

# Initiation aux frameworks : *Hibernate*

## Hibernate et introduction du concept d'ORM

Thomas Duchatelle (duchatelle.thomas@gmail.com)

Capgemini, pour Yves Rocher

January 3, 2013

- 1 Introduction au concept ORM
- 2 Session (EntityManager)
- 3 Relation objets / tables
- 4 Requêtes de recherches
- 5 Conclusion

# Sommaire

- 1 Introduction au concept ORM
  - Définitions
  - Architecture n-tiers
  - Concept Object-relational Mapping
- 2 Session (EntityManager)
- 3 Relation objets / tables
- 4 Requêtes de recherches
- 5 Conclusion

# Quelques définitions

- **Classe** : fichier de code, plan d'un objet (*plan d'une voiture*)

# Quelques définitions

- **Classe** : fichier de code, plan d'un objet (*plan d'une voiture*)
- **Attributs** : variables déclarées au niveau d'une classe (*couleur de la voiture*)

# Quelques définitions

- **Classe** : fichier de code, plan d'un objet (*plan d'une voiture*)
- **Attributs** : variables déclarées au niveau d'une classe (*couleur de la voiture*)
- **Accesseurs / Getter – Setter** : méthodes permettant d'accéder aux attributs d'une classe

# Quelques définitions

- **Classe** : fichier de code, plan d'un objet (*plan d'une voiture*)
- **Attributs** : variables déclarées au niveau d'une classe (*couleur de la voiture*)
- **Accesseurs / Getter – Setter** : méthodes permettant d'accéder aux attributs d'une classe
- **Instance** : réalisation d'une classe (*la voiture*)

# Quelques définitions

- **Classe** : fichier de code, plan d'un objet (*plan d'une voiture*)
- **Attributs** : variables déclarées au niveau d'une classe (*couleur de la voiture*)
- **Accesseurs / Getter – Setter** : méthodes permettant d'accéder aux attributs d'une classe
- **Instance** : réalisation d'une classe (*la voiture*)
- **singleton** : classe n'ayant qu'une seule instance



# Quelques définitions

- **Classe** : fichier de code, plan d'un objet (*plan d'une voiture*)
- **Attributs** : variables déclarées au niveau d'une classe (*couleur de la voiture*)
- **Accesseurs / Getter – Setter** : méthodes permettant d'accéder aux attributs d'une classe
- **Instance** : réalisation d'une classe (*la voiture*)
- **singleton** : classe n'ayant qu'une seule instance
- **Factory** : fabrique d'objets d'un certain type. (*usine de voitures*)

# Quelques définitions

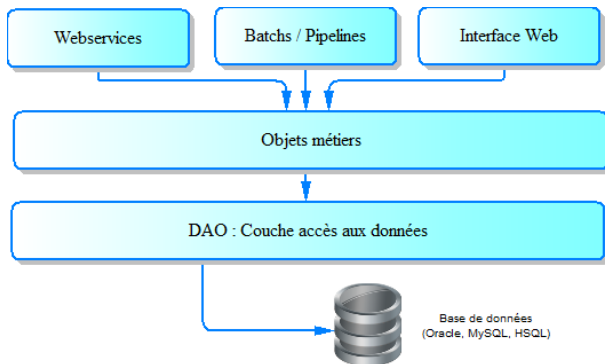
- **Classe** : fichier de code, plan d'un objet (*plan d'une voiture*)
- **Attributs** : variables déclarées au niveau d'une classe (*couleur de la voiture*)
- **Accesseurs / Getter – Setter** : méthodes permettant d'accéder aux attributs d'une classe
- **Instance** : réalisation d'une classe (*la voiture*)
- **singleton** : classe n'ayant qu'une seule instance
- **Factory** : fabrique d'objets d'un certain type. (*usine de voitures*)
- **Bean** : objets ayant des accesseurs pour chaque attributs et un constructeur par défaut

# Quelques définitions

- **Classe** : fichier de code, plan d'un objet (*plan d'une voiture*)
- **Attributs** : variables déclarées au niveau d'une classe (*couleur de la voiture*)
- **Accesseurs / Getter – Setter** : méthodes permettant d'accéder aux attributs d'une classe
- **Instance** : réalisation d'une classe (*la voiture*)
- **singleton** : classe n'ayant qu'une seule instance
- **Factory** : fabrique d'objets d'un certain type. (*usine de voitures*)
- **Bean** : objets ayant des accesseurs pour chaque attributs et un constructeur par défaut
- **Entité** : bean persistant (configuré pour être sauvegardé dans une base de données)

