JPA (*Jakarta Persistence*) pour Jakarta EE

EMSI - Université Côte d'Azur
Richard Grin
Version 1.78 – 3/11/23

1

Plan de ce cours

- Présentation de JPA
- Entité
- Gestionnaire d'entités
- Compléments sur les entités : identité, associations, héritage
- Requêtes, JPQL
- Modification en volume
- Concurrence
- Adaptation à une base préexistante
- Bibliographie

2

Présentation de JPA

3

JPA

- Spécification pour la persistance des objets Java dans une base de données (BD) relationnelle
- ORM, Object-Relational mapping, fait correspondre le monde relationnel et le monde des objets
- Au-dessus de JDBC ; donc un driver JDBC du SGBD doit être installé sur le serveur d'application ou dans l'application
- Ce support étudie JPA dans Jakarta EE mais JPA peut aussi être utilisé avec Java SE

4

Principales propriétés de JPA

- Des appels simples permettent de gérer la persistance, par exemple « `persist(objet)` » pour rendre un objet persistant
- Les données de la base de données relationnelle sont traitées avec une vision objet, en utilisant les classes Java entités
- Le code Java pour la persistance est transformé en ordres SQL par JPA qui les envoie ensuite à la BD

5

Fournisseur de persistance

- L'utilisation de JPA nécessite un fournisseur de persistance qui implémente les interfaces Java de l'API
- *EclipseLink* et *Hibernate* sont des fournisseurs de persistance

6

Entité

- Classe dont les instances peuvent être persistantes
- Annotée par `@Entity`
- « entité » peut aussi désigner une instance de classe entité

7

Exemple d'entité – identificateur et attributs

```
@Entity
public class Departement {
  @Id
  @GeneratedValue
  private Long id;

  private String nom;
  private String lieu;
```

Clé primaire dans la base – obligatoire

Valeur id générée automatiquement – optionnel

Par défaut, tous les attributs sont persistants

8

Exemple d'entité – constructeurs

```
public Departement() { }

public Departement(String nom,
                   String lieu) {
  this.nom = nom;
  this.lieu = lieu;
}
```

Obligatoire

9

Exemple d'entité – une association

Un département - plusieurs employés

Autre bout de l'association dans entité `Employe`

```
@OneToMany(mappedBy="dept")
private List<Employe> employes =
    new ArrayList<>();

public List<Employe> getEmployes() {
  return employes;
}
public void addEmploye(Employe emp) {
  employes.add(emp);
}
```

Comment l'association sera traduite dans entité `Employe` ?

10

Fichier `persistence.xml`

- `META-INF/persistence.xml` : informations sur les BDs utilisées
- Contient le nom JNDI de la source de données sur le serveur, mais pas les caractéristiques de la BD, même pas son emplacement

  Mais alors, où trouve-t-on ces caractéristiques ?
- Des informations complémentaires peuvent être données par des propriétés

11

Exemple

```
<persistence version= "3.0"
  xmlns="https://jakarta.ee/xml/ns/persistence">
<persistence-unit name="employes"
                  transaction-type="JTA">
  <jta-data-source>java:app/jdbc/b1</jta-data-source>
  <properties>
    <property
        name="jakarta.persistence.schema-generation-action"
        value="create"/>
    <property
        name="eclipselink.logging.level"
        value="FINE"/>
  </properties>
</persistence-unit>
</persistence>
```

1 unité par BD

nom JNDI dans le serveur d'application

tables créées si n'existent pas

propriété non standard du fournisseur de persistance ; requêtes SQL générées par JPA enregistrées dans les logs du serveur

12

Définition d'une source de données

- Soit directement dans le serveur d'application (pas standardisé)
- Soit dans l'application,
  - de façon standard
    - par `@DataSourceDefinition`
    - dans un fichier de configuration standard (web.xml par exemple)
  - ou d'une façon particulière au serveur d'application (dans payara-resources.xml par exemple)

Exemple 1 (annotation)

```
@DataSourceDefinition(
  name = "java:app/jdbc/b1",
  className = "com.mysql.cj.jdbc.MysqlDataSource",
  portNumber = 3306,
  serverName = "localhost",
  user = "toto", password = "fflkq7dlfj",
  databaseName = "db1",
  properties= {
      "useSSL=false",
      "allowPublicKeyRetrieval=true"
  }
)
@ApplicationScoped
public class MonBean { ... }
```

Nom JNDI dans persistence.xml

Nom de la BD dans le SGBD

Propriétés non standards

Exemple 2 (dans web.xml)

```
<data-source>
  <description>Pour MySQL</description>
  <name>java:app/jdbc/b1</name>
  <class-name>com.mysql.cj.jdbc.MysqlDataSource</class-name>
  <server-name>machin.unice.fr</server-name>
  <port-number>3306</port-number>
  <database-name>db1</database-name>
  <user>toto</user>
  <password>fflkq7dlfj</password>
  <property>
    <name>useSSL</name>
    <value>false</value>
  </property>
  ... <!– Autres propriétés -->
</data-source>
```

Avantage par rapport à `@DataSourceDefinition` : le mot de passe peut être modifié au déploiement

Pourquoi un avantage ?

Gestionnaire d'entités (EM)

- Interface `jakarta.persistence.EntityManager`
- Interlocuteur principal pour le développeur
- Fournit les méthodes pour gérer les entités :
  - les rendre persistantes
  - supprimer leurs valeurs dans la BD
  - retrouver leurs valeurs dans la BD
  - …

Où se trouve le code de ces méthodes ?

