Accès en lecture aux Objets

Chargement des associations et Lazy-loading

JPQL / HQL

Criteria API

SQL Query

Lazy loading
Problématique

- Accéder au numéro isbn du livre correspondant à la 3ème location faite par l'adhérent d'id 1.

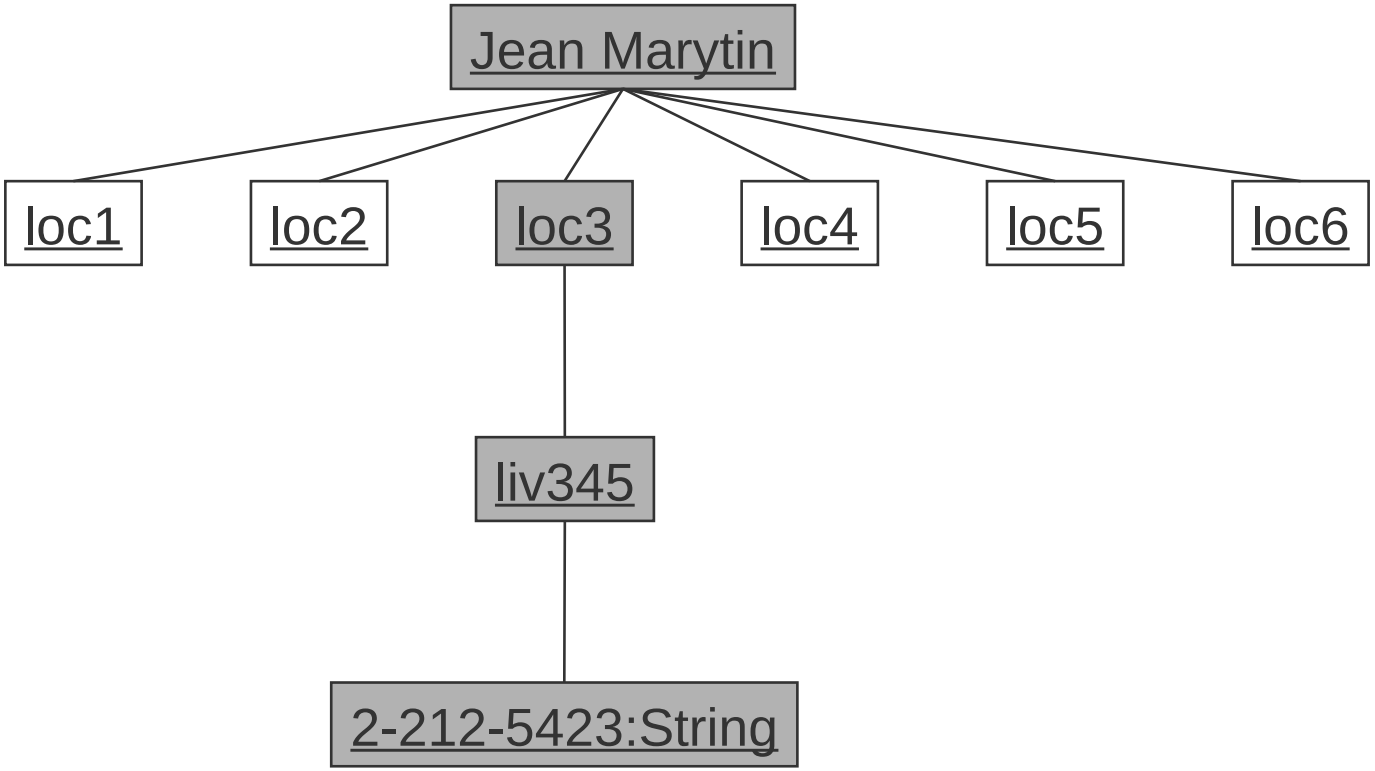
```
tx.begin();

Adherent jean = (Adherent)em.find(Adherent.class,new Long(1));
String isbn =
((Livre)jean.getLocations().get(2).getItem()).getIsbn();

tx.commit();
em.close()
```