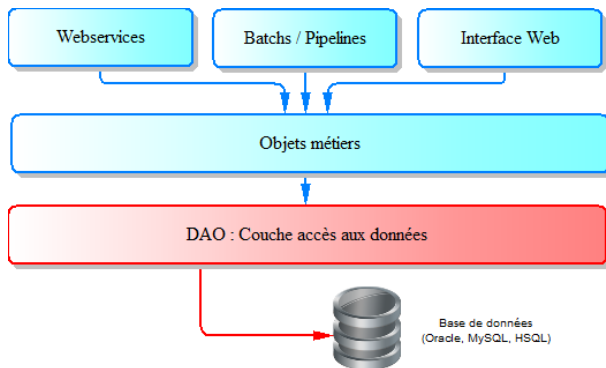
# Architecture n-tiers

## Base des applications SOA



# Architecture n-tiers

## Utilisation Hibernate ORM



# Définition de ORM

## ORM

Mapping objet-relationnel (Object Relational Mapping). Relation entre les objets et les tables.

# Définition de ORM

## ORM

Mapping objet-relationnel (Object Relational Mapping). Relation entre les objets et les tables.

## Objectif

Donne l'illusion de travailler avec une base de données **Orientée Objets**.

# Exemple sans mapping objet-relationnel

## Liste des employés

```
1  List<Employee> employees = new ArrayList<Employee>();
2  Connection conn = getConnection();
3  try {
4      PreparedStatement ps = conn.prepareStatement("SELECT ... FROM EMPLOYEE WHERE ... "
5          );
6      try {
7          ResultSet rs = ps.executeQuery();
8          try {
9              while (rs.next()) {
10                 Employee employee = new Employee();
11                 employee.setId( rs.getInt(1) );
12                 employee.setName ( rs.getString(2) );
13                 // autres parametres ...
14                 employees.add(employee);
15             }
16             finally {
17                 rs.close();
18             }
19             finally {
20                 ps.close();
21             }
22             catch (SQLException e) {
23                 // rollback...
24             }
25             finally {
26                 conn.close();
27             }
28         }
29     }
30     return list;
```



## Problématiques de l'utilisation de JDBC<sup>1</sup> :

- Gestion manuelle de la connexion : dupliquée dans chaque méthode
- Mapping *Table / Objets* réalisé manuellement, au moins 1 fois en lecture et 1 fois en écriture
- Écriture en SQL natif : nom des champs, dialecte utilisé, ...

---

<sup>1</sup>couche bas niveau de la persistance SQL en Java

## Problématiques de l'utilisation de JDBC<sup>1</sup> :

- Gestion manuelle de la connexion : dupliquée dans chaque méthode
- Mapping *Table / Objets* réalisé manuellement, au moins 1 fois en lecture et 1 fois en écriture
- Écriture en SQL natif : nom des champs, dialecte utilisé, ...

Et en utilisant un ORM comme **Hibernate** ?

---

<sup>1</sup>couche bas niveau de la persistance SQL en Java

# Exemple avec Hibernate

## ...et Spring

- Recherche

```
1 List<Employee> employees=session.createQuery("FROM _Employees").list();
```

# Exemple avec Hibernate

## ...et Spring

- Recherche

```
1 List<Employee> employees=session.createQuery("FROM Employees").list();
```

- Sauvegarde

```
1 session.saveOrUpdate(employee);
```

# Hibernate va plus loin !

Et si l'employé avait des attributs *contrat*, *chef* ?

# Hibernate va plus loin !

Et si l'employé avait des attributs *contrat*, *chef* ?

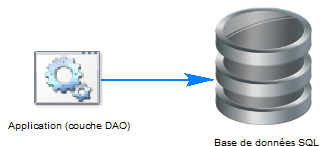
Ça ne change rien !

- pendant la recherche les attributs sont chargés et accessibles via les *getters*
- pendant la sauvegarde, les attributs sont créés ou mis à jour.

# Sommaire

- 1 Introduction au concept ORM
- 2 Session (EntityManager)
  - Le principe
  - Limitations
  - La SessionFactory
  - Un peu de code ...
  - Résumé sur l'utilisation des sessions
- 3 Relation objets / tables
- 4 Requêtes de recherches
- 5 Conclusion

# Architecture JDBC

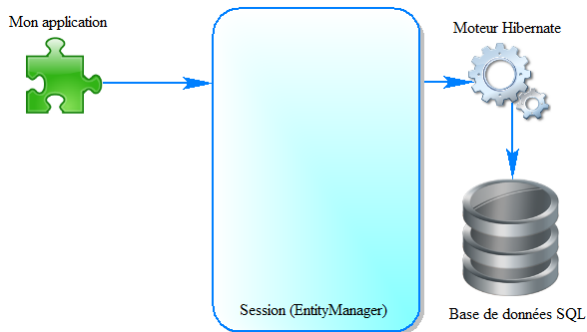


Tous les traitements métiers sont codés dans l'application.