Contexte de persistance

- `em.persist(`*`entité`*`)` rend *`entité`* persistante ⇔ elle sera géré par `em`
- Toute modification apportée à *`entité`* sera enregistrée dans la BD au moment du *commit*
- L'ensemble des entités gérées par un EM s'appelle un contexte de persistance (CP)

Exemple 1

```
@RequestScoped
public class DepartementFacade {
  @PersistenceContext(unitName = "employes")
  private EntityManager em;

  @Transactional
  public void create(Dept dept) {
    em.persist(dept);
    dept.setLieu("Paris");
  }
  ...
}
```

nom dans persistence.xml ; optionnel si une seule unité

EM injecté par serveur d'application

enregistré dans la BD ?

Transaction démarrée au début de la méthode et fermée à la fin

Pourquoi ?

Exemple 2

Que fait cette méthode ?

```
@Transactional
public void augmenter(String poste, int prime) {
  String requete =
    "SELECT e FROM Employe e "
    + " WHERE e.poste = :poste";
  Query query = em.createQuery(requete);
  query.setParameter("poste", poste);
  List<Employe> liste = query.getResultList();
  for (Employe e : liste) {
    e.setSalaire(e.getSalaire() + prime);
  }
}
```

Pas du SQL, c'est du JPQL ; `Employe` est une classe entité

Est-ce que les augmentations seront dans la base de données ?

Oui, car les employés récupérés dans la BD sont ajoutés automatiquement par `em` dans le CP

19

Entité

20

Conditions pour une classe entité

- Constructeur sans paramètre `protected` ou `public`
- Attribut qui représente la clé primaire dans la BD
- Pas `final`
- Méthodes ou champs liés à la persistance pas `final`
- Classe pas nécessairement `Serializable` mais c'est requis dans certaines circonstances

21

Types d'accès à une valeur persistante

- JPA peut accéder à la valeur d'un attribut de 2 façons
  - par la variable d'instance ; accès « par champ »
  - par les *getters* et *setter*s ; accès « par propriété »

22

Exemple

- Accès par champ :

```
@Id
private int id;
```

- Accès par propriété :

```
@Id
public int getId() {
  ...
}
```

Accès par champ ou par propriété, quelle est votre préférence ?

23

Quel type d'accès choisir ?

- L'accès par propriété oblige à ajouter des getters et des setters pour tous les attributs
- Choisir plutôt l'accès par champ

24

Attributs persistants

- Par défaut, tous les attributs d'une entité sont persistants
- Exception : attribut avec un champ `transient` (mot-clé Java lié à la sérialisation) ou annoté par `@Transient`

25

Cycle de vie d'une instance d'entité

- L'instance peut être
  - nouvelle : créée, mais pas gérée par un EM
  - gérée par un EM (méthode `persist`, `merge`, ou instance récupérée dans la BD)
  - détachée : a une identité dans la BD mais n'est pas gérée par un EM
  - supprimée : gérée par un EM ; données seront supprimées dans la BD au commit (méthode `remove`)

26

Table de la base de données

- Dans les cas simples, une classe ↔ une table
  - nom de la table = nom de la classe
  - noms des colonnes = noms des attributs
- Par exemple, la classe `Departement` correspond à la table `Departement` avec les colonnes `id`, `nom`, `lieu`

27

Convention plutôt que configuration

- ```
  @Entity
  public class Departement {
  ```
  Instances sauvegardées dans table Departement (casse peut être différente)
- ```
  @Entity
  @Table(name = "DEPT")
  public class Departement {
  ```
- `@Column` pour changer nom ou caractéristiques d'une colonne dans la table

28

Types persistants

- Un attribut persistant peut être d'un des types suivants :
  - « basic »
  - « Embeddable »
  - collection d'éléments de type « basic » ou Embeddable

29

Types « basic »

- `java.util.UUID` utilisé pour les identificateurs
- Types primitifs (`int`, `double`,…) ou enveloppes de type primitifs (`Integer`, `Double`,…)
- `String`, `BigInteger`, `BigDecimal`
- Types API « time » (`LocalDate`,…) ; `Date` et `Calendar` de `java.util` ; `Date`, `Time` et `Timestamp` de `java.sql` (peu utilisés)
- Énumérations
- Tableaux de `byte`, `Byte`, `char`, `Character`
- Plus généralement `Serializable` (sauvegardé comme un tout dans la base)

30

Exemples

- `private LocalDateTime dateOperationBancaire;`
-
```
// Par défaut le numéro est enregistré dans la BD
@Enumerated(EnumType.STRING)
private TypeEmploye typeEmploye;
```


31

Classe *Embeddable*

- Classe dont les données sont insérées dans les lignes de la table d'une entité

32

Exemple

```
@Embeddable
public class Adresse {
  private int numero;
  private String rue;
  private String ville;
  . . .
}
```

Pas besoin d'id puisque les données sont insérées dans celles de `Employe`

```
@Entity
public class Employe {
  @Embedded
  private Adresse adresse;
  ...
}
```

optionnel

3 colonnes pour l'adresse dans table des employés

33

Types d'`EntityManager`

34

2 types d'EM dans un serveur d'application

- Géré par l'application : créé par l'application à partir d'une fabrique d'EM injectée ; à la fin, doit être fermé par l'application
- Géré par le serveur d'application : injecté par le serveur d'application ; automatiquement fermé par le serveur d'application

35

EM géré par l'application

- Un EM géré par l'application crée son CP dès qu'il est créé
- Ce CP reste toujours lié à cet EM, il existe jusqu'à ce que l'EM soit fermé, indépendamment des transactions
- Pour que les modifications faites sur les entités du CP soient répercutées dans la BD au moment du commit d'une transaction, le CP doit être synchronisé avec la transaction

