JPA 2.0
(*Java Persistence API*)

Université Française d'Egypte
Version 2.4 – 11/10/12
Richard Grin

Plan de cette partie

- Présentation de JPA
- Entités persistantes
- Gestionnaire de persistance
- Compléments sur les entités : identité, associations, héritage
- Langage d'interrogation
- Exceptions
- Modifications en volume
- Transaction
- Concurrence

Présentation de JPA

Java EE

- Java Enterprise Edition : plate-forme de développement pour les applications d'entreprises
- S'appuie sur la notion de serveur d'application, logiciel qui fournit des services aux applications (sécurité, transaction,…)

JPA

- Partie de Java EE
- API ORM (*Object-Relational Mapping*) standard pour la persistance des objets Java dans une BD relationnelle
- Peut aussi être utilisé par une application autonome, en dehors de tout serveur d'application

Fournisseur de persistance

- L'utilisation de JPA nécessite un fournisseur de persistance qui implémente les classes et méthodes de l'API
- *EclipseLink* est le fournisseur de persistance de l'implémentation de référence de JPA

Avertissement

- Ce cours étudie l'utilisation de JPA par une application autonome, en dehors de tout serveur d'application

Entité

- Classe dont les instances peuvent être persistantes
- Marquée par l'annotation **`@Entity`**
- « entité » pourra aussi désigner une instance de classe entité

Exemple d'entité – identificateur et attributs

```
@Entity
public class Departement {
  @Id
  @GeneratedValue
  private int id;
  private String nom;
  private String lieu;
}
```

Clé primaire dans la base (obligatoire)

générée automatiquement

Exemple d'entité – constructeurs

```
public Departement() { }

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

Constructeur sans paramètres obligatoire

Exemple d'entité – une association

L'association inverse dans la classe **`Employe`**

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

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


Fichiers de configuration XML

- **`META-INF/persistence.xml`** situé dans le *classpath* contient les informations pour obtenir une connexion à la BD
- Les annotations peuvent être remplacées par des fichiers de configuration XML
- Les fichiers XML l'emportent sur les annotations

persistence.xml

```
<persistence
  xmlns="http://java.sun.com/xml/ns/persistence"
  version= "1.0" >
  <persistence-unit name="employes"
      transaction-type="RESOURCE_LOCAL">
    . . .
  </persistence-unit>
</persistence>
```

Détails de la définition de l'unité de persistance étudiés plus tard

Gestionnaire d'entités (GE)

- Classe **`javax.persistence.EntityManager`**
- Interlocuteur principal pour le développeur
- Fournit les méthodes pour gérer les entités :
  - les rendre persistantes
  - les supprimer de la BD
  - retrouver leurs valeurs dans la BD
  - etc.

Contexte de persistance

- La méthode **`persist(objet)`** rend persistant ***`objet`*** qui sera alors géré par le GE
- Toute modification apportée à ***`objet`*** sera enregistrée dans la BD par le GE au moment du *commit*
- L'ensemble des entités gérées par un GE s'appelle un contexte de persistance (CP)
- Dans un CP il ne peut exister 2 entités différentes qui représentent des données identiques dans la BD

Exemple

```
EntityManagerFactory emf = Persistence.
    createEntityManagerFactory("employes");
EntityManager em =
  emf.createEntityManager();
EntityTransaction tx = em.getTransaction();
tx.begin();
Dept dept = new Dept("Direction", "Nice");
em.persist(dept);
dept.setLieu("Paris");
tx.commit();
```

enregistré dans la BD

Exemple (suite)

```
String queryString =
  "SELECT e FROM Employe e "
  + " WHERE e.poste = :poste";
Query q = em.createQuery(queryString);
q.setParameter("poste", "INGENIEUR");
List<Employe> liste = q.getResultList();
for (Employe e : liste) {
  System.out.println(e.getNom());
}
em.close();
emf.close();
```

Les employés récupérés sont ajoutés au contexte de persistance

Que va-t-il se passer si on en modifie un ?

Exemple (suite)