# La session, ou EntityManager

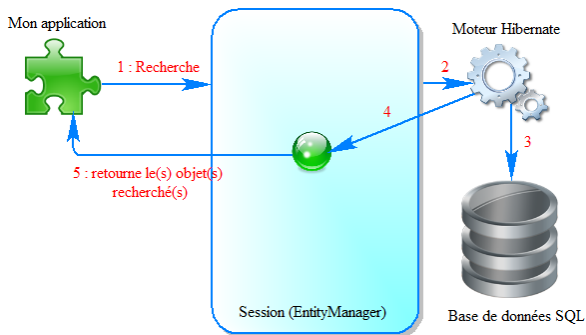
Unique interface à la base de données !



L'application n'a aucune interaction avec la base de données. L'**unique** point d'entrée est la session.

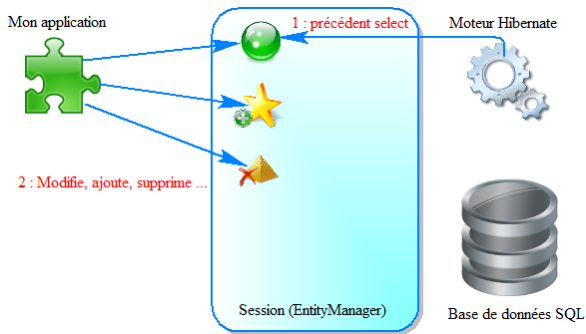
# Recherche d'objets en BDD

... mais en passant par la session



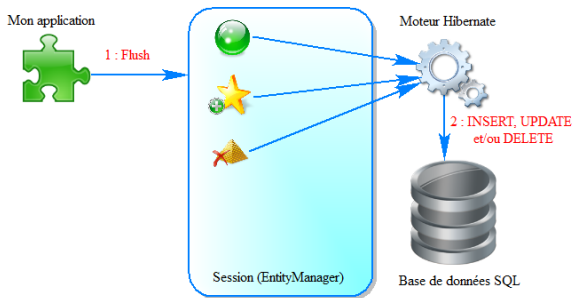
- Requête **objet** sur la *session*
- Hibernate traduit la demande et exécute un SELECT
- Les objets sont retournés, **mais restent liés à la session !**

# Modification de la session



- Modification d'entités déjà liées à la session
- Ajout de nouvelles entités à la session
- Marquage d'entités comme supprimées

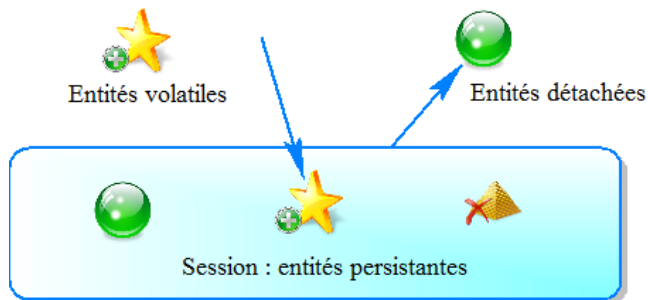
# Flush et commit



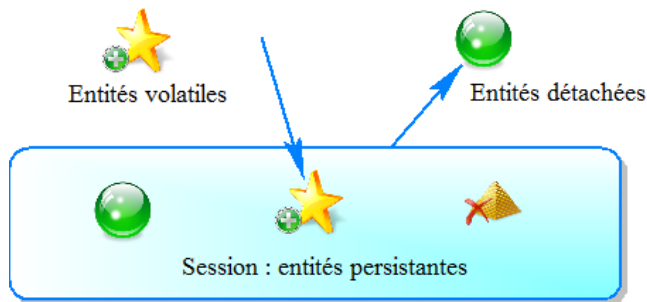
## Flush de la Session

Tous les entités liés à la session, *et leurs dépendances (attributs)*, sont créés, mis à jour ou supprimés dans la base de données.