36

Synchronisation du CP dans le cas d'un EM géré par l'application

- Si une transaction est en cours quand le CP est créé, ce CP est automatiquement synchronisé avec la transaction
- Il est possible de synchroniser le CP avec une transaction en exécutant la méthode (em est l'`EntityManager` lié au CP) `em.joinTransaction()`

37

EM géré par l'application (1/2)

```
@SessionScoped
public class DeptManager {
  @PersistenceUnit
  private EntityManagerFactory emf;

  private EntityManager em;
  private Departement dept;

  public void init(int deptId) {
    em = emf.createEntityManager();
    dept = em.find(Departement.class, deptId);
  }
```

Bean CDI pour gérer un seul département

Fabrique d'EM injectée par serveur d'application

EM créé par application

38

EM géré par l'application (2/2)

```
  @Transactional
  public void addEmploye(int empId) {
      em.joinTransaction();
      Employe emp = em.find(Employe.class, empId);
      dept.add(emp);
      emp.setDepartement(dept);
  }
  ...
}
```


39

EM géré par le serveur d'application

- Le CP ne dure que le temps de la transaction ; quand la transaction se termine, le CP est fermé et toutes les entités qu'il contenait deviennent détachées
- Le CP est lié à la transaction
- Un CP peut être utilisé par plusieurs EMs différents dans une même transaction (propagation de contexte)
- Le CP est automatiquement synchronisé avec la transaction (pas besoin de `joinTransaction`)

40

Portée transaction

```
@RequestScoped
public class DeptFacade {
  @PersistenceContext
  private EntityManager em;

  @Transactional
  public void create(Dept dept) {
    em.persist(dept);
  }

  @Transactional
  public void edit(Dept dept) {
    em.merge(dept);
  }
  ...
```

EM injecté par serveur d'application

CP ne dure que le temps de la transaction donc `dept` va être détaché après la fin de la méthode `create`

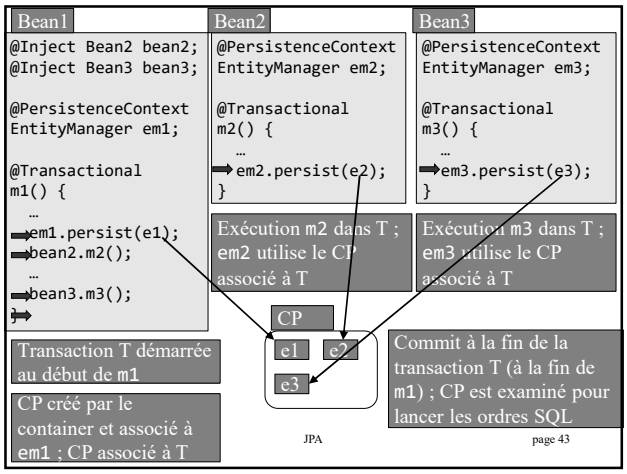
`dept` est détaché et il faut donc le remettre dans un CP pour que les modifications soient prises en compte dans la BD

41

Propagation de contexte

- Avec un EM de portée transaction, à chaque opération, l'EM regarde si un CP est déjà attaché à la transaction
- Si c'est le cas, ce CP est utilisé par l'EM, sinon l'EM demande un nouveau CP au serveur d'application et l'attache à la transaction
- Permet à plusieurs EMs d'utiliser le même CP pendant une transaction entière

42



43

Choix du type d'EM

- Plus simple de le faire gérer par le serveur d'application (propagation du CP, pas besoin de le fermer, synchronisation automatique)
- Parfois les contraintes imposées par ce choix ne conviennent pas ; en particulier, le fait que le CP est limité à une transaction, peut être gênant car il oblige à travailler avec des entités détachées
- Pour la suite du support les EMs seront supposés gérés par le serveur d'application

44

Méthodes `EntityManager`

45

Méthodes de `EntityManager` (1/2)

- `void persist(Object` *`entité`*`)`
- `<T> T merge(T` *`entité`*`)`
- `void remove(Object` *`entité`*`)`
- `<T> T find(Class<T>` *`classeEntité`*`,` `Object` *`cléPrimaire`*`)`
- `void flush()`
- `void refresh(Object` *`entité`*`)`
- `void lock(Object` *`entité`*`,` `LockModeType` *`lockMode`*`)`
- `void close()`

46

Méthodes de `EntityManager` (2/2)

- `void clear()`
- `void detach(Object entity)`
- `Query createQuery(String` *`requête`*`)`
- `Query createNamedQuery(String` *`nom`*`)`
- `Query createNativeQuery(String` *`requête`*`)`
- `void joinTransaction()`

47

`flush()`

Quand INSERT ?
Quand UPDATE ?
Quand DELETE ?

- Enregistre dans la BD les modifications effectuées sur les entités d'un CP
- L'EM lance les commandes SQL pour modifier la BD (INSERT, UPDATE ou DELETE)
- Automatiquement effectué à chaque commit et, optionnellement, avant les requêtes

`flush()` = commit ?