```
String queryString =
  "SELECT e FROM Employe e "
  + " WHERE e.poste = :poste";
Query q = em.createQuery(queryString);
q.setParameter("poste", "INGENIEUR");
List<Employe> liste = q.getResultList();
for (Employe e : liste) {
  System.out.println(e.getNom());
}
em.close();
emf.close();
```

Fermer dans un … ?

Entités

Conditions pour une classe entité

- Doit avoir un constructeur sans paramètre **`protected`** ou **`public`**
- Doit posséder un attribut qui représente la clé primaire dans la BD
- Ne doit pas être **`final`**
- Aucune méthode ou champ persistant ne doit être **`final`**

Méthodes **`equals`** et **`hashcode`**

- Dans certaines circonstances, ces 2 méthodes sont indispensables pour le bon fonctionnement des applications
- Il est donc recommandé de les redéfinir pour toutes les entités, en utilisant la propriété annotée par **`@Id`** comme critère d'égalité

Convention de nommage *JavaBean*

- Un JavaBean possède des propriétés
- Une propriété nommée « prop » est représentée par 2 méthodes pour lire et écrire sa valeur :
  - **`getProp()`** (ou **`isProp()`** si la propriété est de type **`boolean`**)
  - **`setProp(TypeDeProp prop)`**

Types d'accès à une valeur persistante

- L'état persistant d'un objet est constitué par les valeurs de ses attributs
- Le fournisseur de persistance peut accéder à la valeur d'un attribut de 2 façons
  - soit en accédant directement à la variable d'instance ; c'est l'accès « par champ »
  - soit en passant par les accesseurs (*getter* ou *setter*) ; c'est l'accès « par propriété »
- Si l'accès se fait par propriété, les *getter* et *setter* doivent être **`protected`** ou **`public`**

Type d'accès par défaut

- Le type d'accès par défaut est défini par la position des annotations dans la classe entité
- Il doit être le même pour toute une hiérarchie d'héritage
- JPA 2.0 permet de changer le type d'accès pour une classe ou un attribut en ajoutant une annotation : **`@Access(PROPERTY)`** ou **`@Access(FIELD)`**

Exemple

- Accès par champ :

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

- Accès par propriété :

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


Quel type d'accès choisir ?

- L'accès par propriété oblige à ajouter des getters et des setters pour tous les attributs
- Il est conseillé de choisir plutôt l'accès par champ

Attributs persistants

- Par défaut, tous les attributs d'une entité sont persistants
- Exception : attribut avec un champ **`transient`** ou annoté par **`@Transient`**

Types persistants

- Un attribut peut être d'un des types suivants :
  - type « basic »
  - type « Embeddable »
  - collection d'éléments de type « basic » ou Embeddable (nouveauté de JPA 2)
- Pour représenter une association entre entités :
  - entité
  - collection d'entités

Types « basic »

- Types primitifs (**`int`**, **`double`**,…)
- **`String`**, **`BigInteger`**, **`BigDecimal`**, enveloppes de type primitifs (**`Integer`**,…)
- **`Date`**, **`Calendar`**, **`Time`**, **`Timestamp`**
- Enumérations
- Tableaux de **`byte`**, **`Byte`**, **`char`**, **`Character`**
- Plus généralement **`Serializable`** (sauvegardé comme un tout dans la base)

Cycle de vie d'une instance d'entité

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

Les tables 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`**

Configuration « par exception »

- Mapping « JPA » par défaut pour les classes entités
- Ajouter des annotations seulement si les valeurs par défaut ne conviennent pas
- Par exemple, **`@Entity`** suppose que la table a le même nom que la classe
- Si on veut lui donner un autre nom on ajoute : **`@Table(name = "autre_nom")`**

Classe *Embeddable*

- Classe persistante dont les données
  - n'ont pas d'identité dans la BD
  - sont insérées dans une table associée à une entité

Exemple

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

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

optionnel

Gestionnaire d'entités (*Entity Manager*), GE

Principe de base

- La persistance des entités n'est pas transparente
- Une instance d'entité ne devient persistante que lorsque l'application appelle la méthode appropriée du gestionnaire d'entités (**`persist`** ou **`merge`**)

Unité de persistance

- Configuration nommée qui contient les informations nécessaires à l'utilisation d'une base de données
- Définie dans un fichier **`META-INF/persistence.xml`** situé dans le *classpath*

`persistence.xml`