# États des entités

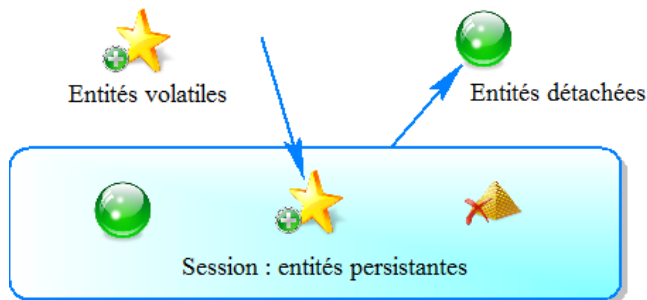


# États des entités



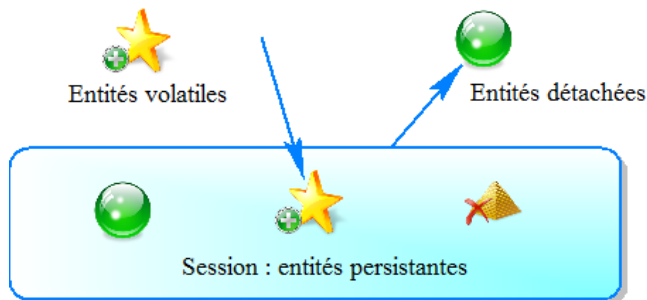
- Les nouvelles entités sont dites : **volatiles** (transient)

# États des entités



- Les nouvelles entités sont dites : **volatiles** (transient)
- les entités qui sont liées sont dites : **persistantes**

# États des entités



- Les nouvelles entités sont dites : **volatiles** (transient)
- les entités qui sont liées sont dites : **persistantes**
- les entités dont la session a été fermées sont dites : **détachées**



# Limitations de la session

## Face aux threads

La *session* est un objet **non-thread-safe**.

# Limitations de la session

## Face aux threads

La *session* est un objet **non-thread-safe**.

- ne pas placer une session en attribut d'un singleton !

# Limitations de la session

## Face aux threads

La *session* est un objet **non-thread-safe**.

- ne pas placer une session en attribut d'un singleton !

## Bonne pratique

1 session = 1 transaction.

# La *SessionFactory*

Usine de sessions

## SessionFactory

La *SessionFactory* est une fabrique de *sessions*. Elle n'est créée qu'une seule fois, au lancement de l'application.

# La *SessionFactory*

Usine de sessions

## SessionFactory

La *SessionFactory* est une fabrique de *sessions*. Elle n'est créée qu'une seule fois, au lancement de l'application.

Configuration dans `hibernate.hbm.xml` :

- Source de données (fabrique de connexions à la BDD)
- Mapping objet : relation entre les tables et les objets
- Politiques de chargement, cache, ... tout ou presque est surchargeable !

# Un peu de code ...

... sans l'aide de Spring

- Recherche uniquement (read only) :

```
1  // Obtention d'une session
2  Session session = sessionFactory.openSession();
3
4  // MON CODE ICI
5
6  // Fermeture de la session
7  session.close();
```

# Un peu de code ...

... sans l'aide de Spring

- Modification (read-write) :

```
1  // Obtention d'une session
2  Session session = sessionFactory.openSession();
3
4  // Debut de la transaction
5  session.beginTransaction();
6  try {
7      // MON CODE ICI
8
9      // Commit de la session si c'est OK
10     session.getTransaction().commit();
11
12 } catch (Exception e) {
13     // On annule tout ce qui a ete fait si une erreur s'est produite
14     session.getTransaction().rollback();
15     throw e;
16
17 } finally {
18     session.close();
19 }
```

# Sauvegarde d'une entité

## Sauvegarder ou mettre à jour

Pour sauvegarder un objet, il faut le lier à la session. Il sera inséré en BDD lors du commit de la transaction (flush).



# Sauvegarde d'une entité

## Sauvegarder ou mettre à jour

Pour sauvegarder un objet, il faut le lier à la session. Il sera inséré en BDD lors du commit de la transaction (flush).

## 1 seule entité par enregistrement

Il ne peut pas y avoir 2 instances d'une même classe avec la même clé primaire.

# Sauvegarde d'une entité

Save, persist, update, merge, ... ?

Plusieurs cas sont possibles :

- `saveOrUpdate` : pour toute entité, quelque soit son état

```
1 // Ajout d'une entite a la session
2 session.saveOrUpdate(myEntity);
```

# Sauvegarde d'une entité

Save, persist, update, merge, ... ?

Plusieurs cas sont possibles :

- `saveOrUpdate` : pour toute entité, quelque soit son état
- `save` : nouvelle entité (et enregistrement en base), pas d'identifiant

```
1 // Ajout d'une entite a la session
2 session.saveOrUpdate(myEntity);
```

# Sauvegarde d'une entité

Save, persist, update, merge, ... ?

Plusieurs cas sont possibles :

- `saveOrUpdate` : pour toute entité, quelque soit son état
- `save` : nouvelle entité (et enregistrement en base), pas d'identifiant
- `update` : entité existant déjà en base, avec son identifiant renseigné

```
1 // Ajout d'une entite a la session
2 session.saveOrUpdate(myEntity);
```

# Sauvegarde d'une entité

Save, persist, update, merge, ... ?