48

refresh(*entité*)

- *entité* doit être gérée (faire un merge en cas de besoin)
- Données de la BD sont copiées dans l'entité (qui doit être dans le CP), écrasant les éventuelles modifications faites dans cette entité
- Pour avoir la dernière version des données enregistrées dans la BD

49

Contexte de persistance - cache

- Un CP est un cache pour les accès à la BD
- `find`, et `Query` qui retourne des entités entières, ramènent les données du CP si elles y sont déjà
- Attention, les données du CP ne tiennent pas compte des modifications faites sans passer par l'EM
- Peut poser des problèmes ; un `refresh` peut être la solution
- Attention au cache de 2ème niveau partagé par tous les EMs d'une même unité de persistance

50

Entité détachée

- Une entité gérée par un EM peut être détachée de son CP Quand ?
- Par exemple par la méthode `detach`, ou si le CP est supprimé à la fin d'une transaction
- Cette entité détachée peut être modifiée
- Pour enregistrer ces modifications dans la BD il est nécessaire de rattacher l'entité à un CP par la méthode `merge`

51

merge(a)

- Sert à mettre dans le CP une entité détachée référencée par a
- Attention, en fait, la méthode `merge` crée une copie de l'entité, et c'est elle qui est mise dans le CP
- La méthode `merge` retourne cette copie
- Si l'application utilise les données de a après un `merge`, il faut souvent écrire ce type de code :

```
a = em.merge(a);
```


52

merge – une erreur à ne pas faire

- On veut ajouter un employé à dept dans la BD :

```
em.merge(dept);
dept.addEmploye(emp);
```

Quel est le problème ?

- L'ajout du nouvel employé ne sera pas enregistré dans la BD car l'objet référencé par `dept` n'est pas géré ; le bon code :

```
dept = em.merge(dept);
dept.addEmploye(emp);
```


53

Différence entre persist et merge

- `persist` sert pour les entités dont les données ne sont pas déjà dans la BD :
  - traduit par un INSERT
  - si l'entité a un id qui existe déjà dans la base, une exception est levée
- Au contraire, `merge` sert le plus souvent pour les entités qui correspondent à des données déjà dans la BD et il est alors traduit par un UPDATE

54

Opérations de EM et transactions

- Les opérations qui modifient des données dans la BD doivent être dans une transaction (persist, merge, remove)
- Les opérations qui ne modifient pas les données (find, utilisation de Query) peuvent ne pas être dans une transaction ; les entités récupérées sont alors détachées

Conversations longues

- Pas bon d'avoir une transaction longue Pourquoi ?
- Pour l'éviter, on peut
  - découper la conversation en plusieurs transactions, ce qui va générer des entités détachées et rendre plus complexe l'implémentation des rollbacks
  - travailler avec un EM géré par l'application
- Utiliser un CP désynchronisé peut aussi être utile (CP qui ne se synchronise que lorsque `joinTransaction()` est lancé (pas étudié dans ce cours)

Identité des entités

Clé primaire

- L'attribut annoté `@Id` correspond à la clé primaire dans la table associée
- Ne jamais le modifier dès que l'entité représente des données de la base

Type de la clé primaire

- Type primitif Java
- Classe qui enveloppe un type primitif
- `java.lang.String`
- `java.math.BigInteger`
- `java.util.UUID`
- (`java.math.BigDecimal`)
- (`java.util.Date`)
- (`java.sql.Date`)

Génération automatique de clé

- Pour les clés de type nombre entier
- `@GeneratedValue` indique que la clé primaire sera générée par le SGBD
- Cette annotation peut avoir un attribut `strategy` qui indique comment la clé sera générée
- La valeur de l'id est mise automatiquement dans l'entité soit après un `persist`, soit après un flush

Exemples

```
@Id
@GeneratedValue(
  strategy = GenerationType.SEQUENCE,
  generator = "EMP_SEQ")
private Long id;
```


61

Clé composite

- Pas recommandé, mais une clé primaire peut être composée de plusieurs colonnes
- Peut arriver quand la BD existe déjà, en particulier quand la classe correspond à une table association (association M:N)
- 2 possibilités :
  - `@EmbeddedId` et `@Embeddable`
  - `@IdClass`

62

- Faire TP 4

63

Associations

64

Rappels

- Exemples : association entre
  - les départements d'une entreprise et leurs employés
  - les cours et les étudiants

Quels types pour ces 2 exemples ?

- Une association entre entités peut être :
  - de type 1-1, 1-N, N-1, M-N
  - uni ou bidirectionnelle

Signification ?

65

Bout propriétaire

- Un des 2 bouts d'une association bidirectionnelle est dit propriétaire de l'association
- Pour les associations 1 – N, c'est celui qui correspond à la table qui contient la clé étrangère

Bout 1 ou N ?

- L'annotation du bout qui n'est pas propriétaire contient l'attribut `mappedBy`
  Cet attribut donne le nom de l'attribut de l'autre bout qui est propriétaire

66

Exemple

- Dans la classe `Employe` :

```
@ManyToOne
private Departement departement;
```

- Dans la classe `Departement` :

```
@OneToMany(mappedBy = "departement")
private Collection<Employe> employes;
```

Quel est le bout propriétaire ?

Dans quelle table sera la clé étrangère ?

Nom de la colonne clé étrangère : `DEPARTEMENT_ID`

67

Association bidirectionnelle - JPA

- Le développeur est responsable de la gestion correcte des 2 bouts de l'association
- Pour faciliter la gestion des 2 bouts, il est commode d'ajouter une méthode qui effectue tout le travail

68

Exemple

- Dans les associations 1-N bidirectionnelle, le bout « 1 » peut comporter ce genre de méthode :

```
public void ajouterEmploye(Employe e) {
  Departement d = e.getDept();
  if (d != null) {
    d.employes.remove(e);
  }
  this.employes.add(e);
  e.setDept(this);
}
```

Dans quelle classe est ce code ?

Qui veut expliquer ce code ?

69

Erreur à ne pas faire

- Si on ne modifie que le bout *non propriétaire* d'une association bidirectionnelle (on oublie de modifier le bout propriétaire), l'association n'est pas enregistrée dans la base de données au moment du commit
- Cette erreur correspondrait à quoi pour l'association `Departement` - `Employe` ?

