Spring et Java Persistence API

GLG 203/Architectures Logicielles Java

Serge Rosmorduc
serge.rosmorduc@lecnam.net
Conservatoire National des Arts et Métiers

2019–2020

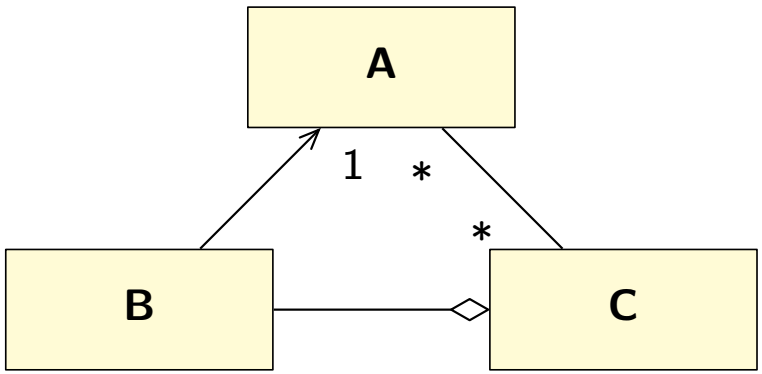
Démonstrations

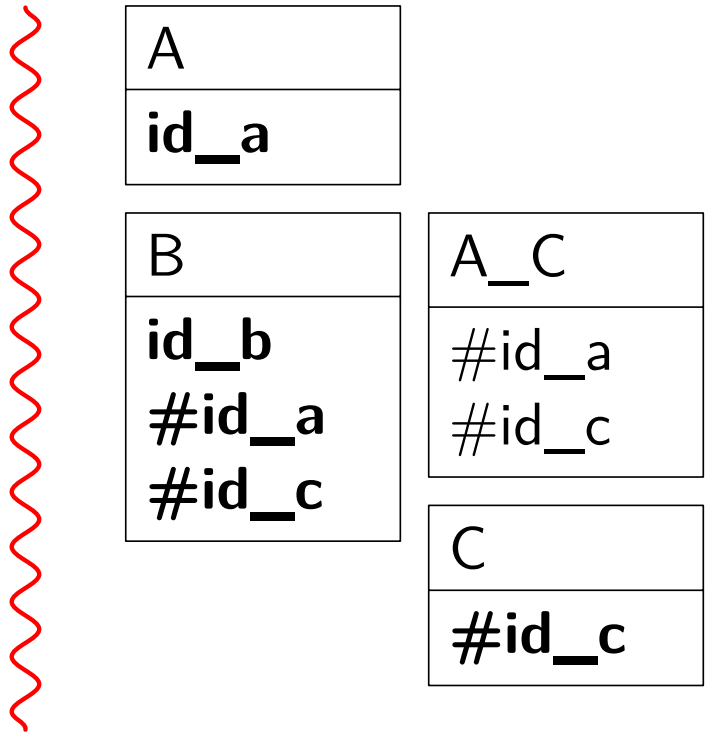
AJOUTER LE FAIT QU'IL FAUT TRAVAILLER SUR L'OBJET RETOURNÉ PAR SAVE !!!
AJOUTER @MODIFYING dans les annotations des repositories...
ENRICHIR : côté pseudo langage des repositories (Containing, par exemple) ; côté JPQL, les fonctions, la concaténation...

`https://gitlab.cnam.fr/gitlab/glg203_204_demos/06_spring_jpa.git`

Mapping Objet-Relationnel

- Résout le « object-relational mismatch »
- Java Persistence API : Spécification J2EE
- plusieurs implémentations : Hibernate, TopLink...





Bases de JPA

- Persistence Unit : paramétrage d'accès à une base
- Paramétrage logiciel possible
- normalement, configuration dans `META-INF/persistence.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence>
 <persistence-unit name="demoPU">
  <class>essainetbeans.model.Prof</class>
  <class>essainetbeans.model.Cours</class>
  <properties>
   <property name="hibernate.connection.url"
             value="jdbc:derby://localhost:1527/cnam"/>
   <property name="hibernate.connection.driver_class"
             value="org.apache.derby.jdbc.ClientDriver"/>
   <property name="hibernate.connection.password" value="test"/>
   <property name="hibernate.connection.username" value="test"/>
 </properties>
 </persistence-unit>
</persistence>
```

Entités/Objets Persistants

- **doivent** avoir un identifiant
- les modifications apportées à l'objet en mémoire seront répercutées dans la base
- annotation ou XML pour associer données java et données relationnelles
- utilisation de conventions pour réduire le travail (association implicite)

Déclaration d'objet persistant

- exemple de déclaration XML (pour Hibernate) ;
- ou annotations (plus simples).