Plusieurs cas sont possibles :

- `saveOrUpdate` : pour toute entité, quelque soit son état
- `save` : nouvelle entité (et enregistrement en base), pas d'identifiant
- `update` : entité existant déjà en base, avec son identifiant renseigné
- `persist` : exécute immédiatement l'insertion (pas/peu de vérification d'existence)

```
1 // Ajout d'une entite a la session
2 session.saveOrUpdate(myEntity);
```

# Sauvegarde d'une entité

Save, persist, update, merge, ... ?

Plusieurs cas sont possibles :

- `saveOrUpdate` : pour toute entité, quelque soit son état
- `save` : nouvelle entité (et enregistrement en base), pas d'identifiant
- `update` : entité existant déjà en base, avec son identifiant renseigné
- `persist` : exécute immédiatement l'insertion (pas/peu de vérification d'existence)
- `merge` : remplace l'instance de même id déjà liée à la session

```
1 // Ajout d'une entite a la session
2 session.saveOrUpdate(myEntity);
```

# Supprimer une entité

```
1 // Supprime l'entité  
2 session.delete(myEntity);
```

# Charger une entité

Rechercher une entité par son identifiant base de données :

```
1 // Charge une entite a partir d'un identifiant (serialisable)
2 session.get(Employee.class, id);
```



# Utilisation d'une session

Rappels...

## Sauvegarde des entités

Pour être sauvegardée, une entité doit être **persistante** : liée à la session.  
Puis la session doit être **commitée**.

# Utilisation d'une session

Rappels...

## Sauvegarde des entités

Pour être sauvegardée, une entité doit être **persistante** : liée à la session. Puis la session doit être **commitée**.

## Rendre persistante une entité

Une entité est rendu persistante lors de l'appel des méthodes de la session : `saveOrUpdate` ou `delete`.

# Utilisation d'une session

## Rappels...

### Sauvegarde des entités

Pour être sauvegardée, une entité doit être **persistante** : liée à la session. Puis la session doit être **commitée**.

### Rendre persistante une entité

Une entité est rendu persistante lors de l'appel des méthodes de la session : `saveOrUpdate` ou `delete`.

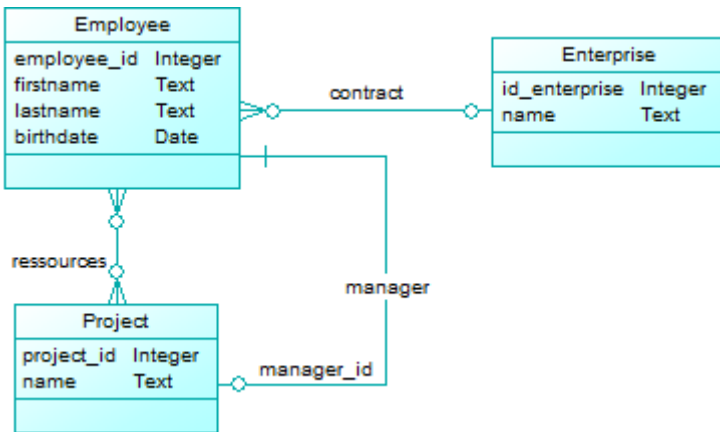
### Entités recherchées

Les entités retournées par la session suite à une recherche sont déjà persistantes.

# Sommaire

- 1 Introduction au concept ORM
- 2 Session (EntityManager)
- 3 Relation objets / tables
  - Modèle d'exemple
  - Configuration de mapping minimale
  - Personnalisations des noms
  - Relations
- 4 Requêtes de recherches
- 5 Conclusion

# Modèle entreprise et employés



# Déclaration d'une entité

## Annotations obligatoires

Seules 2 annotations sont obligatoires pour déclarer qu'une **classe est persistante** :

# Déclaration d'une entité

## Annotations obligatoires

Seules 2 annotations sont obligatoires pour déclarer qu'une **classe est persistante** :

- `@Entity` : indique a Hibernate que la classe est persistante

# Déclaration d'une entité

## Annotations obligatoires

Seules 2 annotations sont obligatoires pour déclarer qu'une **classe est persistante** :

- `@Entity` : indique a Hibernate que la classe est persistante
- `@Id` : les entités doivent obligatoirement avoir un ID



# Déclaration d'une entité

## Configuration de mapping minimale

Table minimale :

```
1 CREATE TABLE employee (  
2     id INTEGER PRIMARY KEY  
3 );
```

# Déclaration d'une entité

## Configuration de mapping minimale

Table minimale :

```
1 CREATE TABLE employee (  
2     id INTEGER PRIMARY KEY  
3 );
```

Objet correspondant :

```
1 @Entity  
2 public class Employee implements Serializable {  
3     private Integer id;  
4  
5     @Id  
6     public Integer getId() {  
7         return id;  
8     }  
9  
10    public void setId(Integer id) {  
11        this.id = id;  
12    }  
13 }
```

# Mapping des autres attributs/champs

## Nommage des champs

Par défaut, le nom des champs et tables de la BDD sont ceux des attributs et classes.