```
<persistence
  xmlns="http://java.sun.com/xml/ns/persistence"
  version= "1.0" >
  <persistence-unit name="Employes"
      transaction-type="RESOURCE_LOCAL">
    <class>p1.Employe</class>
    <class>p1.Dept</class>
    <properties>
    . . . <!– Voir transparent suivant-->
    </properties>
  </persistence-unit>
</persistence>
```

Classes associées à l'unité de persistance

Exemple de section `properties`

```
<properties>
  <property
     name="javax.persistence.jdbc.driver"
     value="oracle.jdbc.OracleDriver"/>
  <property
     name="javax.persistence.jdbc.url"
     value="jdbc:oracle:thin:@...:INFO "/>
  <property
     name="javax.persistence.jdbc.user"
     value="toto"/>
  <property
     name="javax.persistence.jdbc.password"
     value="mdp"/>
</properties>
```


Cycle de vie d'un GE

- *En dehors d'un serveur d'application*, c'est l'application qui décide de la durée de vie d'un GE
- Création d'un GE :

```
EntityManagerFactory emf =
  Persistence.
  createEntityManagerFactory("employes");
EntityManager em =
  emf.createEntityManager();
```

- Fermeture du GE (plus possible de l'utiliser ensuite) :

```
em.close();
```


Méthodes de `EntityManager`

- **`void persist(Object entité)`**
- **`<T> T merge(T entité)`**
- **`void remove(Object entité)`**
- **`<T> T find(Class<T> classeEntité, Object cléPrimaire)`**
- **`void flush()`**
- **`void setFlushMode(FlushModeType flushMode)`**

Méthodes de `EntityManager`

- **`void lock(Object entité, LockModeType lockMode)`**
- **`void refresh(Object entité)`**
- **`void clear()`**
- **`boolean contains(Object entité)`**
- **`void close()`**
- **`boolean isOpen()`**
- **`EntityTransaction getTransaction()`**

Méthodes de **`EntityManager`**

- **`Query createQuery(String `*`requête`*`)`**
- **`Query createNamedQuery(String `*`nom`*`)`**
- **`Query createNativeQuery(String `*`requête`*`)`**
- **`Query createNativeQuery(String `*`requête`*`, Class `*`classeRésultat`*`)`**

`flush`

- Enregistre dans la BD les modifications effectuées sur les entités d'un contexte de persistance
- Le GE lance les commandes SQL adaptées pour modifier la BD (INSERT, UPDATE ou DELETE)
- Automatiquement effectué à chaque commit

`persist(entité)`

- **`entité`** devient une entité gérée
- L'état de **`entité`** sera sauvegardé dans la BD au prochain *flush* ou *commit*
- Aucune autre instruction ne sera nécessaire pour que les modifications effectuées ensuite sur **`entité`** soient enregistrées au moment du commit

`refresh(entité)`

- Les données de la BD sont copiées dans l'entité
- Peut être utile pour les transactions longues

Contexte de persistance - cache

- Joue le rôle de cache pour les accès à la BD
- Un **`find`** ou un **`query`** ramène les données du cache
- Bénéfice : meilleures performances
- Attention : les données du cache ne tiennent pas compte des modifications effectuées sans passer par le GE
- Peut poser des problèmes ; un **`refresh`** peut alors être la solution

Entité détachée

- Une entité gérée par un GE peut être détachée de son contexte de persistance (par exemple si le GE est fermé ou l'entité envoyée « au loin »)
- Cette entité détachée peut être modifiée
- Pour que ces modifications soient enregistrées dans la BD, il est nécessaire de rattacher l'entité à un GE par la méthode **`merge`**

`merge(a)`

- Attention, la méthode **`merge`** n'attache pas *a*
- Elle retourne une entité gérée qui a la même identité dans la BD que *a*, mais ça n'est pas le même objet (sauf si l'objet était déjà géré)
- Après **`merge(a)`**, l'application devra donc ne plus utiliser l'objet référencé par **`a`** ; il faut donc écrire ce type de code :

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


merge – une erreur à ne pas faire

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

Quel est le problème ?

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

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


Identité des entités

Clé primaire

- Une entité doit avoir un attribut, annoté **`@Id`**, qui correspond à la clé primaire dans la table associée
- La valeur de cet attribut ne doit jamais être modifiée 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.util.Date`**
- **`java.sql.Date`**

Génération automatique de clé

- Pour les clés de type nombre entier
- L'annotation **`@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

Type de génération

- Énumération **`GenerationType`** ; les valeurs :
- **`AUTO`** : choisi par le fournisseur de persistance, selon le SGBD (séquence, table,…) ; valeur par défaut
- **`SEQUENCE`** : séquence
- **`IDENTITY`** : colonne de type IDENTITY
- **`TABLE`** : table qui contient la prochaine valeur de l'identificateur

Exemple

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


Clé composite

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

Classe pour la clé composite

- Dans les 2 cas la clé composite doit être représentée par une classe à part
- Cette classe doit
  - être **`public`**
  - posséder un constructeur sans paramètre
  - être sérialisable
  - redéfinir **`equals`** et **`hashcode`**

Exemple avec `@EmbeddedId`