Lazy loading
Problématique

Il n’est pas nécessaire de tout charger



Lazy loading
Solution

- Application du pattern Lazy Loading
  - Seul l'id de l'objet associé est chargé dans un premier temps
  - Chargement à la demande lors de l'usage des getter/setter..
- Paramétrage au niveau des fichiers de mapping
  - Stratégie de chargement par défaut
    - lazy / fetch
  - Peut être surchargé par du code (JPQL, autre)

Chargement des associations
Attributs *fetch* et *lazy*

- Pour bien appréhender la problématique de chargement des associations, Il faut distinguer 2 aspects orthogonaux :
  - **Quand** l'association est chargée : ***lazy***
  - **Comment** est elle chargée : ***fetchMode***
- Ces 2 aspects sont configurés via les annotations ou le fichier *hbm*
- Ils peuvent être surchargés par une requête JPQL/HQL, Criteria, Entity Graph

Chargement des associations
Fetch mode

- 3 stratégies distinctes de chargement :
  - ***Join*** : Récupère les entités associés en un seul SELECT via un OUTER JOIN
  - ***Select*** ou ***SubSelect***: Utilise un second select
  - ***Batch*** : Optimisation, plusieurs entités en 1 seul select via leurs id

Lazy Loading
JPA Attribut *fetch*

Le moment du chargement (*fetch type*) peut être précisé dans l'attribut *fetch* des annotations:

FetchType.LAZY ou ***FetchType.EAGER***

JPA ne permet pas de spécifier comment l'association est chargée

```
@OneToMany(cascade={CascadeType.ALL},
   fetch=FetchType.EAGER)

@Fetch(FetchMode.JOIN) // Hibernate specific

public Collection<Phone> getPhoneNumbers() {

return phoneNumbers;

}
```

Chargement des associations
Attribut lazy

- 6 scénarios pour le moment de chargement, dépendant de l'association :
  - ***Immediate*** (Eager) : L'association est chargée immédiatement
  - ***Lazy collection*** : La collection est chargée seulement lorsqu'elle est accédée.
    Mode par défaut des *@OneToMany*
  - ***Extra-lazy collection*** : Les éléments de la collection sont chargés individuellement lors de leur accès.
    Nécessite : *@LazyCollection(LazyCollectionOption.EXTRA)*
  - ***Proxy*** : L'association simple valeur est chargée lorsqu'une méthode get (!= getId) est appelée
  - ***No-proxy*** : L'association simple valeur est chargée dés que la variable d'instance est accédée
  - ***Lazy attribute*** : L'attribut ou l'association simple valeur est chargée lorsqu'il est accédée.
    Nécessite une instrumentation du code au build.

Lazy Loading
Valeurs par défaut

Par défaut, Hibernate utilise des associations *lazy* pour les collections (*@OneToMany* ou *@ManyToMany*) et pour les associations simples valeur

- Les autres associations (*@OneToOne* ou *@ManyToOne*) sont par défaut de type ***eager***

Avec les associations Lazy, les développeurs peuvent être confrontés à des *LazyInitializationException* <=> *Tentative d'accès à une association non chargée, à l'extérieur d'une session Hibernate*

Exemple *LazyInitializationException*

```
s = sessions.openSession();
Transaction tx = s.beginTransaction();
User u = (User) s.createQuery("from User u where
   u.name=:userName")
   .setString("userName", userName).uniqueResult();
// permissions n'est qu'un proxy
Map permissions = u.getPermissions();
tx.commit();
s.close() ;

// Error : LazyInitializationException
Integer accessLevel = (Integer) permissions.get("accounts");
```

LazyInitializationException

Pour contrer les *LazyInitializationException*, plusieurs options :