# Mapping des autres attributs/champs

## Nommage des champs

Par défaut, le nom des champs et tables de la BDD sont ceux des attributs et classes.

## Déclaration d'autres attributs

Les méthodes commençant par `get` seront considérées comme des attributs persistant. L'annotation `@Transient` annule cette définition.

# Personalisations des noms

## Table simple :

```
1 CREATE TABLE employees_table (  
2     employee_id INTEGER NOT NULL AUTOINCREMENT,  
3     name VARCHAR(255),  
4     birthdate DATE,  
5  
6     PRIMARY KEY (employee_id)  
7 );
```

# Personalisations des noms

## Table simple :

```
1 CREATE TABLE employees_table (  
2     employee_id INTEGER NOT NULL AUTOINCREMENT,  
3     name VARCHAR(255),  
4     birthdate DATE,  
5  
6     PRIMARY KEY (employee_id)  
7 );
```

## Équivalence :

```
1 @Entity  
2 @Table(name = "employees_table")  
3 public class Employee implements Serializable {  
4  
5     @Id  
6     @GeneratedValue(strategy=GenerationType.IDENTITY)  
7     @Column(name = "employee_id")  
8     public Integer getId() {...}  
9  
10    @Column(name = "name")  
11    public String getLastname() { ... }  
12  
13    public Date getBirthdate() { ... }  
14 }
```

# Définition des relations

## Les 3 types de relations principales

Exemple des associations les plus communes :

- OneToOne : relation entre une personne et son passeport

# Définition des relations

## Les 3 types de relations principales

Exemple des associations les plus communes :

- OneToOne : relation entre une personne et son passeport
- OneToMany et ManyToOne : Relations entre un régiment et des soldats



# Définition des relations

## Les 3 types de relations principales

Exemple des associations les plus communes :

- OneToOne : relation entre une personne et son passeport
- OneToMany et ManyToOne : Relations entre un régiment et des soldats
- ManyToMany : relations entre les magasins et les clients

# Relations OneToMany – ManyToOne

Un employé et son entreprise

## Structure BDD :

```
1 CREATE TABLE employee ( ...  
2     enterprise_id INTEGER REFERENCES enterprise(enterprise_id)  
3 );  
4  
5 CREATE TABLE enterprise (  
6     enterprise_id INTEGER PRIMARY KEY, ...  
7 )
```

# Relations OneToMany – ManyToOne

Un employé et son entreprise

## Structure BDD :

```
1 CREATE TABLE employee ( ...
2     enterprise_id INTEGER REFERENCES enterprise(enterprise_id)
3 );
4
5 CREATE TABLE enterprise (
6     enterprise_id INTEGER PRIMARY KEY, ...
7 )
```

## Classe Employee :

```
1 // 'enterprise_id' est le nom de la colonne clef etrangere presente dans la table
   // employee.
2 @ManyToOne(cascade=CascadeType.ALL)
3 @JoinColumn(name = "enterprise_id")
4 public Enterprise getEnterprise() { return this.enterprise; }
```

# Relations OneToMany – ManyToOne

Un employé et son entreprise

## Structure BDD :

```
1 CREATE TABLE employee ( ...
2     enterprise_id INTEGER REFERENCES enterprise(enterprise_id)
3 );
4
5 CREATE TABLE enterprise (
6     enterprise_id INTEGER PRIMARY KEY, ...
7 )
```

## Classe Employee :

```
1 // 'enterprise_id' est le nom de la colonne clef etrangere presente dans la table
   employee.
2 @ManyToOne(cascade=CascadeType.ALL)
3 @JoinColumn(name = "enterprise_id")
4 public Enterprise getEnterprise() { return this.enterprise; }
```

## Classe Enterprise :

```
1 // 'enterprise' est le nom de l'attribut dans la classe Employee
2 @OneToMany(mappedBy = "enterprise", cascade=CascadeType.ALL)
3 public Set<Employee> getEmployees() { return this.employees; }
```