```
@Entity
public class Employe {
  @EmbeddedId
  private EmployePK employePK;
  ...
}
```

```
@Embeddable
public class EmployePK {
  private String nom;
  private Date dateNaissance;
  ...
}
```


Associations

Association bidirectionnelle

- Le développeur est responsable de la gestion correcte des 2 bouts de l'association
- Par exemple, si un employé change de département, les collections des employés des départements concernés doivent être modifiées

Méthode de gestion de l'association

- Pour faciliter la gestion des 2 bouts, il est commode d'ajouter une méthode qui effectue tout le travail
- En particulier, dans les associations 1-N, 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);
  employe.setDept(this);
}
```

Dans quelle classe est ce code ?

Bout propriétaire

- Un des 2 bouts est dit propriétaire de l'association ; celui qui contient la clé étrangère
- Pour les associations 1 – N, c'est le bout … ?
- N
- L'autre bout contient l'attribut **mappedBy** qui donne le nom de l'attribut dans le bout propriétaire

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 ? — Classe **Employe**

Dans quelle table sera la clé étrangère ? — Table **EMPLOYE**

Nom de la colonne clé étrangère : DEPARTEMENT_ID

Valeurs par défaut

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

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


Association M:N

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

Exemple

Dans quelle classe ?

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

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

Quel est le bout propriétaire ?

Dans quelle classe ?

Dans quelle table sera la clé étrangère ?

`@JoinTable`

- Si les valeurs par défaut ne conviennent pas, des informations sur la table association peuvent être données dans le bout propriétaire de l'association (annotation **`@JoinTable`**)

Exemple (classe **`Employe`**)

```
@ManyToMany
@JoinTable(
  name = "EMP_PROJET",
  joinColumns = @JoinColumn(name="matr"),
  inverseJoinColumns=
    @JoinColumn(name="code_Projet")
)
private Collection<Projet> projets;
```


Exemple (classe **`Employe`**)

```
@ManyToMany
@JoinTable(
  name = "EMP_PROJET",
  joinColumns = @JoinColumn(name="matr"),
  inverseJoinColumns=
    @JoinColumn(name="code_Projet")
)
private Collection<Projet> projets;
```

Nom de la table association

Exemple (classe **`Employe`**)

```
@ManyToMany
@JoinTable(
  name = "EMP_PROJET",
  joinColumns = @JoinColumn(name="matr"),
  inverseJoinColumns=
    @JoinColumn(name="code_Projet")
)
private Collection<Projet> projets;
```

Colonne de la table association qui référence le bout propriétaire

Exemple (classe **`Employe`**)

```
@ManyToMany
@JoinTable(
  name = "EMP_PROJET",
  joinColumns = @JoinColumn(name="matr"),
  inverseJoinColumns=
    @JoinColumn(name="code_Projet")
)
private Collection<Projet> projets;
```