- Déplacer le code afin que l'accès s'effectue lorsque l'entité est attachée à un session ouverte
- Positionner l'attribut *lazy* à *false* => quelque soit le cas d'utilisation, l'association sera chargée : **mauvaise pratique !**
- Surcharger le comportement lazy pour le cas d'utilisation posant problème
  - Faire explicitement un appel au getter
  - Requête HQL
  - *Hibernate.initialize()*
  - Utiliser les profils de fetch

Lazy Loading
FetchType.EAGER

```
/* Les Phone sont systématiquement chargés
*  Aucun risque de LazyInitializationException
*/
@OneToMany(cascade={CascadeType.ALL},
   fetch=FetchType.EAGER)
@Fetch(FetchMode.JOIN) // Hibernate specific
public Collection<Phone> getPhoneNumbers() {
return phoneNumbers;
}
```

Exemple
Chargement explicite

```
s = sessions.openSession();
Transaction tx = s.beginTransaction();
User u = (User) s.createQuery("from User u where
   u.name=:userName")
.setString("userName", userName).uniqueResult();
Map permissions = u.getPermissions() ;
// Provoque le chargement de la collection
System.out.println(permissions.keySet());
tx.commit();
s.close() ;

Integer accessLevel = (Integer) permissions.get("accounts");
```

JOIN FETCH

Pour des raisons de performance, le fetch type par défaut (LAZY) est néanmoins préférable dans la plupart des cas

Les requêtes JPQL/HQL avec jointure ***JOIN FETCH*** permettent le chargement des objets d'une association comportant un *fetch* de type *LAZY*

- l'intérêt est d'éviter le fetch EAGER pour l'association, tout en permettant un chargement des objets en relation
- une requête JOIN FETCH est efficace car effectuée en une seule requête

Exemple *Join Fetch*

```
s = sessions.openSession();
Transaction tx = s.beginTransaction() ;
// Provoque le chargement de la collection
User u = (User) s.createQuery("from User u LEFT JOIN FETCH
   u.permissions where u.name=:userName")
.setString("userName", userName).uniqueResult();
Map permissions = u.getPermissions() ;

tx.commit();
s.close() ;

Integer accessLevel = (Integer) permissions.get("accounts");
```

Exemple *Hibernate.initialize*

```
s = sessions.openSession();
Transaction tx = s.beginTransaction();
User u = (User) s.createQuery("from User u where
   u.name=:userName")
.setString("userName", userName).uniqueResult();
Map permissions = u.getPermissions() ;
// Provoque le chargement de la collection
Hibernate.initialize(u.getPermissions());
tx.commit();
s.close() ;

Integer accessLevel = (Integer) permissions.get("accounts");
```

Entity Graph
Problématique

JPA 2.1 a introduit les ***Entity Graph*** qui permettent de contrôler finement le chargement des associations

L'idée est de pouvoir charger un graphe d'objet défini au runtime en une seule requête

- Peut améliorer les performances
- Permet de surcharger la configuration statique

Entity Graph
Définition

Pour définir un graphe d'entité, on peut utiliser l'annotation ***@NamedEntityGraph*** sur l'entité et préciser les associations que l'on veut charger par ***attributeNodes***

Il est également possible de faire référence à des sous-graphes, attribut ***subgraphs***

Entity Graph
Exemple, *attributeNodes*

```
@NamedEntityGraph(
  name = "post-entity-graph",
  attributeNodes = {
    @NamedAttributeNode("subject"),
    @NamedAttributeNode("user"),
    @NamedAttributeNode("comments"),
  }
)
@Entity
public class Post {
    private String subject;
    @OneToMany(mappedBy = "post")
    private List<Comment> comments = new ArrayList<>();
    @ManyToOne(fetch = FetchType.LAZY)
    private User user ;
```

Entity Graph
Exemple, *subgraphs*

```
@NamedEntityGraph(
  name = "post-entity-graph-with-comment-users",
  attributeNodes = {
    @NamedAttributeNode("subject"),
    @NamedAttributeNode("user"),
    @NamedAttributeNode(value = "comments", subgraph = "comments-subgraph"),
  },
  subgraphs = {
    @NamedSubgraph(
      name = "comments-subgraph",
      attributeNodes = {
        @NamedAttributeNode("user")
      }
    )
  }
)
@Entity
public class Post {
```