```
<!DOCTYPE hibernate-mapping PUBLIC
   "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="demoHibernate.model">
  <class name="Article">
    <id name="id" column="idArticle">
      <generator class="native"/> <!-- ou increment -->
    </id>
    <property name="designation" column="description"/>
    <property name="prix"></property>
  </class>
</hibernate-mapping>
```

Annotations

```java
@Entity
@Table(name = "PROF")
public class Prof{
    // Définition obligatoire de l'ID
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "PROF_ID")
    private Integer profId;

    @Column(name = "PROF_NAME")
    private String profName;

    @Column(name = "PROF_FIRST_NAME")
    private String profFirstname;

    // au minimum un constructeur par défaut (éventuellement privé)
    public Prof() {
    }

... reste de la classe : POJO style.
}
```

Remarques

- Le **ID** est obligatoire.
- il peut être composite (rarement)
- beaucoup de possibilités
- **Le setter de l'ID peut être privé.**
- les indications `@Column` sont généralement optionnelles ;
- sauf pour utiliser des noms différents dans la base et dans les classes ;
- ou pour préciser des informations supplémentaires (taille de texte, etc...)
- outils de génération : permettent de créer la base à partir des classes ou inversement.
- `@GeneratedValue` n'est pas forcément débrayable (pas forcément possible de *fixer* un id donné après ça)

Déclaration des entités

- La liste des entités doit apparaître dans la définition du persistance Unit ;
- Quand on utilise un environnement j2EE ou Spring, on peut aussi se contenter de fournir le nom du package qui contient les entités.

```
// création d'un EntityManager factory
EntityManagerFactory entityManagerFactory =
    Persistence.createEntityManagerFactory("demoPU");
// Création de l'entity manager
EntityManager entityManager = entityManagerFactory.createEntityManager();
// On commence une transaction
EntityTransaction entityTransaction =
    entityManager.getTransaction();
entityTransaction.begin();
// On crée une entité
Prof p = new Prof();
p.setPrenom("Jean Michel");
p.setNom("Douin");
// On la sauve
entityManager.persist(p);
// On valide la transaction (commit)
entityTransaction.commit();
// On ferme !!!!
entityManager.close();
entityManagerFactory.close();
```