70

Configuration

- Si une valeur par défaut ne convient pas, ajouter une annotation
- Par exemple, pour indiquer le nom de la colonne clé étrangère :

```
@ManyToOne
@JoinColumn(name="DEPT")
private Departement departement;
```

Ce code est dans quelle classe ?

Que signifie l'annotation `@JoinColumn` ?

71

Association M:N

- Traduite par 1 ou 2 collections annotées par `@ManyToMany`
- Pourquoi « 1 ou 2 » ?
- On peut choisir le bout propriétaire si bidirectionnel

72

Exemple - bidirectionnel

```
@ManyToMany
private Collection<Projet> projets;
```

Dans quelle classe ?

```
@ManyToMany(mappedBy = "projets")
private Collection<Employe> employes;
```

Quel est le bout propriétaire ?

Dans quelle table sera la clé étrangère ?

Dans quelle classe ?

- `@JoinTable` sert à donner le nom de la table association et des colonnes clés étrangères de cette table si les valeurs par défaut ne conviennent pas

73

Association M:N avec information portée par l'association

- Exemple :
  Association entre les employés et les projets ; un employé a une fonction dans chaque projet auquel il participe

  Où mettre la fonction dans le modèle objet ?
- Association M:N traduite par une classe association

74

Classe association

- Chaque classe qui participe à l'association peut avoir une collection d'objets de la classe association

  Pourquoi « peut avoir » ?
- 2 possibilités pour la classe association :
  - un attribut identificateur (`@Id`) unique (le plus simple)
  - un attribut identificateur par classe participant à l'association
- Association *bidirectionnelle* pour le code des transparents suivants

75

Code classe association (début)

```
@Entity
public class Participation {
  @Id @GeneratedValue
  private int id;

  @ManyToOne
  private Employe employe;
  @ManyToOne
  private Projet projet;

  private String fonction;
```


76

Code classe association (fin)

```
public Participation() { }
```

Obligatoire ?

```
public Participation(Employe employe,
    Projet projet, String fonction) {
  this.employe = employe;
  this.projet = projet;
  this.fonction = fonction;
}
// Accesseurs pour employe, projet et
// fonction
...
```


77

Classes `Employe` et `Projet`

```
@Entity public class Employe {
  @Id private int id;
  @OneToMany(mappedBy="employe",
             cascade=CascadeType.ALL)
  private Collection<Participation> participations;
  . . .
}
```

ALL = PERSIST, MERGE, REMOVE, REFRESH, DETACH

```
@Entity public class Projet {
  @Id private int id;
  @OneToMany(mappedBy="projet",
             cascade=CascadeType.REMOVE)
  private Collection<Participation> participations;
  . . .
}
```

2 types de cascade, juste pour montrer 2 possibilités

78

Pas de persistance par transitivité automatique avec JPA

- Par défaut, JPA n'effectue pas automatiquement la persistance par transitivité :
  rendre persistant un département ne suffit pas à rendre persistants ses employés
- La cohérence des données de la BD repose donc sur le code de l'application

Cohérence des données

- Classe `Departement` contient une collection d'`Employe`
- Au moment du commit, si un département est persistant et si un des employés du département ne l'est pas, une `IllegalStateException` est lancée
- L'attribut `cascade` des associations peut simplifier le code

Cascade

- Les opérations `persist`, `merge`, `remove`, `refresh`, `detach` peuvent être transmises par transitivité à l'autre bout d'une association
- Exemple :

```
@OneToMany(
  cascade = { CascadeType.PERSIST,
              CascadeType.MERGE },
  mappedBy = "client")
private Collection<Facture> factures;
```

Dans quelle classe est ce code ?

Qu'indique l'attribut cascade ?

Récupération des entités associées

- Lorsqu'une entité est récupérée depuis la BD (`Query` ou `find`), est-ce que les entités associées sont aussi récupérées ?
- Si elles sont récupérées, est-ce que les entités associées à ces entités sont elles aussi récupérées ?
- Le risque est de récupérer un très grand nombre d'entités inutiles

EAGER ou LAZY

- JPA laisse le choix de récupérer ou non immédiatement les entités associées
- Mode `EAGER` : les données associées sont récupérées immédiatement
- Mode `LAZY` (*lazy loading*) : les données associées ne sont récupérées que lorsque c'est vraiment nécessaire

Récupération tardive

- Supposons une association en mode lazy entre un département et ses employés
- On récupère un département
- Plus tard on veut avoir le nom d'un employé de ce département
- Cet employé est récupéré automatiquement par l'EM *si l'entité département n'est pas détachée* (requête SQL SELECT lancée automatiquement)
- Et si le département est détaché ?
- Il faut faire avant un merge du département (ou bien lancer une requête JPQL explicite)

Mode de récupération par défaut des entités associées

- Mode EAGER pour les associations OneToOne et ManyToOne
- Mode LAZY pour les associations OneToMany et ManyToMany

Vous voyez une raison pour ce choix ?