EntityGraph
Définition via l'API

Il est également possible de définir le graphe d'objets via son API :

```
EntityGraph<Post> entityGraph =
  entityManager.createEntityGraph(Post.class);

entityGraph.addAttributeNodes("subject");
entityGraph.addAttributeNodes("user");

entityGraph.addSubgraph("comments")
  .addAttributeNodes("user");
```

EntityGraph
Usage

```
EntityGraph entityGraph =
   entityManager.getEntityGraph("post-entity-graph");

Map<String, Object> properties = new HashMap<>();
properties.put("javax.persistence.fetchgraph", entityGraph);

Post post = entityManager.find(Post.class, id, properties);
```

FetchProfile (Spécifique Hibernate) Annotations

```
@Entity
@FetchProfile(name = "user-with-permissions", fetchOverrides = {
@FetchProfile.FetchOverride(entity = User.class, association =
   "permissions", mode = FetchMode.JOIN)
})
public class User {
@Id
@GeneratedValue
private long id;
private String name;
@OneToMany
private Map<String,Integer> permissions;
// standard getter/setter
...
}
```

Exemple
Activation d'un profil

```
em = factory.createEntityManager() ;
((Session)em).enableFetchProfile( "user-with-permissions" );
Transaction tx = s.beginTransaction();
User u = (User) s.createQuery("from User u where
   u.name=:userName")
.setString("userName", userName).uniqueResult();
Map permissions = u.getPermissions() ;
tx.commit();
s.close() ;

Integer accessLevel = (Integer) permissions.get("accounts");
```

Applications web
Pattern *Open Session in View*

Dans un environnement web, le pattern « **Open Session in View** » permet de s'affranchir des *LazyInitializationException*

Un filtre servlet est utilisé pour ne fermer la session *Hibernate* qu'à la fin de la requête

Les vues JSP/JSF ou même les sérialiseurs JSON peuvent accéder aux associations même si celles-ci n'ont pas été chargées dans le tiers métier

Accès en lecture aux Objets

Chargement des associations et Lazy-loading

JPQL / HQL

Criteria API

SQL Query

Techniques de récupération d'objet

- HQL/JPQL
  - Java Persistence/Hibernate Query Language
  - Langage de requête orienté objet
  - Puissant et simple
- API Criteria
  - Création de requête via l'API
- NativeQuery
  - Permet de jouer des requêtes SQL
  - Legacy system
  - Permet l'intervention du DBA

JPQL
Généralités

- Encapsulation SQL mais logique orientée objet.
- Clauses
  - select [**instance** d'une classe] from [des **classes**]
    where [des restrictions sur les attributs]
    order by [attributs] group by [attributs]
  - Usage des jointures
  - Usage de directive de chargement (fetch).
- Classe Query
  - *setParameter(), setHint()*
  - *setFirstResult(), setMaxResult()*
  - *getResultList(), getSingleResult()*

JPQL
from

- La clause ***from*** indique des entités et éventuellement des alias qui pourront être réutilisés dans la requête

```
// Forme la plus simple
from Adherent
// Utilisation d'un alias
from Adherent as adh
from Adherent adh
// Cross join (produit cartesien)
from Formula, Parameter
from formula as formula, Paramater as parameter
```

JPQL
Association et join

- ***join*** permet d'assigner des alias à des associations (nécessaire en général pour des associations **ToMany*)

```
// inner et left outer
from Cat as cat
inner join cat.mate as mate
left outer join cat.kittens as kitten
// Utilisation d'un alias
from Cat as cat left join cat.mate.kittens as kittens

// Cross join (produit cartesien)
from Formula form full join form.parameter param
```

Types de join

Les types de join supportés sont ceux de ANSI SQL :

