Unidirectionnelle : le bean X possède une référence vers le bean Y
  - Bidirectionnelle : le bean X possède une référence vers le bean Y et réciproquement





Cardinalités des associations

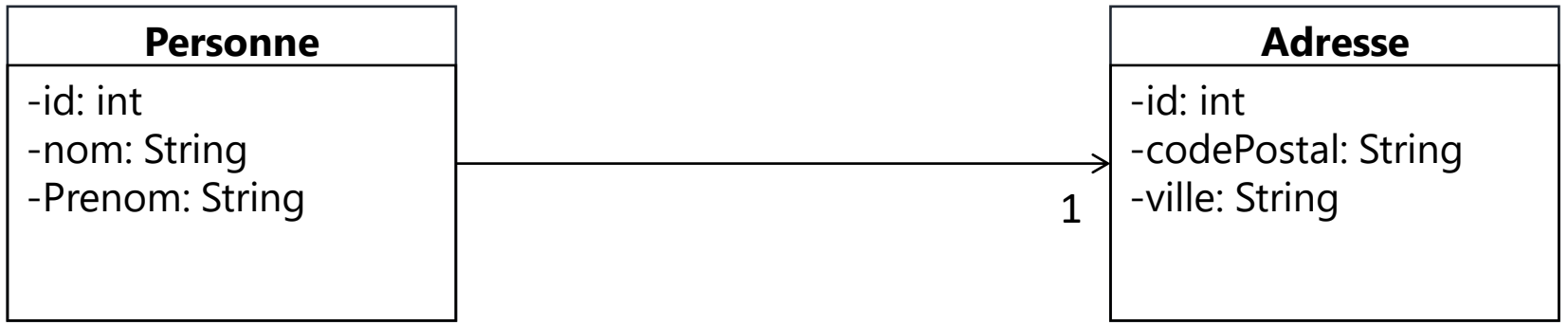
- Cardinalité
  - Indique combien d'instances vont intervenir de chaque côté d'une relation
  - One to one (1:1)
  - One to Many (1:N)
  - Many to One (N:1)
  - Many to Many (M:N)



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
    - fetch = LAZY ou EAGER
  - @JoinColumn



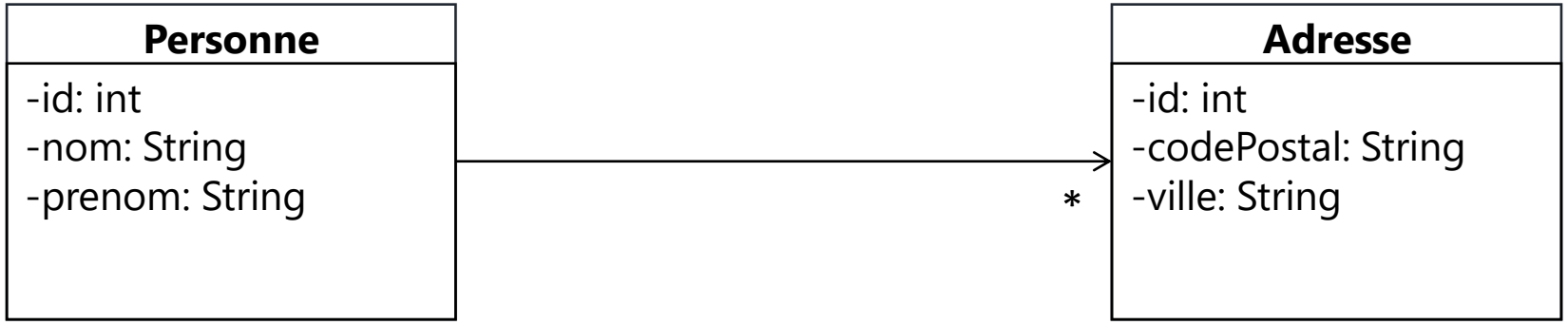
Association 1-1 bidirectionnelle

- 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.

L'association 1-1

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
    - fetch = LAZY ou EAGER
  - @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.

L'association 1-N

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

L'association N-1 unidirectionnelle

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
    - fetch = LAZY ou EAGER

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«)

L'association M-N

TP – Filmothèque - Partie 07

- Trois stratégies pour enregistrer une hiérarchie de classes en base :
  - SINGLE_TABLE :
    - Chaque hiérarchie d'entités JPA est enregistrée dans une table unique
    - Stratégie efficace pour les modèles de faible profondeur d'héritage
  - JOINED :
    - Chaque entité JPA est enregistrée dans sa propre table
    - Les entités d'une hiérarchie sont en jointure les unes des autres
    - Stratégie inefficace dans le cas de hiérarchies trop importantes
  - TABLE_PER_CLASS :
    - Seules les entités associées à des classes concrètes sont enregistrées dans leur propre table
    - Efficace, notamment dans le cas des hiérarchies importantes

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

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

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

Java Persistence Query Language (JPQL)

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

- 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*]

- 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

Plus loin avec le Repository

TP – Filmothèque - Partie 08