```java
// création d'un EntityManager factory
EntityManagerFactory entityManagerFactory =
    Persistence.createEntityManagerFactory("demoPU");
// Création de l'entity manager
EntityManager entityManager = entityManagerFactory.createEntityManager();
// On commence une transaction
EntityTransaction entityTransaction =
    entityManager.getTransaction();
entityTransaction.begin();
// On crée une entité
Prof p = new Prof();
p.setPrenom("Jean Michel");
p.setNom("Douin");
// On la sauve
entityManager.persist(p);
// On valide la transaction (commit)
entityTransaction.commit();
// On ferme !!!!
entityManager.close();
entityManagerFactory.close();
```

Nom de la PU

Mise à jour des objets

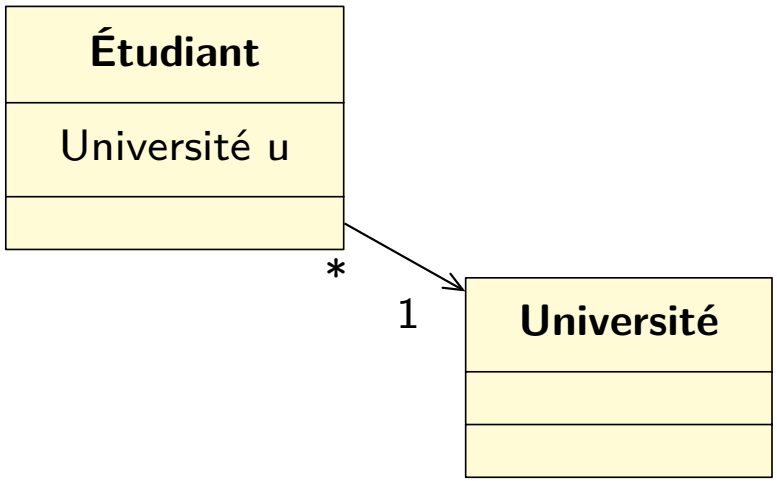
Les modifications en mémoire sont répercutées dans la base :

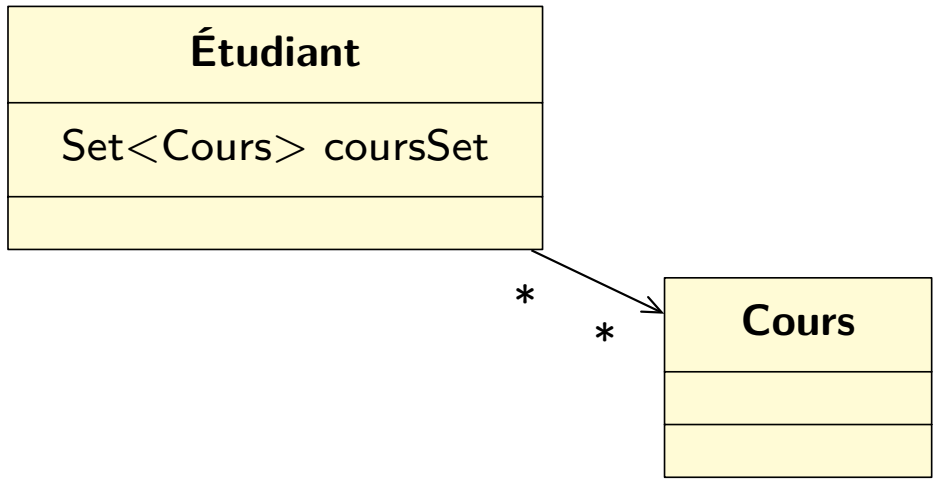
```
EntityManagerFactory entityManagerFactory =
    Persistence.createEntityManagerFactory("demoPU");
EntityManager entityManager =
    entityManagerFactory.createEntityManager();
EntityTransaction entityTransaction =
                    entityManager.getTransaction();
entityTransaction.begin();
// Trouver le prof d'id 1
Prof p = entityManager.find(Prof.class,1l);
// le modifier...
p.setPrenom(// "Jean-Michel");
// C'est tout ! maintenant on commit et on ferme.
entityTransaction.commit();
entityManager.close();
entityManagerFactory.close();
```

Plus d'annotations

Liens unidirectionnels

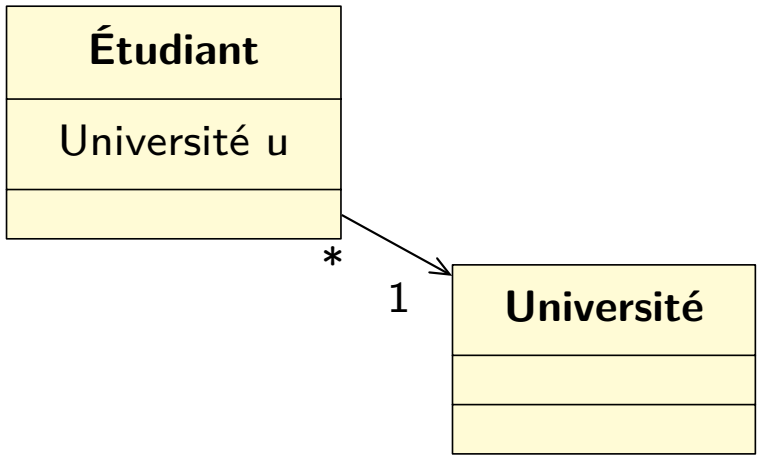
- Plus simples à réaliser en java (un seul côté à maintenir) ;
- couplage réduit ;
- 1..* ou n..n.





Liens Unidirectionnels Many-to-one

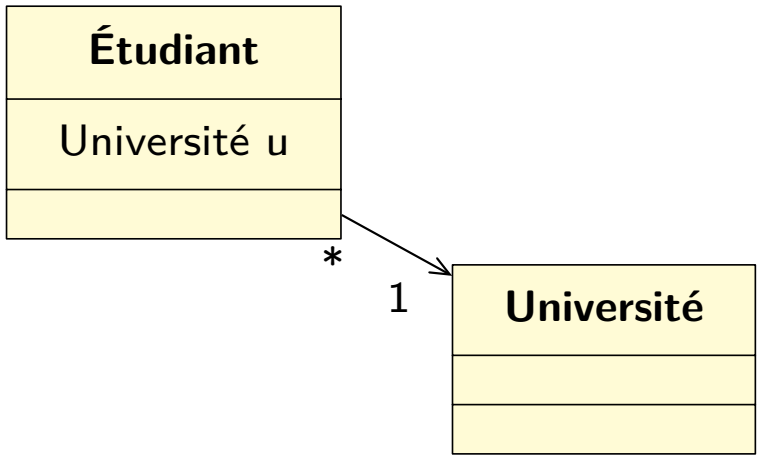
```
@Entity
public class Etudiant {
    @Id
    @Column(name="ID_ETUDIANT")
    private Long idEtudiant;