- ***inner join*** : peut être abrégé en *join*
- ***left outer join*** : peut être abrégé en *left join*
- ***right outer join*** : peut être abrégé en *right join*
- ***full join***

Des conditions peuvent être ajoutées sur le *join* avec ***with***

```
from Cat as cat
left join cat.kittens as kitten
with kitten.bodyWeight > 10.0
```

fetch join

fetch permet d'initialiser les associations, il s'utilise avec *inner join* et *left outer join*

Il est souvent utiliser avec ***distinct*** pour éviter les doublons

Par défaut, on charge pas

```
@OneToMany
List<MotClef> motsClefs = new ArrayList()
```

```
select theme
   from Theme theme
        left join fetch theme.motclefs mot
```

Force le chargement avec un SQL outer join

join implicite

Le join implicite n'utilise pas le mot clé join mais la notation **.**

Il est équivalent à un *inner join*

```
// inner join
from Cat as cat where cat.mate.name like '%s%'
```

Clause select

La clause select spécifie les objets et les propriétés retournés par la requête

Il peut spécifier plusieurs objets/propiétés

```
// Objets en retour
select cat.mate from Cat cat
// Propriétés en retour
select cat.name from DomesticCat cat
// Object[] en retour
select mother, mate.name from DomesticCat as mother
inner join mother.mate as mate
// List en retour
select new list(mother, mate.name) from DomesticCat as
mother inner join mother.mate as mate
// Objets en retour
select new Family(mother, mate) from DomesticCat as mother
join mother.mate as mate
```

Fonctions d'agrégation

Les fonctions d'agrégation supportées sont
avg(...), sum(...), min(...), max(...), count(*), count(...), count(distinct ...), count(all...)

```
// Agrégation
select avg(cat.weight), sum(cat.weight), max(cat.weight),
count(cat) from Cat cat
// Agrégation et +
select cat.weight + sum(kitten.weight)
from Cat cat join cat.kittens kitten
group by cat.id, cat.weight
// Concaténation
select firstName||' '||initial||' '||upper(lastName)
from Person
```

Where clause

La clause *where* permet d'affiner les résultats en précisant des contraintes sur les objets ou leurs propriétés

```
// Restriction sur propriétés
from Cat as cat where cat.name='Fritz'

select foo from Foo foo, Bar bar
where foo.startDate = bar.date

from Cat cat where cat.mate.name is not null
// Restriction sur objet
select cat, mate from Cat cat, Cat mate
where cat.mate = mate
```

Where clause

La clause *where* peut utiliser :

Opérateurs math. : +, - *, /

Opérateurs de comparaison : =, >=, <=, <>, *!=, like*

Opérateurs logiques : *and, or, not*

Parenthèses *( )*

Opérateurs : *in, not in, between, is null, is not null, is empty, is not empty, member of* et *not member of*

Concaténation : ...||... ou concat(...,...)

String : *substring(), trim(), lower(), upper(),*

length(), locate(), abs(), sqrt(), bit_length(), mod()

Dates : *current_date(), current_time(), and current_timestamp()*

second(...), minute(...), hour(...), day(...), month(...), et *year(...)*

...

Exemples

```
from DomesticCat cat where cat.name not between 'A' and 'B'

from DomesticCat cat where cat.name in ('Foo', 'Bar', 'Baz')


from Cat cat where cat.kittens.size > 0

select mother from Cat as mother, Cat as kit
where kit in elements(foo.kittens)

from Order order where order.items[0].id = 1234
```

Clauses *order* et *group*

La liste peut être ordonné par rapport aux propriétés des entités retournés

Les valeurs d'aggrégation peuvent être groupées et une clause HAVING peut-être ajoutée

```
from DomesticCat cat
order by cat.name asc, cat.weight desc, cat.birthdate

select cat.color, sum(cat.weight), count(cat)
from Cat cat
group by cat.color
having cat.color in (eg.Color.TABBY, eg.Color.BLACK)
```