85

Indiquer le type de récupération des entités associées

- L'attribut `fetch` permet de modifier le comportement par défaut

Signification ?

-
```
@OneToMany(mappedBy = "departement",
           fetch = FetchType.EAGER)
private Collection<Employe> employes;
```
- Il est rare de vouloir toujours récupérer une collection en mode `EAGER` ; le plus souvent on la récupère « au coup par coup », lors d'une requête « select » (`Query`) ou d'un find

86

Héritage

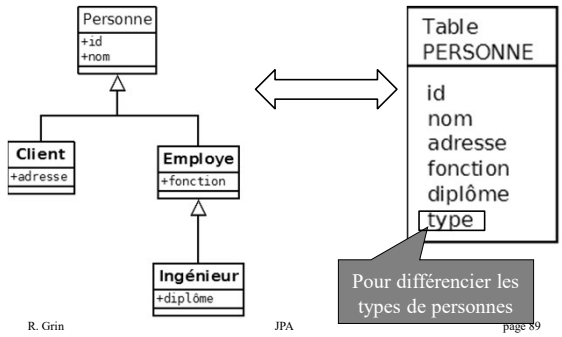
87

Stratégies

- 2 stratégies pour traduire une hiérarchie d'héritage en relationnel :
  - une seule table (`SINGLE_TABLE`) ; par défaut
  - une table par classe ; les tables sont jointes pour reconstituer les données (`JOINED`)
- Stratégie « une table par classe non abstraite » est moins utilisée (`TABLE_PER_CLASS`)

88

Toutes les classes traduites par une seule table



89

Exemple « 1 table par hiérarchie »

```
@Entity
@Inheritance(strategy =
  InheritanceType.SINGLE_TABLE)
public abstract class Personne {...}
```

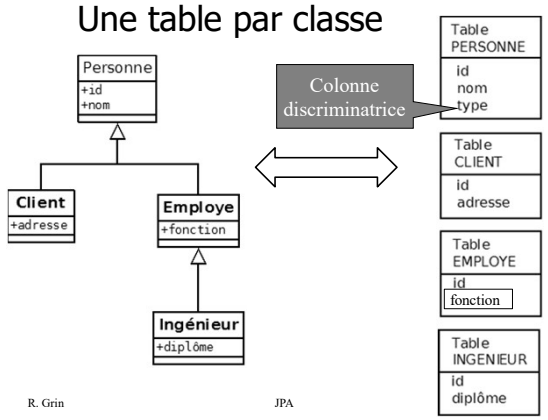
Dans la classe racine de la hiérarchie ; optionnel

```
@Entity
@DiscriminatorValue("E")
public class Employe extends Personne {
  ...
}
```

« `Employe` » par défaut

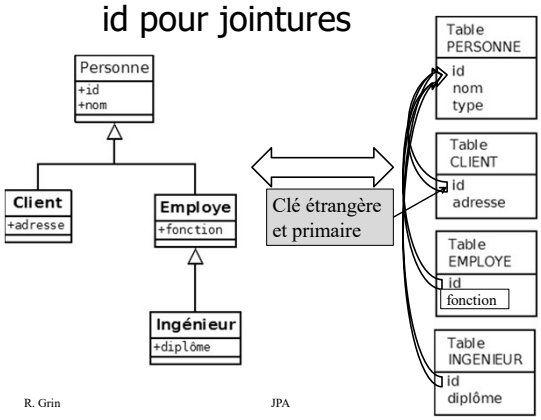
90

Une table par classe



91

id pour jointures



92

Exemple

```
@Entity
@Inheritance(strategy =
    InheritanceType.JOINED)
public abstract class Personne {...}
```

```
@Entity
@DiscriminatorValue("E")
public class Employe extends Personne {
  ...
}
```

« Employe » par défaut

93

```
@Entity
public abstract class Personne {...}
```

```
@Entity
public class Employe extends Personne {
  ...
}
```

```
@Entity
public class Ingenieur extends Employe {
  ...
}
```

Combien de tables relationnelles pour enregistrer les informations contenues dans ces entités ?

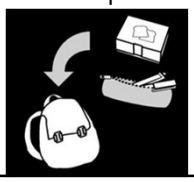
94

Requêtes – JPQL
Java Persistence Query Language

95

IMPORTANT

- Cette section concerne les entités retrouvées en interrogeant la base de données
- Rappel : ces entités sont automatiquement gérées par le gestionnaire d'entités
- Les modifications apportées à ces entités seront enregistrées au prochain commit



96

Chercher par identité

- `find` retrouve une entité en donnant son identificateur dans la BD :

```
Employe employe = em.find(Employe.class, 10);
```


97

- Il est possible de rechercher des données sur des critères plus complexes que la simple identité

98

Étapes pour récupérer des données

1. Décrire ce qui est recherché (langage JPQL)
2. Créer une instance de type `Query`
3. Initialiser la requête (paramètres, pagination)
4. Lancer l'exécution de la requête

```
String s = "select e from Employe as e";
Query query = em.createQuery(s);
List<Employe> liste = query.getResultList();
```


99

Exemple plus complexe

Qui veut expliquer ?

```
String q = "select e from Employe as e "
 + "where e.departement.numero = :numero";
Query query = em.createQuery(q);
query.setParameter("numero", 10);
query.setMaxResults(30);
for (int i = 0; i < 5; i++) {
  query.setFirstResult(30 * i);
  List<Employe> liste = query.getResultList();
  ... // Traitement « page » numéro i + 1
}
```


100

Langage JPQL

- *Java Persistence Query Language* décrit ce qui est recherché en utilisant le modèle objet (pas le modèle relationnel)
- Les seules classes qui peuvent être utilisées sont les entités et les Embeddable

101

Exemples de requêtes

Alias de classe obligatoire

- `select e from Employe e`
  ou bien
  `select e from Employe as e`
- `select e.nom, e.salaire from Employe e`
- `select e from Employe e`
  `where e.departement.nom = 'Direction'`
- `select d.nom, avg(e.salaire)`
  `from Departement d join d.employes e`
  `group by d.nom`
  `having count(d.nom) > 5`

Les champs doivent être préfixés par l'alias de classe : e.nom

102

Alias pour attributs

- « as » peut aussi être utilisé pour donner des alias aux champs du select :

```
SELECT AVG(e.salaire) AS s, e.adresse.ville
from Employe e
group by e.adresse.ville
order by s
```