    @ManyToOne
    @JoinColumn(name="ID_UNIVERSITE")
    private Universite universite;
}
```



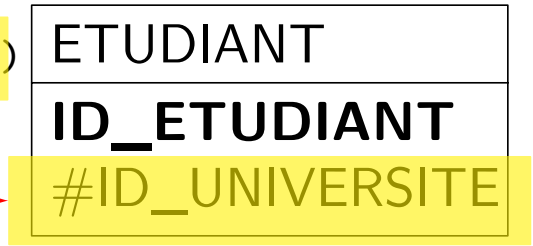
ETUDIANT
ID_ETUDIANT
#ID_UNIVERSITE

UNIVERSITE
ID_UNIVERSITE

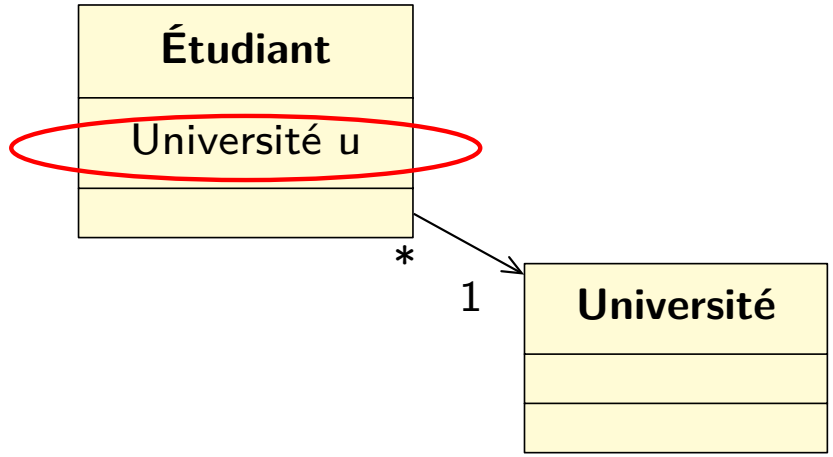
Liens Unidirectionnels Many-to-one




```
@Entity
public class Etudiant {
    @Id
    @Column(name="ID_ETUDIANT")
    private Long idEtudiant;

    @ManyToOne
    @JoinColumn(name="ID_UNIVERSITE")
    private Universite universite;
}
```

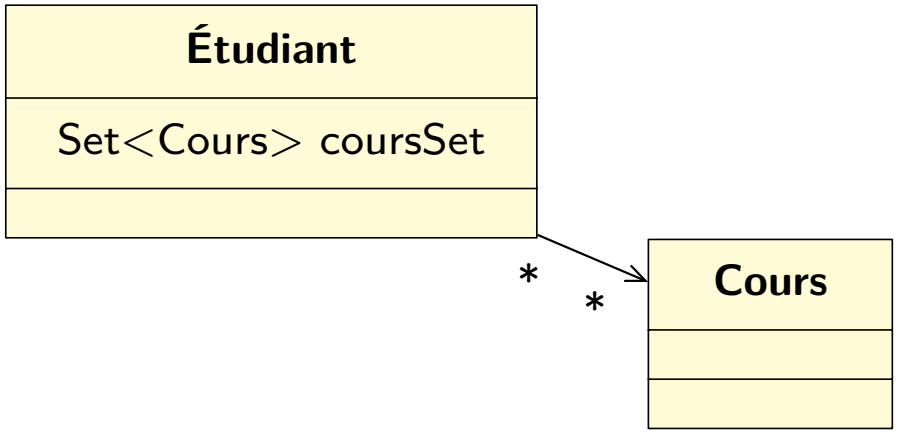


Liens Unidirectionnels Many-to-one



- Note : la classe java qui gère la relation correspond à la table qui contient la clef étrangère.
- Ici, c'est ETUDIANT.

Liens unidirectionnels Many-to-Many

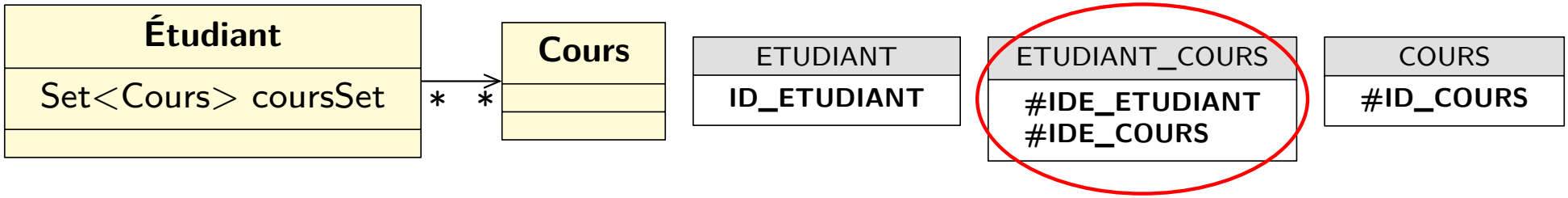


ETUDIANT
ID_ETUDIANT

ETUDIANT_COURS
#IDE_ETUDIANT **#IDE_COURS**

COURS
#ID_COURS

Liens unidirectionnels Many-to-Many