Sous-requêtes

Si la base de données supporte les sous-select, on peut utiliser des sous-requêtes Hibernate dans dans les clauses SELECT et WHERE

```
from Cat as fatcat where fatcat.weight >
(select avg(cat.weight) from DomesticCat cat)


select cat.id, (select max(kit.weight) from cat.kitten kit)
from Cat as cat
```

setParameter

Une requête HQL peut être paramétrée comme les *PreparedStatement* de JDBC

- cela est notamment plus efficace lorsque les requêtes sont répétées avec des valeurs différentes
- un objet *Query* est créé à partir d'une requête HQL comportant un ou plusieurs paramètres positionnels ou nommés

• les méthodes ***setParameter*** de *Query* permettent ensuite de donner une valeur à chacun des paramètres:

```
Query setParameter(int position,Object value);

Query setParameter(String name,Object value);
```

Paramètres

Les paramètres positionnels sont représentés par le caractère **?** suivi du numéro de l'argument

```
public List selectByVille(String ville){
Query query=manager.createQuery("SELECT c FROM Client AS c WHERE c.ville= ?1");
query.setParameter(1,ville);
return query.getResultList();
}
```

Les paramètres nommés sont représentés par le caractère **:** suivi du nom de l'argument

```
public List selectByVille(String ville){
Query query=manager.createQuery("SELECT c FROM Client c WHERE c.ville= :town");
query.setParameter("town",ville);
return query.getResultList();
}
```

Paramètres Date

Lorsque les paramètres sont des dates (de *type java.util.Date* ou *java.util.Calendar*), il faut utiliser l'une des méthodes suivantes:

```
public Query setParameter(String name,Date value,TemporalType temporalType);

public Query setParameter(String name,Calendar value, TemporalType
    temporalType);

public Query setParameter(int position,Date value, TemporalType temporalType);

public Query setParameter(int position,Calendar value, TemporalType
    temporalType);
```

Le type *javax.persistence.TemporalType* permet d'indiquer à la couche de persistance le type SQL correspondant:

```
public enum TemporalType{

DATE,// java.sql.Date

TIME,// java.sql.Time

TIMESTAMP // java.sql.Timestamp

}
```

Interface *Query*
org.hibernate ou *javax.persistence*

les requêtes de type select sont émises par les méthodes ***uniqueResult*** ou ***list*** (Hibernate) ***getSingleResult*** ou ***getResultList*** (JPA)

- les requêtes de type mise-à-jour sont émises par la méthode ***executeUpdate*** (Hibernate et JPA)

uniqueResult/getSingleResult

La méthode ***getSingleResult*** exécute la requête et renvoie un résultat sous la forme d'un unique objet

- une exception *NonUniqueResultException* est lancée si plus d'un résultat est trouvé
- une exception *EntityNotFoundException* est lancée si aucun résultat n'est trouvé

```
Query query=manager.createQuery(
  "SELECT c from Customer AS c where
  c.firstName='MARTIN'");

Customer
  customer=(Customer)query.getSingleResult();
```

list/getResultList

la méthode ***getResultList*** exécute la requête et renvoie un résultat sous la forme d'une collection

- la liste retournée est vide si aucun résultat n'est trouvé

```
Query query=manager.createQuery(
"SELECT c from Customer AS c where
c.firstName='MARTIN' ");
List customers=query.getResultList();
```

setMaxResults et *setFirstResults*

Les méthodes ***setMaxResults*** et ***setFirstResult*** permettent de paginer les résultats, principalement lorsque ces derniers sont nombreux

– ces deux méthodes renvoient un objet *Query* permettant d'enchaîner les appels

```
public List getCustomers(int max, int index){
Query query=manager.createQuery(
"SELECT c from Customer AS c");
return query.setMaxResults(max).
setFirstResult(index).getResultList();
}
```