# Relations ManyToMany

Des employés et des projets

## Structure BDD :

```
1 CREATE TABLE employee (  
2     employee_id INTEGER PRIMARY KEY, ...  
3 );  
4  
5 CREATE TABLE project (  
6     enterprise_id INTEGER PRIMARY KEY, ...  
7 )  
8  
9 CREATE TABLE employee_project (  
10     employee_id INTEGER,  
11     enterprise_id INTEGER,  
12     CONSTRAINT employee_project_pk PRIMARY KEY (employee_id , enterprise_id )  
13 )
```

# Relations ManyToMany

## Des employés et des projets

- Classe Employee :

```
1  // 'EMPLOYEE_PROJECT' est le nom de la table de jointure
2  // EMPLOYEE.ID est le nom de la clef etrangere table de jointure -> table
   EMPLOYEE
3  // PROJECT.ID est le nom de la clef etrangere table de jointure -> table PROJECT
4
5  @ManyToMany(cascade = CascadeType.ALL)
6  @JoinTable(
7      name="EMPLOYEE_PROJECT" ,
8      joinColumns=@JoinColumn(name="EMPLOYEE.ID" ) ,
9      inverseJoinColumns=@JoinColumn(name="PROJECT.ID" )
10 )
11 public Set<Project> getProjects() { ... }
```

# Relations ManyToMany

## Des employés et des projets

- Classe Employee :

```
1  // 'EMPLOYEE_PROJECT' est le nom de la table de jointure
2  // EMPLOYEE.ID est le nom de la clef etrangere table de jointure -> table
   EMPLOYEE
3  // PROJECT.ID est le nom de la clef etrangere table de jointure -> table PROJECT
4
5  @ManyToMany(cascade = CascadeType.ALL)
6  @JoinTable(
7      name="EMPLOYEE_PROJECT" ,
8      joinColumns=@JoinColumn(name="EMPLOYEE.ID" ) ,
9      inverseJoinColumns=@JoinColumn(name="PROJECT.ID" )
10 )
11 public Set<Project> getProjects() { ... }
```

- Classe Project

```
1  // 'projects' est le nom de l'attribut dans la classe Employee
2
3  @ManyToMany(
4      cascade = CascadeType.ALL,
5      mappedBy = "projects"
6  )
7  public Set<Employee> getEmployees() { ... }
```

# Sommaire

- 1 Introduction au concept ORM
- 2 Session (EntityManager)
- 3 Relation objets / tables
- 4 Requêtes de recherches
  - Langage HQL
  - Langage Criteria
- 5 Conclusion



# Les différents langages de requêtage

2 façons d'exécuter une requête de recherche :

- **HQL** : dérivé du langage SQL, se présente comme une chaîne de caractères. Permet aussi les UPDATE et DELETE.

# Les différents langages de requêtage

2 façons d'exécuter une requête de recherche :

- **HQL** : dérivé du langage SQL, se présente comme une chaîne de caractères. Permet aussi les UPDATE et DELETE.
- **L'API Criteria** : d'écriture des requêtes sous forme d'objets

# Le langage HQL

## Ecriture d'une requête

Une requête s'écrit :

```
1 Query query = session.createQuery(myRequest);
2 query.setParameter("prenom", "HisFirstName");
3 query.setParameter("nom", "HisLastName");
4
5 List<Employee> employees = query.list();
```

# Le langage HQL

## Les basics

Lister le contenu de la table employee

SQL

```
SELECT * FROM employee;
```

# Le langage HQL

## Les basics

Lister le contenu de la table employee

SQL

```
SELECT * FROM employee;
```

HQL

```
FROM Employee
```

# Le langage HQL

## Les basics

Lister le contenu de la table employee

### SQL

```
SELECT * FROM employee;
```

### HQL

```
FROM Employee
```

- Le terme `SELECT` est facultatif. Cette notation équivaut à "`SELECT e FROM Employee e`".
- La valeur de la clause `FROM` est **le nom de la classe** (pas de la table).

# Le langage HQL

## Les conditions

Rechercher des employés par leur nom et leur prénom

SQL

```
SELECT * FROM employee WHERE lastname = ? AND firstname =  
?;
```

# Le langage HQL

## Les conditions

Rechercher des employés par leur nom et leur prénom

### SQL

```
SELECT * FROM employee WHERE lastname = ? AND firstname =  
?;
```

### HQL

```
FROM Employee WHERE lastname = :nom AND firstname = :prenom
```



# Le langage HQL

Les conditions avec des associations \*ToOne

Rechercher les employés qui travaillent dans une entreprise (retrouvée par son nom).