```
@Entity
public class Etudiant {
...
@ManyToMany
@JoinTable(name = "ETUDIANT_COURS",
 joinColumns = @JoinColumn(name = "IDE_ETUDIANT"},
 inverseJoinColumns =
                @JoinColumn(name = "IDE_COURS"))
    private Collection<Cours> cours;
    ...
}
```

Recherches : JPQL

Langage OO similaire à SQL (mais différent)

```
EntityManager em = ...
List<Prof> listeProfs =
    em.createQuery(
      "select p from Prof p where p.nom like :nom",
      Prof.class
    )
                  .setParameter("nom", "T%")
    .getResultList();

for (Prof p: listeProfs) {
    System.out.println(p);
}
```

API de Recherche

Sur l'EntityManager...

- `T find(Class<T> c, Object id)` : retourne l'unique entrée de classe c, et d'identifiant id ;
- `createQuery(String jpql, Class class)` : crée une requête à partir d'un texte en jpql
- Classe Criteria : construction modulaire d'un objet requête

Find

Méthode de EntityManager.

Prof p= em.find(Prof.class, 2L) ;

- retourne le prof d'id 2 (long) ;
- ou null si l'objet n'existe pas.

CreateQuery

- Méthode de EntityManager
- Retourne un objet Query
- on peut injecter des paramètre
- on peut récupérer un ou plusieurs résultats

CreateQuery

```
List<Prof> listeProfs =
   em.createQuery(
     "select p from Prof p where p.nom = :nom"
      + " and p.prenom = :prenom",
     Prof.class
     ).setParameter("nom", nom)
      .setParameter("prenom", prenom)
      .getResultList();
```

`:prenom` nom d'un paramètre à remplacer ;

`setParameter` permet de donner une valeur à un paramètre ;

`getResultList()` renvoie la liste des résultats.

Récupération du résultat

getResultList() renvoie une liste (éventuellement vide)

getSingleResult() renvoie un résultat unique

Exceptions pour `getSingleResult` :

NoResultException si pas de résultat

NonUniqueResultException si plus d'un résultat

JPQL

forme générale : **`select`** *`data`* **`from`** *`source`* **`where`** *`condition`*

- select : liste d'identifiants d'**objets**/de références à des données ;
- from : déclare les identifiants, et explique de quelles sources ils proviennent ;
- where : conditions

Exemple simple :

```
select p from Personne p where p.nom = 'Turing';
```

Résultats des requêtes

- Si le select est une unique donnée de type T : une liste d'éléments T ;
- Si le select contient plusieurs valeurs : une liste de tableaux d'objets.
  Exemple : `select s.nom, s.age from Etudiant s;`
  retourne une liste d'`Object[ ]`, où `t[0]` est le nom de l'étudiant (`String`), et `t[1]` est l'âge de l'étudiant, un `Integer`.

Propriétés des objets

On peut y accéder dans la requête, en cascade si besoin :

```
select c from Command c
    where c.client.address.city = 'Paris'
```

Join

- nécessaires quand une propriété est une collection.
- permet de nommer un élément d'une collection

```
select p from
  University u join u.professors p
where u.name= 'cnam';

select u from
  University u join u.professors p
where p.name= 'Trèves';
```

Join

On peut les combiner :

```
select u from
 University u
  join u.professors p1 join u.professors p2
where
  p1.name= 'Pollet' and p2.name= 'Trèves'
```

Join

On peut les mettre en cascade :

```
select t from
  University u
    join u.department d
    join d.teams t
where u.name= 'cnam'and t.domain='math';
```

Produit cartésien

Une requête peut porter sur des classes indépendantes, et faire l'éventuelle jointure dans le where :

