Accès en lecture aux Objets

Plan

- Lazy loading et stratégie de fetch
- Les techniques d’accès
  - HQL
  - Criteria
  - SQL Query

Lazy loading
Problématique

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

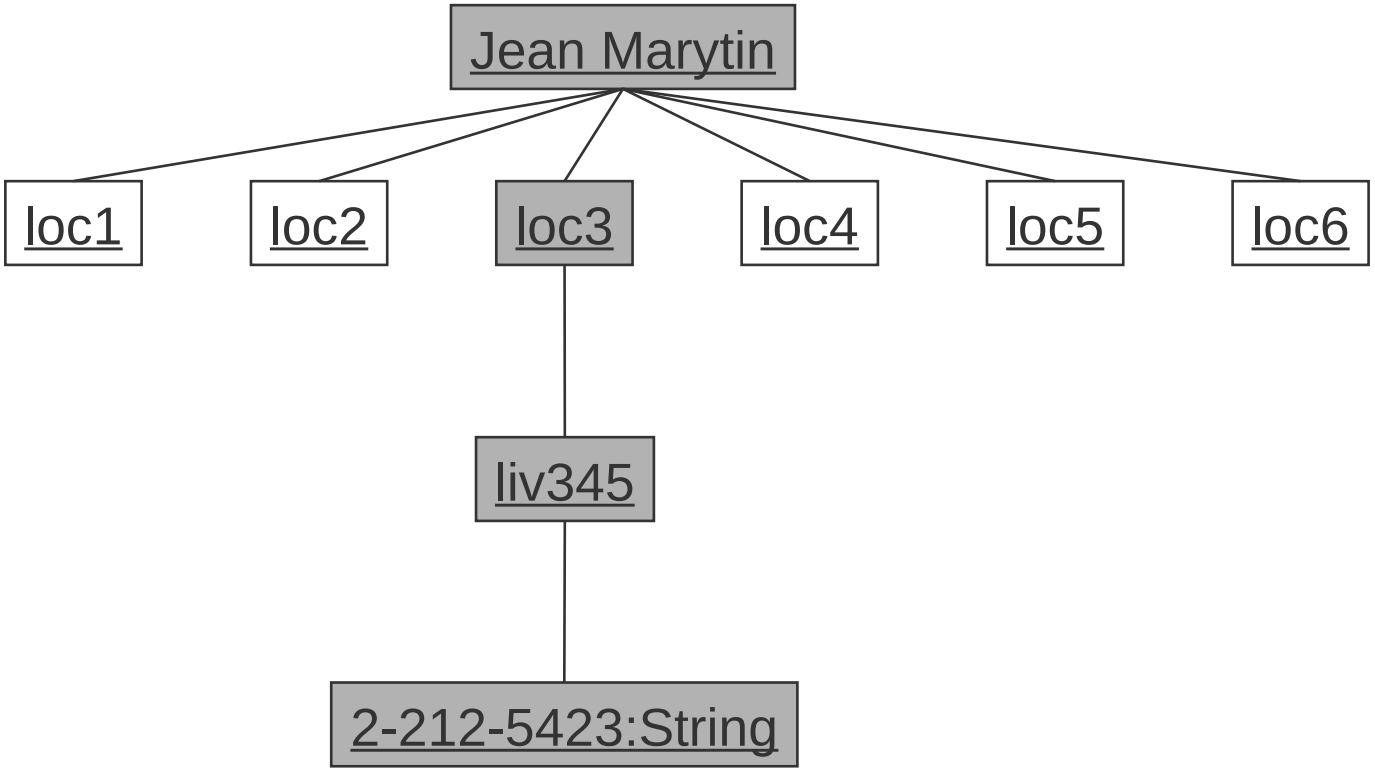
```
tx = session.beginSession();

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

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

Lazy loading
Problématique

Il n’est pas nécessaire de tout charger



Lazy loading
Solution

- Application du pattern Lazy Loading
  - Chargement à la demande lors de l'usage des getter/setter.
  - Seul l'id est chargé et stocké dans un proxy encapsulant l'instance persistante.
- Paramétrage
  - Au niveau des fichiers de mapping (stratégie de chargement par défaut)
    - lazy / fetch
  - Au niveau des requêtes de recherche

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 : ***fetch***
- Ces 2 aspects sont configurés via les annotations et le fichier *hbm*
- Ils peuvent être surchargés par une requête HQL ou Criteria

Chargement des associations
Attribut fetch

- 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

Chargement des associations
Attribut lazy

- 6 scénarios pour le moment de chargement :
  - ***Immediate*** (Eager) : L'association est chargée immédiatement
  - ***Lazy collection*** : La collection est chargée seulement lorsqu'elle est accèdée
  - ***Extra-lazy collection*** : Les éléments de la collection sont chargés individuellement lors de leur accès
  - ***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 association simple valeur

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

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();
Map permissions = u.getPermissions();
tx.commit();
s.close() ;

// Error!
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

Exemple hbm
lazy=false, fetch= "join"

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping SYSTEM
"C:\users\hve\Veille_technologique\cours_hibernate\
            hibernate-3.0\hibernate-mapping-3.0.dtd">

<hibernate-mapping  package="com.plb.etechno.j12.exemple.metier">
    <class name="Theme" table="TTheme">
           <id  name="id" column="id">
                <generator class="native"/>
           </id>

           <property name="label" column="label"/>

           <set name="motclefs" fetch="join" lazy="false">
               <key column="IDTheme"/>
               <one-to-many class="MotClef"/>
           </set>
    </class>
</hibernate-mapping>
```