## SQL

```
1  SELECT emp.*
2  FROM employee emp
3      INNER JOIN enterprise ent ON emp.enterprise_id = ent.enterprise_id
4  WHERE ent.name = ?;
```

# Le langage HQL

Les conditions avec des associations \*ToOne

Rechercher les employés qui travaillent dans une entreprise (retrouvée par son nom).

## SQL

```
1  SELECT emp.*
2  FROM employee emp
3      INNER JOIN enterprise ent ON emp.enterprise_id = ent.enterprise_id
4  WHERE ent.name = ?;
```

## HQL

```
FROM Employee WHERE enterprise.name = :enterpriseName
```

# Le langage HQL

Les conditions avec des associations \*ToMany

Rechercher des employés qui travaillent pour un chef de projets.

## SQL

```
1  SELECT DISTINCT e.*
2  FROM employee e
3      INNER JOIN employee_project ep ON e.employee_id = ep.employee_id
4      INNER JOIN project p ON ep.project_id = p.projet_id
5  WHERE p.manager_id = ?;
```

# Le langage HQL

## Les conditions avec des associations \*ToMany

Rechercher des employés qui travaillent pour un chef de projets.

### SQL

```
1  SELECT DISTINCT e.*
2  FROM employee e
3      INNER JOIN employee_project ep ON e.employee_id = ep.employee_id
4      INNER JOIN project p ON ep.project_id = p.project_id
5  WHERE p.manager_id = ?;
```

### HQL (plusieurs solutions ...)

```
1  SELECT DISTINCT p.employees FROM Project p WHERE manager = :manager
```

# Le langage HQL

## Les conditions avec des associations \*ToMany

Rechercher des employés qui travaillent pour un chef de projets.

### SQL

```
1  SELECT DISTINCT e.*
2  FROM employee e
3      INNER JOIN employee_project ep ON e.employee_id = ep.employee_id
4      INNER JOIN project p ON ep.project_id = p.projet_id
5  WHERE p.manager_id = ?;
```

### HQL (plusieurs solutions ...)

```
1  SELECT DISTINCT p.employees FROM Project p WHERE manager = :manager
```

```
1  SELECT DISTINCT e
2  FROM Employee e INNER JOIN e.projects p
3  WHERE p.manager = :manager
```

# L'API Criteria

## L'alternative au HQL...

### Criteria

Écriture de la requête sous la forme d'un objet JAVA.

# L'API Criteria

## L'alternative au HQL...

### Criteria

Écriture de la requête sous la forme d'un objet JAVA.

Recherche des employés par leur nom et leur prénom :

```
1 // SELECT * FROM employee WHERE lastname = ? AND firstname = ?;  
2 Criteria criteria = session.createCriteria(Employee.class);  
3 criteria.add(Restrictions.eq("lastname", "HisLastName"));  
4 criteria.add(Restrictions.eq("firstname", "HisFirstName"));  
5  
6 List<Employee> employees = criteria.list();
```

# Sommaire

- 1 Introduction au concept ORM
- 2 Session (EntityManager)
- 3 Relation objets / tables
- 4 Requêtes de recherches
- 5 **Conclusion**
  - Résumé
  - Fin



# Résumé

Hibernate c'est trop bien !

Les points les plus importants :

- pour déclarer une *classe persistante*, les annotations obligatoires sont : @Entity et @Id

# Résumé

Hibernate c'est trop bien !

Les points les plus importants :

- pour déclarer une *classe persistante*, les annotations obligatoires sont : @Entity et @Id
- accesseurs (getters et setters) obligatoires

# Résumé

Hibernate c'est trop bien !

Les points les plus importants :

- pour déclarer une *classe persistante*, les annotations obligatoires sont : `@Entity` et `@Id`
- accesseurs (getters et setters) obligatoires
- seules les entités liées à la session seront sauvegardées, updatées ou supprimées

# Résumé

Hibernate c'est trop bien !

Les points les plus importants :

- pour déclarer une *classe persistante*, les annotations obligatoires sont : `@Entity` et `@Id`
- accesseurs (getters et setters) obligatoires
- seules les entités liées à la session seront sauvegardées, updatées ou supprimées
- pour lier une entité à la session : `saveOrUpdate` ou `delete`

# Résumé

Hibernate c'est trop bien !

Les points les plus importants :

- pour déclarer une *classe persistante*, les annotations obligatoires sont : `@Entity` et `@Id`
- accesseurs (getters et setters) obligatoires
- seules les entités liées à la session seront sauvegardées, updatées ou supprimées
- pour lier une entité à la session : `saveOrUpdate` ou `delete`
- les entités seront sauvegardées/supprimées lors du **commit de la transaction**.

Merci, des questions ?