```
select u, c from
  University u, Company c
where u.address.town= c.address.town;
```

Opérateur « member of »

Opérateur entre un élément e et une collection c, vrai si e membre de c

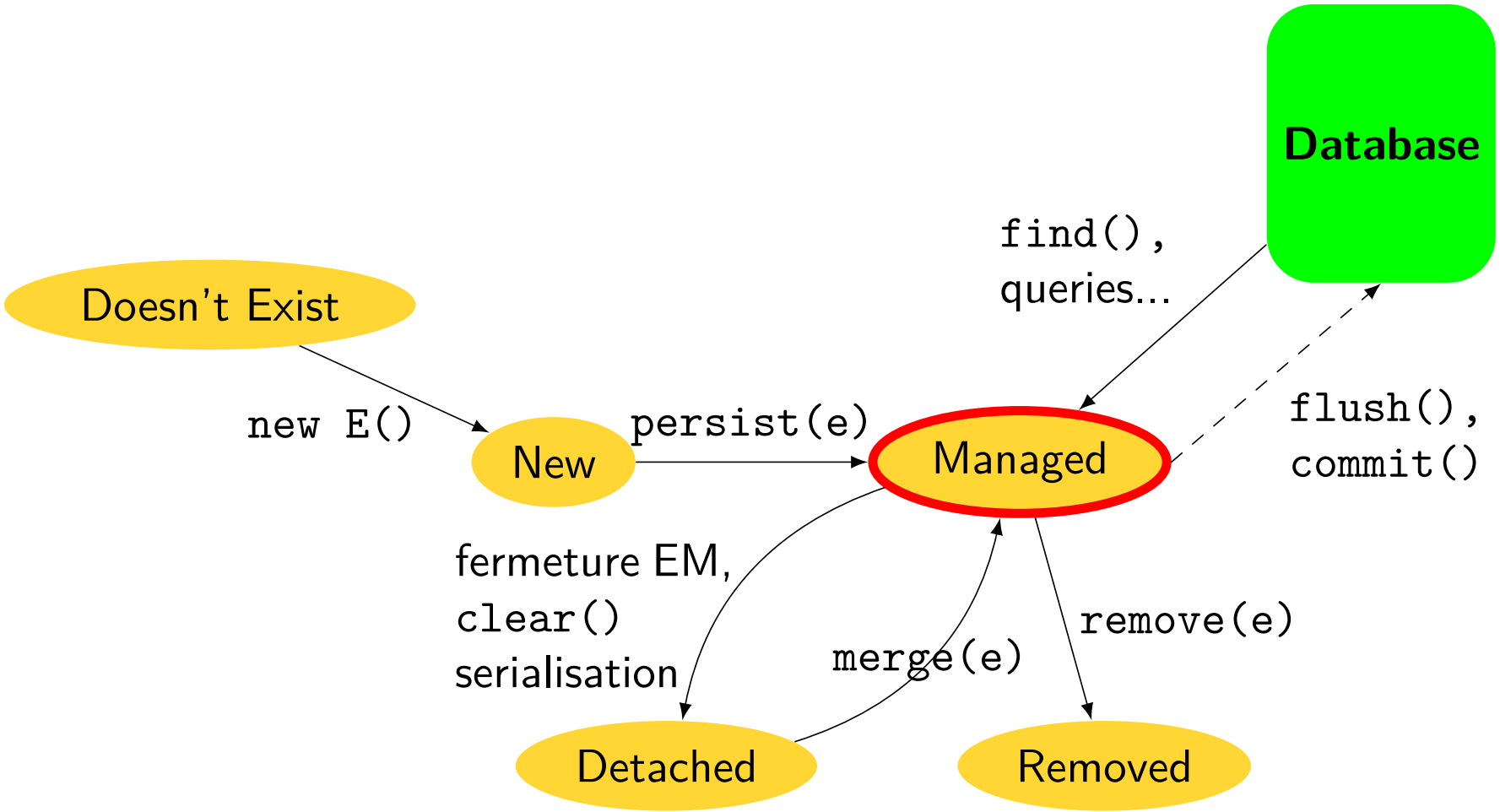
Exemple :

```
select p from Professor p, University u
  where p member of u.professors
    and u.name = 'cnam'
```

Il y a plus...

- sous-select
- any ou all comme opérateur sur les sous-select
- fonctions d'agrégats (moyenne, somme...)
- ...

Cycle de vie des objets



Cycle de vie des objets

- Pour une entité logique donnée, JPA conservera **un et un seul** objet **géré (managed).**
- Donc, *si tous les objets sont gérés* « == » et `equals` coïncident
- problème : les applications peuvent utiliser des objets détachés.

Récupération paresseuse d'objets (Lazy loading)

Si l'objet University contient l'ensemble des professeurs, etc... est-ce que

```
select u from University u;
```

chargera la totalité de la base en mémoire ?
NON

Récupération paresseuse (LAZY LOADING)

les données dans les collections ne sont récupérées que quand on en a besoin.

Lazy loading

```
List<Cours> l= em.createQuery(
  "select c from Cours c", Cours.class)
  .getResultList();
 for (Cours c : l) {
   for (Etudiant e : c.getEtudiants()) {
     String n = e.getNom();
     System.out.println(n);
   }
}
```

- Une requête pour tous les cours.
- Une requête pour les professeurs d'un cours ;
- pas de requête pour d'autres collections comprises dans `Cours`.

Limites du Lazy loading

- Fonctionne seulement quand on est connecté (EclipseLink peut ouvrir la connexion à la demande, contrairement à hibernate, mais on perd la transactionalité)
- problème des **N+1 select**

Lazy loading : N+1 Select