103

Obtenir le résultat de la requête

- Résultat = une seule valeur ou entité : `Object getSingleResult()`
- Résultat = plusieurs valeurs ou entités : `List getResultList()`
- On peut aussi récupérer les données dans un `Stream` avec la méthode `getResultStream()`

  Attention, ne pas utiliser le stream pour faire des choses que le SGBD sait mieux faire, par exemple une sélection

104

Type d'un élément du résultat

- Le type d'un élément du résultat est
  - `Object` si la clause select ne comporte qu'une seule expression
  - `Object[]` sinon

105

Exemple

```
texte = "select e.nom, e.salaire "
      + " from Employe as e";
query = em.createQuery(texte);
List<Object[]> liste = query.getResultList();
for (Object[] info : liste) {
  System.out.println(info[0] + " gagne "
                     + info[1]);
}
```

Qui peut expliquer ?

106

Types de requête

- Requête dynamique : texte donné en paramètre de `createQuery`
- Requête nommée : texte statique donné dans une annotation d'une entité ; le nom est passé en paramètre de `createNamedQuery`
- Requête native (ou SQL) particulière à un SGBD : requête SQL (pas JPQL) avec tables et colonnes (pas classes et attributs) ; pour créer des tables ou des vues ou pour une requête non standard ; `createNativeQuery`

107

Exemple de requête nommée

```
@Entity
@NamedQuery (
  name = "employe.nomsEmployesDuDept",
  query = "select e.nom from Employe as e where
  upper(e.departement.nom) = :nomDept"
)
public class Employe extends Personne {
  ...
```

```
Query q = em.createNamedQuery(
  "employe.nomsEmployesDuDept");
```


108

Exemples de paramètres

```
Query query = em.createQuery(
  "select e from Employe as e "
  + "where e.nom = :nom");
query.setParameter("nom", "Dupond");
```

```
Query query = em.createQuery(
  "select e from Employe as e "
  + "where e.nom = ?1");
query.setParameter(1, "Dupond");
```


TypedQuery<T>

- Quand on connait le type `T` retourné par une requête, il vaut mieux utiliser l'interface `TypedQuery<T>` plutôt que `Query` car le compilateur peut faire plus de vérifications
- Par exemple, `getResultList()` retourne `List<T>` (type générique) et pas `List`

Exemples

```
String s = "select e from Employe as e";
TypedQuery<Employe> query =
    em.createQuery(s, Employe.class);
List<Employe> liste = query.getResultList();
```

```
TypedQuery<Compte> query =
  em.createNamedQuery("Compte.findAll",
                      Compte.class);
```


Clauses d'un select

- `select`
- `from`
- `where`
- `group by`
- `having`
- `order by` (suivi de `asc` ou `desc` ; `asc` par défaut)

JPQL permet les sous-requêtes et même les sous-requêtes synchronisées

Navigation

- Il est possible de suivre le chemin correspondant à une association avec la notation « pointée »
- Si e alias pour `Employe`,
  - « `e.departement` » désigne le département d'un employé ; l'entité `Employe` doit avoir un attribut `departement`
  - De même « `e.projets` » désigne les projets auxquels participe un employé

Règles pour les expressions de chemin

- Si une navigation ne donne qu'une seule entité (OneToOne ou ManyToOne) on peut chaîner une navigation à la suite

```
select e.nom,
     e.departement.nom,
     e.superieur.departement.nom
from Employe e
```

Contre-exemple

- « ~~e.projets.nom~~ » est interdit car `e.projets` est une collection
- Pour avoir les noms des employés du département « Direction » et les noms des projets auxquels ils participent, il faut une jointure :

```
select e.nom, p.nom
from Employe e
     join e.projets p
where e.departement.nom = 'Direction'
```


115

Jointure

- Interne (jointure standard **`join`**)
- Externe (`left outer join`)
- Avec récupération de données en mémoire (`join fetch`) ; pour éviter le problème des « N +1 selects »
- Jointure « à la SQL » : égalité de 2 valeurs, sans tenir compte des associations entre entités

116

Récupération entités associées

- Si on veut récupérer des entités qui sont associées à d'autres entités, le plus simple est la clause `join fetch` de JPQL
- La méthode `find` (`Map` passée en dernier paramètre), l'interface `Query` (méthode `setHint`) ou l'annotation `@NamedQuery` (attribut `hints`) permettent de donner des « hints » (indications, conseils)
- Par ces hints on peut désigner un graphe d'entités (pas étudié dans ce cours) qui indique quelles entités associées on veut récupérer

117

`join fetch` (1/2)

-
```
select e
from Employe as e
     join fetch e.participations
```
  Les participations sont créées dans la mémoire centrale en même temps que les employés (mais pas retournées par le select)
- L'instruction
  `unEmploye.getParticipations()`
  ne nécessitera donc pas d'accès à la base

118

`join fetch` (2/2)

- Occasionne souvent des doublons ; pour les éviter, ajouter `distinct` après le `select` (`select distinct e …`)
- Il faut éviter plusieurs `join fetch` dans un ordre JPQL, par exemple en décomposant en plusieurs ordres JPQL

119

Problème des N + 1 selects (1/2)

- `join fetch` permet de résoudre le problème des « N + 1 selects »
- Exemple : on veut les noms des projets auxquels participent les N employés
- On peut commencer par récupérer les employés avec un select JPQL (1 select SQL sera généré) :

```
select e from Employe e
```


120

Problème des N + 1 selects (2/2)