@NamedQuery

Les requêtes nommées permettent d'associer un nom à une chaîne JPQL via l 'annotation ***@NamedQuery***

Le nom de la requête est ensuite passée en paramètre de la méthode ***createNamedQuery***

```
@NamedQuery (name="clientVille",query="SELECT c FROM Client AS
   c WHERE c.ville= ?1")
@Entity public class Client implements Serializable{
…
public List selectByVille(String ville){
  Query query= manager.createNamedQuery("clientVille");
  query.setParameter(1,ville);
  return query.getResultList();
}
```

@NamedQueries

Pour déclarer plusieurs requêtes nommées sur une entité, l'annotation ***@NamedQueries*** doit être utilisée:

```
@NamedQueries({
  @NamedQuery (name="clientVille",
     query= "SELECT c FROM Client AS c WHERE c.ville= ?1"),
  @NamedQuery (name="clientAll",
     query= "SELECT c FROM Client AS c")
})
@Entity public class Client implements Serializable{
```

Mise à jour

Les requêtes de mise à jour contiennent le mot clé ***UPDATE*** et s'exécute via ***executeUpdate()***

```
Query query = em.createQuery(
      "UPDATE Country SET population = population * 11 / 10 " +
      "WHERE population < :p");
  int updateCount = query.setParameter(p, 100000).executeUpdate();
```

Suppression

Les requêtes de supression contiennent le mot clé ***DELETE*** et s'exécute via ***executeUpdate()***

```
Query query = em.createQuery(
    "DELETE FROM Country c WHERE c.population < :p");
int deletedCount = query.setParameter(p, 100000).executeUpdate();
```

Accès en lecture aux Objets

Chargement des associations et Lazy-loading

JPQL / HQL

Criteria API

SQL Query

L'API Criteria
Généralités

- Moyen d'écriture programmatique de requêtes.

- Se crée à partir d'un ***CriteriaBuilder***
```
CriteriaQuery<Adherent> cb.createQuery(Adherent.class)
```

- On ajoute des restrictions à l'aide de la méthode ***where()*** et d'expressions créées avec ***CriteriaBuilder***
```
cr.select(root).where(cb.like(root.get("nom"), "Thib %"));
```

- Validé à la compilation ce qui n'est pas le cas du HQL.
- Depuis Hibernate 5.2, l'API Hibernate Criteria est obsolète et le nouveau développement se concentre sur l'API JPA Criteria.

L'API Criteria
Exemples (1/3)

- Sélectionnez tous les adhérents

```
Session s = DBHelper.getFactory().openSession();
Transaction tx = s.beginTransaction();
System.out.
    println("\n Test Requete : sélectionnez  tous les adhérents");
CriteriaBuilder cb = session.getCriteriaBuilder();
CriteriaQuery<Adherent> cr = cb.createQuery(Adherent.class);
Root<Adherent> root = cr.from(Adherent.class);
cr.select(root);

Query<Adherent> query = session.createQuery(cr);
List<Adherent> results = query.getResultList();
tx.commit();
s.close();
```

L'API Criteria
Exemples (2/3)

- Sélectionnez les adhérents dont le nom commence par « Val ».

```
Session s = DBHelper.getFactory().openSession();
Transaction tx = s.beginTransaction();
System.out
 .println("\n Test Requete : sélectionnez l'adhérent de nom :[Val%]");
CriteriaBuilder cb = session.getCriteriaBuilder();
CriteriaQuery<Adherent> cr = cb.createQuery(Adherent.class);
Root<Adherent> root = cr.from(Adherent.class);
cr.select(root).where(cb.like(root.get("name"), "Val%"));


Query<Adherent> query = session.createQuery(cr);
List<Adherent> results = query.getResultList();

tx.commit();
s.close();
```

L'API Criteria
Exemples (3/3)