```
List<Cours> l= em.createQuery(
  "select c from Cours c", Cours.class)
  .getResultList();
 for (Cours c : l) {
   for (Etudiant e : c.getEtudiants()) {
     String n = e.getNom();
     System.out.println(n);
   }
}
```

- un select SQL pour les cours ;
- ensuite, un select pour chaque `c.getStudents()`
- si 5 cours, 6 select...
- beaucoup trop...

Chargement glouton

```
String jpql=
    "select distinct c from Cours c left join fetch c.students e";
Query q = em.createQuery(jpql, Cours.class);
for (Courss c: q.getResultList()) {
   System.out.println(c.getName() + "followed by ");
   for (Student s: c.getStudents()) {
     System.out.println(s);
   }
}
```

- **LEFT JOIN FETCH** force le chargement glouton des données ;
- un seul « gros » select au lieu de n+1 petits
- sans LEFT : les cours sans étudiants ne seraient pas listés
- sans DISTINCT : un cours apparaîtrait autant de fois qu'il a d'étudiants
- en JPQL, LEFT JOIN est un OUTER join

Remarque : autre solution : @EntityGraph.

Code SQL Produit : chargement paresseux

Une fois :

```
select cours0_.id as id1_0_,
       cours0_.label as label2_0_
       from Cours
cours0_
```

Pour chaque cours :

```
select etudiants0_.Cours_id as Cours_id1_1_0_,
       etudiants0_.etudiants_id as etudiant2_1_0_,
       etudiant1_.id as id1_2_1_,
       etudiant1_.nom as nom2_2_1_
from Cours_Etudiant etudiants0_ inner join Etudiant etudiant1_
on etudiants0_.etudiants_id=etudiant1_.id
where etudiants0_.Cours_id=?
```

JPA, transactions et le Web

Session JPA et requête Web

- Accès paresseux → on charge les valeurs au dernier moment ;
- … il faut une connexion à la base de données pour ça…
- mais typiquement : chargement des données dans la couche service, puis affichage par Spring MVC ;
- → problème potentiel si on place une `@Entity` complexe dans un modèle Spring MVC…

Solutions

- … ne pas placer d'entités JPA dans un modèle Spring MVC ;
- → DTO, le Service isole le modèle métier ;
- autre solution : utiliser « open session in view » ;
- `spring.jpa.open-in-view=true` dans `application.properties` ;
- attention : **mauvaise idée.**

Une note sur l'égalité

- Exemple : classe Point avec x, y
- On a envie de redéfinir equals et hashCode...
- et si x et y changent ?
- problème avec les hashmaps et les hashSets : données deviennent fausses. **les données utilisées dans equals et hashCode doivent être immuables**

Entités et valeurs

Sémantique d'entité

une entité bien définie du monde est représentée par une seule instance d'un objet.

- ses propriétés sont modifiables sans changer son identité (un étudiant déménage : il reste égal à lui-même) ;
- la méthode `equals` de `Object` convient

Entités et valeurs

Sémantique de valeur

Un objet représente une valeur (PI, "ATTENTION", 150 $...)

- deux objets représentant la même valeur sont égaux
- on a intérêt à redéfinir equals et hashCode
- ... et à rendre l'objet immuable

Exemples

- String
- Integer
- la classe LocalDate (java 8) est immuable contrairement à Date.

Et pour nos entités JPA ?

Plusieurs approches possibles :

1. Ne pas redéfinir equals et hashCode : sémantique d'entité, mais problème avec objets déconnectés et objets non encore connectés **ok,** si on fait attention à bien utiliser merge().
2. Utiliser l'ID. Proposé par l'implantation engendrée par netbeans. Fonction equals **fausse** pour les objets non enregistrés
3. Utiliser une clef métier : ok, mais il faut qu'elle existe (exemple : numéro de sécurité sociale)

Merge

- Les objets détachés ne sont pas gérés par l'entity manager
- merge vous permet de les ré-attacher

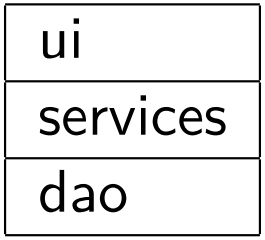
Utilisation correcte

```
Student managedStudent= em.merge(detachedStudent);
```

- retourne une version gérée (managed) de l'objet détaché
- il faut par la suite utiliser la version de managedStudent, et non l'ancienne version

Utilisation dans SpringBoot

Architecture classique :



La configuration peut être stockée en bean, dans `application.properties`...
Pour la DAO :

- On peut l'écrire à la main en injectant un `EntityManager` ;
- Spring s'occupe d'en assurer le partage si multi-thread ;
- ... mais le plus souvent on utilise les possibilités des *repositories* de Spring.

JpaRepository

Interface pour laquelle Spring sait générer à la volée une implémentation...
Il suffit de définir par exemple :