- Pour chaque employé, on récupère les projets de l'employé en lançant `e.getProjets()` qui déclenche un select SQL (N select SQL pour les N employés)
- N + 1 select SQL ont donc été exécutés, alors qu'un seul select SQL (avec jointure externe) aurait suffi à récupérer les informations
- La solution est de lancer :

```
select e from Employe e
        join fetch e.participations
```


Fonctions dans une requête

- Pour les chaînes de caractères : `concat`, `substring`, `trim`, `lower`, `upper`, `length`, `locate`
- Calculs : `abs`, `sqrt`, `mod`, `exp`, `ln`, `power`, `size`, `ceiling`, `floor`, `round`, `sign`
- Temporelles : `current_date`, `current_time`, `current_timestamp`, `local date`, `local datetime`, `local time`, `extract`
- Fonctions de regroupement : `count`, `max`, `min`, `avg` ; `count(distinct` *`expression`*`)`
- `case` (semblable CASE de SQL)
- Fonction du SGBD : `function('nomFonction')`

Exemples

- `select count(c) from Customer c`
-
```
select a.description,
  case type(a)
    when Stylo then 'stylo'
    when Ramette then 'ramette'
    else 'autre type'
  end
  from Article a
```


API « criteria »

- Tout ce qui peut être fait avec JPQL peut l'être avec cette API
- Les requêtes JPQL sont des `String` qui peuvent contenir des erreurs (par exemple le nom d'une classe qui n'existe pas)
- L'avantage de l'API « critères » est que les requêtes peuvent être vérifiées à la compilation
- L'inconvénient est que le code est un peu plus complexe à écrire, et moins lisible

Opération de modification en volume

Un cas d'utilisation

- Augmenter les 10 000 employés de 5%
- Une solution :
  1. Créer toutes les entités « Employe » en lisant les données dans la base
  2. Modifier le salaire de chaque entité
  3. Enregistrer ces modifications dans la base
- Mauvaise solution ! Pourquoi ?
- JPQL permet de modifier les données de la base directement, sans créer les entités correspondantes et avec une seule requête SQL

Exemple

```
String ordre =
    "update Employe e " +
    " set e.salaire = e.salaire * 1.05";
Query q = em.createQuery(ordre);
int nbEntitesModif = q.executeUpdate();
```


127

Concurrence

128

Gestion de la concurrence

- Par défaut, les problèmes d'accès concurrents sont gérés avec une stratégie optimiste :
  - on pense qu'aucune autre transaction ne modifiera les données sur lesquelles on va travailler
  - donc aucun blocage n'est effectué dans la base de données
  - au moment du commit, blocage court et vérifications pour voir si on avait raison d'être optimiste
  - si on avait tort, rollback de la transaction

129

@Version

- Pour savoir si une entité a été modifié entre 2 moments différents, un numéro de version est ajouté à l'entité et enregistré dans la base de données
- Exemple :

```
@Version
private int version;
```

- Ce numéro est incrémenté *automatiquement* par JPA à chaque modification
- Il ne doit jamais être modifié par l'application

130

Stratégie optimiste avec JPA

- JPA utilise le numéro de version pour déterminer si une entité a été modifiée par une autre transaction
- Si c'est le cas, une `OptimisticLockException` est levée, et la transaction est marquée pour un rollback, ce qui cause un rollback à la fin de la transaction
- `OptimisticLockException` est une exception runtime

131

lock(A, mode)

- Protège une entité contre les accès concurrents pour les cas où la protection offerte par défaut ne suffit pas
- Permet en particulier d'effectuer un blocage pessimiste (blocage dans la base des données sur lesquelles on travaille)

132

Variantes avec `lock`

- Les méthodes `find` et `refresh` de `EntityManager` ont une variante qui permet de bloquer les données liées aux entités retrouvées ou rafraichies
- L'interface `Query` a la méthode `setLockMode` pour bloquer les données récupérées par la requête
- `@NamedQuery` a un attribut `lockMode`

Comparaison optimiste - pessimiste

- Un traitement pessimiste peut conduire à bloquer parfois inutilement les autres utilisateurs et peut conduire à des interblocages
- Si les conflits ne sont pas trop fréquents le traitement optimiste est recommandé
- Sinon un traitement optimiste peut conduire à trop d'annulations de transactions et alors il vaut mieux faire un traitement pessimiste

Configuration des tables SQL – Adaptation du code Java à une base préexistante

Position du problème

- Parfois les structures des bases préexistantes ne respectent pas les bons usages ou ne correspondent pas aux valeurs par défaut supposées par JPA, pour des raisons diverses
- C'est souvent le cas pour ce qui concerne la modélisation des associations entre entités
- JPA peut s'adapter à presque toutes les situations préexistantes, le plus souvent en utilisant des annotations

Annotations

- `@Column` pour changer le nom
- `@Table` pour changer le nom d'une table
- `@JoinColumn` permet de changer le nom d'une colonne clé étrangère
- On peut même répartir les données d'une entité sur plusieurs tables avec `@SecondaryTable`

Définition des colonnes

- Si les tables relationnelles sont créées à partir du code Java des entités, la notation `@Column` permet de donner une définition précise du type SQL utilisé pour un attribut d'une entité

Exemple

- Le type Java `BigDecimal` correspond par défaut au type SQL DECIMAL mais sans chiffres après la virgule
- `@Column` permet d'indiquer la définition que l'on veut :

```
@Column(columnDefinition = "DECIMAL(12,2)")
private BigDecimal prix;
```


Bibliographie

- Pro Jakarta Persistence in Jakarta EE 10 de Lukas Jungmann, Mike Keith, Merrick Schincariol, Massimo Nardone, édtions Apress (4ème édition)