- Chargez un adhérent avec un graphe d' entité

```
EntityManager em = DBHelper.getFactory().getEntityManager();
System.out
.println("\n Test Requete : sélectionnez l'adhérent de nom :[Marytin]
             et chargez aussi ses locations");
EntityGraph entityGraph = em.getEntityGraph("adherent-graph-with-locations");
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Adherent> cQuery = cb.createQuery(Adherent.class);
Root<Adherent> root = cQuery.from(Adherent.class);
cQuery.where(cb.equal(root.<Long>get("id"), id));
TypedQuery<Adherent> typedQuery = entityManager.createQuery(cQuery);
typedQuery.setHint("javax.persistence.loadgraph", entityGraph);
Adherent adherent = typedQuery.getSingleResult();

tx.commit();
s.close();
```

API Criteria
Mise à jour

Depuis JPA 2.1, l'API Criteria permet d'effectuer des mises à jour via l'objet ***CriteriaUpdate*** et sa méthode ***set()***

```
CriteriaUpdate<Item> criteriaUpdate =
   cb.createCriteriaUpdate(Item.class);
Root<Item> root = criteriaUpdate.from(Item.class);
criteriaUpdate.set("itemPrice", newPrice);
criteriaUpdate.where(cb.equal(root.get("itemPrice"), oldPrice));
Transaction transaction = session.beginTransaction();
session.createQuery(criteriaUpdate).executeUpdate();
transaction.commit();
```

API Criteria
Suppression

De la même façon, il est possible de faire des suppressions en lot avec ***CriteriaDelete***

```
CriteriaDelete<Item> criteriaDelete =
   cb.createCriteriaDelete(Item.class);
Root<Item> root = criteriaDelete.from(Item.class);
criteriaDelete.where(cb.greaterThan(root.get("itemPrice"),
   targetPrice));
Transaction transaction = session.beginTransaction();
session.createQuery(criteriaDelete).executeUpdate();
transaction.commit();
```

Accès en lecture aux Objets

Chargement des associations et Lazy-loading

JPQL / HQL

Criteria API

SQL Query

SQL Query
Généralités

- Utiliser des requêtes SQL natives
- Tirer parti de spécificités de la base
- Facilite l'intégration ou la reprise de code SQL.
- Nécessite un mapping des noms des tables vers les classes.
- N'utiliser qu'en dernier recours !

Native Query
createNativeQuery()

- La méthode ***createNativeQuery()*** crée une *javax.persistence.Query* classique sans parser la chaîne de caractère de la requête
- Dans l'exemple ci-dessous, le résultat est une liste d'objet car aucune information de mapping n'a été fournie

```
Query q = em.createNativeQuery("SELECT a.firstname, a.lastname FROM Author a");
List<Object[]> authors = q.getResultList();
```

Native Query
Mapping direct

- Il est possible de fournir des informations de mapping afin de récupérer directement des entités.
- Par exemple, en fournissant une classe et en récupérant l'intégralité des colonnes de la table

```
Query q = em.createNativeQuery(
"SELECT a.id, a.version, a.firstname, a.lastname FROM Author a"
, Author.class);
List<Author> authors = q.getResultList();
```

Native Query
@SqlResultSetMapping

- Le mapping peut être précisé dans une annotation ***@SqlResultSetMapping*** placée sur l'entité

```
@SqlResultSetMapping(
        name = "AuthorValueMapping",
        classes = @ConstructorResult(
                targetClass = AuthorValue.class,
                columns = {
                    @ColumnResult(name = "id", type = Long.class),
                    @ColumnResult(name = "firstname"),
                    @ColumnResult(name = "lastname"),
@ColumnResult(name = "numBooks", type = Long.class)}))
```

- Puis utilisée

```
Query q = em.createNativeQuery(
"SELECT a.id, a.firstname, a.lastname, count(b.id) as numBooks FROM Authora
JOIN BookAuthor ba on a.id = ba.authorid
JOIN Book b ON b.id = ba.bookid GROUP BY a.id"
, "AuthorValueMapping");
List<AuthorValue> authors = q.getResultList();
```


Atelier : Accès en lecture