```
public interface ProfRepository
    extends JpaRepository<Prof,Long> {
}
```

et de l'injecter dans un service :

```
@Service
public class ProfService {

    @Autowired
    private final ProfRepository repository;

...
}
```

JpaRepository

- Deux arguments génériques : classe prise en charge, type de la clef ;
- méthodes fournies :

  C,U save, saveAll
  R findById, existsById, findAll, findAllById, count
  D deleteById, delete, deleteAll

  les méthodes de sauvegarde prennent en charge à la fois la création et la mise à jour par merge d'entités.
- `JpaRepository` étend deux autres interfaces : `CrudRepository`, qui est plus basique, et `PagingAndSortingRepository`, qui fournit des fonctionnalités de pagination ;

Exemple d'utilisation

```
@Service
public class ProfService {

    private final ProfRepository repository;

    // injection dans le constructeur.
    public ProfService(ProfRepository repository) {
        this.repository = repository;
    }

    public Long creerProf(String nom, String prenom) {
        Prof p = new Prof(nom, prenom);
        repository.save(p);
        return p.getProfId();
    }

    public Optional<Prof> find(Long id) {
        return repository.findById(id);
    }
...
}
```

Exemple d'utilisation

```
@Service
public class ProfService {
    public List<Prof> findAll() {
        // La méthode de repository renvoie un Iterable.
        return repository.findAll() ;
    }

}
```

Création de nouvelles requêtes

Le JpaRepository peut être étendu en suivant des conventions relativement simples :

- le nom des méthodes commence par `find, get, count...` ;
- elles retournent un Iterable (ou une collection, ou un Stream) ;
- les conditions sont introduites par « `By` » suivi de noms de propriétés : `findByAge` ;
- on peut combiner plusieurs conditions par And, Or, Not...
- et Même utiliser des intervalles ;
- Distinct permet de garantir l'unicité des résultats ;

Exemple

```
public interface EtudiantRepository
                 extends JpaRepository<Etudiant,Long> {
 public List<Etudiant> findByAddresse(String addresse);

 public List<Etudiant> findByNomOrPrenom(String nom, String prenom);
}
```

findByAddress comprend automatiquement a) que l'argument est l'adresse et b) qu'il faut faire la recherche sur l'adresse ;

findByNomOrPrenom recherche composite. Cherche l'un des deux champs.

les arguments sont reconnus.

Requêtes complexes

Quand les requêtes sont plus complexes, une première solution est de les écrire en JPQL :

```
public interface EtudiantRepository extends JpaRepository<Etudiant,Long> {
    ...

    @Query("select e from Etudiant e where :cours member of e.cours")
    public Set<Etudiant> findByCours(@Param("cours") Cours cours);
}
```

Paramètres éventuels :

- injectés par nom (annotation @Param) ;
- injectés par position (?1, ?2... dans la requête).

application.properties

Par défaut : configuré pour une base en mémoire (ajouter h2 dans les dépendances gradle).

```
# Configuration générique de Spring Boot...
debug=true

spring.datasource.driver-class-name=org.h2.Driver
# Base de donnée (h2, mais en fichier...)
spring.datasource.url=jdbc:h2:~/mabase
spring.datasource.username=sa
spring.datasource.password=

# Configuration JPA

spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto=update
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.H2Dialect
```

Configuration de application.properties

On peut externaliser tout ou partie de application.properties :

- définir des paramètres sur la ligne de commande :

```
java -jar prog.jar java --spring.datasource.url=jdbc:h2:~/foo
```

- fichier propriétés sur la ligne de commande :

```
1  java -jar prog.jar \
2     --spring.config.location=file:///monfichier.properties
```

- fichier trouvé dans un « emplacement bien connu » :
  - fichier `application.properties` dans le dossier courant ou dans un dossier `config` ;
- variable d'environnement `SPRING_CONFIG_LOCATION`

À suivre...

- héritage pour les entités, collections complexes ;
- plus de JPQL ;
- API Criteria ;
- configuration Spring ;

Guide des exemples

Tous les exemples fonctionnent par eux mêmes (ils intègrent une base de donnée H2 en mémoire).

01_jpa_seul : démonstration minimaliste de la configuration et de la programmation JPA, en dehors d'un conteneur.

02_spring_jpa : petite application spring/web/jpa. La partie web est très simple. Pour des exemples de requêtes complexes, voir les jeux de tests.

03_jpa_glouton : exemple de différence entre une requête naïve (N+1 Select) et une requête gloutonne (left join fetch).