Colonne de la table association qui référence l'autre bout

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

Classe association

- Association M:N traduite par une classe association
- Chaque classe qui participe à l'association peut avoir une collection d'objets de la classe association (suivant directionnalité)
- 2 possibilités pour la classe association :
  - un attribut identificateur (**`@Id`**) unique (le plus simple)
  - un attribut identificateur par classe participant à l'association

Code classe association (début)

```
@Entity
public class Participation {
  @Id @GeneratedValue
  private int id;
  @ManyToOne
  private Employe employe;
  @ManyToOne
  private Projet projet;
  private String fonction;
```


Code classe association (fin)

```
public Participation() { }

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


Persistance des objets associés

- Lorsqu'un objet **`obj`** est rendu persistant, il est naturel que les objets référencés par **`obj`** deviennent eux-aussi persistants
- Sinon, une partie de l'état de l'objet ne serait pas persistante
- Ce concept s'appelle la « persistance par transitivité » (*reachability*)
- Si c'est fait automatiquement, quand un objet A est rendu persistant, le service de persistance doit parcourir tous les objets référencés par A, et ceci récursivement

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

- Soit une classe **`Departement`** qui contient une collection d'**`Employe`**
- Si un département est rendu persistant et si un des employés du département est non persistant, une **`IllegalStateException`** est lancée au moment du commit
- L'attribut **`cascade`** des associations peut simplifier le code

Cascade

- Les opérations **`persist`**, **`remove`**, **`refresh`**, **`merge`** 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 doivent elles aussi être 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`** : les données associées ne sont récupérées que lorsque c'est vraiment nécessaire

Type 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 ?

Indiquer le type de récupération des entités associées

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

```
@OneToMany(mappedBy = "departement",
           fetch = FetchType.EAGER)
private Collection<Employe> employes;
```

Héritage

Stratégies

- 2 stratégies pour la traduction de l'héritage :
  - une seule table pour une hiérarchie d'héritage (**`SINGLE_TABLE`**) ; par défaut
  - une table par classe ; les tables sont jointes pour reconstituer les données (**`JOINED`**)
- La stratégie « une table distincte par classe concrète » est optionnelle (**`TABLE_PER_CLASS`**)

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

Table

- La table a le nom de la table associée à la classe racine de la hiérarchie d'héritage
- Elle doit comporter une colonne discriminatrice

Une table par classe

- La table qui correspond à la classe racine de la hiérarchie d'héritage doit comporter une colonne discriminatrice

Exemple

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

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

Classe mère persistante

- Une entité peut avoir une classe mère dont l'état est persistant, sans que cette classe mère ne soit une entité
- L'état de la classe mère sera enregistré dans la même table que les entités filles
- En ce cas, la classe mère doit être annotée par **`@MappedSuperclass`**
- Une classe entité peut aussi hériter d'une classe mère dont l'état n'est pas persistant

Requêtes – JPQL
Java Persistence Query Language

IMPORTANT

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

Chercher par identité

- **`find`** (de **`EntityManager`**) retrouve une entité en donnant son identificateur dans la BD
- 2 paramètres de type
  - **`Class<T>`** pour le type de l'entité recherchée
  - **`Object`** pour la clé primaire
- Exemple :

```
Departement dept =
  em.find(Departement.class, 10);
```


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

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

Exemple

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


Exemple plus complexe

```
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> listeEmployes =
    (List<Employe>)query.getResultList();
  ... // Affichage page numéro i + 1
}
```


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

Exemples de requêtes

Alias de classe obligatoire

- `select e from Employe 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
```


Obtenir le résultat de la requête

- Résultat = une seule valeur ou entité : **`Object getSingleResult()`**
- Résultat = plusieurs valeurs ou entités : **`List getResultList()`**

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

Exemple

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


Interface **`Query`**

- Instance obtenue par les méthodes **`createQuery`**, **`createNativeQuery`** ou **`createNamedQuery`** de **`EntityManager`**

Types de requête

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

Exemple de requête nommée

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

```
Query q =
  em.createNamedQuery("findNomsEmpsDept");
```


Paramètre des requêtes

- Désigné par son numéro (***`?n`***) ou par son nom (***`:nom`***)
- Valeur donnée par une méthode **`setParameter`**

Exemples

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

Le meilleur choix ?

Clauses d'un select

- **select**
- **from**
- **where**
- **group by**
- **having**
- **order by**

Polymorphisme dans les requêtes

- Les requêtes sont polymorphes : un nom de classe dans la clause **from** désigne cette classe et toutes les sous-classes
- Exemple :

```
select count(a) from Article as a
```

- Depuis JPA 2 on peut restreindre à un type donné :

```
select a from Article a
where type(a) in (Stylo, Lot)
```


Clauses **where** et **having**

- Peuvent comporter :
  - **[NOT] LIKE**, **[NOT] BETWEEN**, **[NOT] IN**
  - **AND**, **OR**, **NOT**
  - **[NOT] EXISTS**
  - **ALL**, **SOME/ANY**
  - **IS [NOT] EMPTY**, **[NOT] MEMBER OF**

Pour les collections

Sous-requête

- **where** et **having** peuvent contenir des sous-requêtes (et peuvent être synchronisées)
- {**ALL** | **ANY** | **SOME**} (*sous-requête*) fonctionne comme dans SQL
- Exemple :

```
select e from Employe e
where e.salaire >= ALL (
  select e2.salaire from Employe e2
  where e2.departement =
        e.departement)
```


Navigation

- Il est possible de suivre le chemin correspondant à une association avec la notation « pointée »

Exemple

- e alias pour **Employe**,
  - « **e.departement** » désigne le département d'un employé
  - « **e.projets** » désigne les projets auxquels participe un employé
-

```
select e.nom
from Employe as e
where e.departement.nom = 'Qualité'
```

Règles pour les expressions de chemin

- Une navigation peut être chaînée à une navigation précédente si la navigation précédente ne donne qu'une seule entité (OneToOne ou ManyToOne)

Exemple

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


Contre-exemple

- « ~~`d.employes.nom`~~ » est interdit car **`d.employes`** est une collection
- Pour avoir les noms des employés d'un département, il faut utiliser une jointure :

```
select e.nom
from Departement d
     join d.employes e
where d.nom = 'Direction'
```


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 »

Exemples

- **`select e.nom, particips.projet.nom`**
  `from Employe e`
  `join e.participations particips`
- **`select e.nom, d.nom`**
  `from Employe e, Departement d`
  `where d = e.departement`

`join fetch`

- Évite le problème des « N + 1 selects »
-
```
select e
from Employe as e
     join fetch e.participations
```
- Les participations placées à droite de **`join fetch`** sont créées en même temps que les employés (mais pas renvoyées par le select)
- L'instruction **`employe.getParticipations()`** ne nécessitera pas d'accès à la base

Fonctions

- Pour les chaînes de caractères : **`concat`**, **`substring`**, **`trim`**, **`lower`**, **`upper`**, **`length`**, **`locate`**
- Arithmétique : **`abs`**, **`sqrt`**, **`mod`**, **`size`**
- Temporelles : **`current_date`**, **`current_time`**, **`current_timestamp`**
- Fonctions de regroupement : **`count`**, **`max`**, **`min`**, **`avg`**
- case (semblable CASE de SQL)

API « criteria »

- Introduite par JPA 2.0
- 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 sans doute moins lisible

Opération de modification en volume

Utilité

- Augmenter les 10000 employés de 5%
- Mauvaise solution :
  - Créer toutes les entités « Employe » en lisant les données dans la base
  - Modifier le salaire de chaque entité
  - Enregistrer ces modifications dans la base
- JPQL permet de modifier les données de la base directement, sans créer les entités correspondantes

Exemple

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


Exceptions

Exceptions de JPA

- JPA n'utilise que des exceptions non contrôlées
- Elles héritent de **`PersistenceException`**

Types d'exception

- **`EntityNotFoundException`**
- **`EntityExistsException`**
- **`TransactionRequiredException`**
- **`RollbackException`**
- **`OptimisticLockException`**
- **`PessimisticLockException`**
- **`* LockTimeoutException`**
- **`* NonUniqueResultException`**
- **`* NoResultException`**

* : ne marquent pas la transaction pour un rollback

Transaction

`EntityTransaction`

- En dehors d'un serveur d'applications, une application utilise l'interface **`EntityTransaction`** pour travailler avec une transaction (transaction locale)

`EntityTransaction`

```
public interface EntityTransaction {
  public void begin();
  public void commit();
  public void rollback();
  public void setRollbackOnly();
  public boolean getRollbackOnly();
  public boolean isActive();
}
```


Exemple

```
EntityManager em;
...
try {
  em.getTransaction().begin()
  ...
  em.getTransaction().commit();
}
finally {
  em.close();
}
```

Modifications et commit

- Les modifications effectuées sur les entités gérées sont enregistrées dans la base au moment d'un *commit*
- Les modifications sont enregistrées dans la base, même si elles ont été effectuées avant le début de la transaction (avant le **`tx.begin()`**)

Rollback (1)

- En cas de *rollback*,
  - *rollback* dans la base de données
  - le contexte de persistance est vidé ; toutes les entités deviennent détachées

Rollback (2)

- Les instances d'entités Java gardent les valeurs qu'elles avaient au moment du *rollback*
- Mais ces valeurs sont le plus souvent fausses
- Le plus souvent il faut donc relancer des requêtes pour récupérer des entités avec des valeurs correctes

Concurrence

Exemple de problème de concurrence « BD »

- Une entité est récupérée depuis la BD (par un **`find`** par exemple)
- L'entité est ensuite modifiée par l'application puis la transaction est validée
- Si une autre transaction a modifié entre-temps les données de la BD correspondant à l'entité, il y aura un problème de mise à jour perdue

Gestion de la concurrence

- Par défaut, les problèmes d'accès concurrents sont gérés avec une stratégie optimiste :
  - T1 lit une entité
  - Une autre transaction T2 modifie alors les données de la BD correspondant à l'entité
  - T1 ne pourra valider une modification de l'entité

@Version

- Annote un attribut qui représente un numéro de version, incrémenté automatiquement par JPA à chaque modification
- Pour savoir si l'état d'une entité a été modifié entre 2 moments différents
- Correspond à une colonne de la base de données
- Ne doit jamais être modifié par l'application
- Requis pour la portabilité de la gestion de la concurrence par JPA

Exemple

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


lock(*A, mode*)

- Protège une entité contre les accès concurrents pour les cas où la protection offerte par défaut ne suffit pas

Modes de blocage optimistes

- Valeurs de l'énumération **LockModeType**
- **OPTIMISTIC** : si une autre transaction T2 a modifié l'entité A, rollback de la transaction T1

Quelle différence avec le mode de protection par défaut ?

Modes de blocage optimistes

- Valeurs de l'énumération **LockModeType**
- **OPTIMISTIC** : si une autre transaction T2 a modifié l'entité A, rollback de la transaction T1
- **OPTIMISTIC_FORCE_INCREMENT** : en plus, incrémenter le numéro de version de l'entité bloquée, même si elle n'a pas été modifiée
- Il est parfois intéressant de considérer que l'entité a été modifiée si des objets associés ont été modifiés

Blocage pessimistes

- Ils empêchent les autres transactions de modifier les données bloquées

Modes de blocage pessimistes (1)

- **`PESSIMISTIC_WRITE`** pour, en plus, empêcher les autres transactions de bloquer en mode PESSIMISTIC_READ ou PESSIMISTIC_WRITE (jusqu'à la fin de la transaction) ; correspond à un « select for update »
- **`PESSIMISTIC_READ`** pour empêcher les lectures non répétables de l'entité bloquée ; empêche les autres transactions de bloquer la même entité en mode PESSIMISTIC_WRITE

Modes de blocage pessimistes (2)

- **`PESSIMISTIC_FORCE_INCREMENT`** pour, en plus de PESSIMISTIC_WRITE, incrémenter le numéro de version de l'entité bloquée, même si elle n'a pas été modifiée

Concurrence et entité détachée

- Une entité est détachée de son GE
- Elle est modifiée
- Quand elle est rattachée à un GE, le GE vérifie au moment du commit que le numéro de version n'a pas changé
- Si le numéro a changé, une **`OptimisticLockException`** est levée