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;

}
```

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 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");
```

FetchProfile 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
...
}
```

FetchProfile hbm

```
<hibernate-mapping>
<class name="User">
...
<set name="permissions">
<key column="user_id"/>
<one-to-many class="Permission"/>
</set>
</class>
<class name="Order">
...
</class>
<fetch-profile name="user-with-permissions">
<fetch entity="User" association="permissions" style="join"/>
</fetch-profile>
</hibernate-mapping>
```

Exemple
Activation d'un profil

```
s = sessions.openSession() ;
session.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

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 qu'à la fin de la requête

Les pages JSP, JSP peuvent accéder aux association même si celles-ci n'ont pas été chargées dans le tiers métier

Techniques de récupération d'objet

- HQL
  - Hibernate Query Language
  - Langage de requête orienté objet
  - Puissant et simple
- API Criteria
  - Permet la création de requête dynamique
- SQLQuery
  - Permet de jouer des requêtes SQL
  - Legacy system
  - Permet l'intervention du DBA

HQL
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
  - iterate(), list(), scroll()
  - setParameter()

HQL
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
```

HQL
Association et join

- ***join*** permet d'assigner des alias à des associations

```
// 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

Il ne peut pas être utilisé avec les requêtes utilisant *iterate()*

HQL
Problème du N + 1 Select

- Problème
  - Chargement de la liste des motclefs en lazy-loading
  - Parcours de la liste des thèmes
    - Pour chaque thème, chargement (Select) des motclefs.
  - Autant de Select que de thème + 1
- Solution
  - Charger les mot-clefs en une requête avec les thèmes.

HQL
Passer outre le lazy loading

- Charger les motclefs en même temps que les thèmes.

Par défaut, on charge pas

```
<set name=« motclefs » lazy=« true »>
   <key column=« idTheme » />
   <one-to-many class=« MotClef » />
</set>
```

```
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
```

HQL
Requêtes polymorphiques

- Requête polymorphe : renvoie tous les objets persistants en base

```
from java.lang.Object
```

- sélectionnez les items ayant comme mot clé [guerre]

```
select distinct item
   from Item item
        join item.motclefs mot
   where mot.mot = :motdonne
```

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(...,...)
- EJB-QL : *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
```

Order et group clause

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 EJB QL select et renvoie un résultat sous la forme d'un objet Query

- une exception *NonUniqueResultException* est lancée si plus 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 EJB QL select 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 EJB-QL nommées sur les beans entités sont crées au moyen de la méthode ***createNamedQuery*** de *EntityManager*

- Il s'agit tout d'abord d'associer un nom à une requête EJB-QL, au moyen d'une annotation ***@NamedQuery*** sur la classe d'un bean entité:

```
@NamedQuery (name="clientVille",query="SELECT c FROM
   Client AS c WHERE c.ville= ?1")

@Entity public class Client implements Serializable{
```

- Il s'agit ensuite de faire appel à cette requête en précisant le nom qui lui a été donné

```
public List selectByVille(String ville){

Query query= manager.createNamedQuery("clientVille");

query.setParameter(1,ville);

return query.getResultList();

}
```

@NamedQueries

Dans le cas où l'on souhaite déclarer plusieurs requêtes nommées sur un bean 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{
```

L'API Criteria
Généralités

- Moyen d'écriture programmatique de requêtes.
- Se crée à partir d'une session
  - session.createCriteria(Adherent.class)
- On lui ajoute les critère créés à l'aide de la classe utilitaire Restrictions
  - Restrictions.like(« nom », »Valron »);
- Validé à la compilation ce qui n'est pas le cas du HQL.

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");
Criteria crit = s.createCriteria(Adherent.class);
List<Adherent> results = (List<Adherent>)crit.list();
for(Adherent a : results ){
    System.out.println(a.getId() + ") " + a.getNom());
}
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%]");
Criteria crit = s.createCriteria(Adherent.class);
Criterion nameEq = Restrictions.like("nom","Val%");
crit.add(nameEq);
List<Adherent> results = (List<Adherent>)crit.list();
for(Adherent a : results ){
    System.out.println(a.getId() + ") " + a.getNom());
}
tx.commit();
s.close();
```

L'API Criteria
Exemples (3/3)

- Chargez les locations avec les adhérents

```
Session s = DBHelper.getFactory().openSession();
Transaction tx = s.beginTransaction();
System.out
.println("\n Test Requete : sélectionnez l'adhérent de nom :[Marytin]
            et chargez aussi ses locations");
Criteria crit = s.createCriteria(Adherent.class);
Criterion nameEq = Restrictions.like("nom","Marytin");
crit.setFetchMode("locations",FetchMode.JOIN);
crit.add(nameEq);
Set<Adherent> results = new HashSet<Adherent>((List<Adherent>)crit.list());
for(Adherent a : results ){
    System.out.println(a.getId() + ") " + a.getNom());
    for(Location l : a.getLocations()){
        System.out.println("   " + l.getId() + "  -> " +
                           l.getItem().getTitre() );
    }
}
tx.commit();
s.close();
```

L'API Criteria
Requête par l'exemple (QBE)

- Sélectionnez les adhérents ressemblant à l'exemple donné

```
Session s = DBHelper.getFactory().openSession();
Transaction tx = s.beginTransaction();
System.out
   .println("\n Test Requete : à partir d'un exemple");
Adherent ex = new Adherent();
ex.setPrenom("Jean");
Criteria crit = s.createCriteria(Adherent.class);
crit.add(Example.create(ex));
List<Adherent> results = (List<Adherent>)crit.list();
for(Adherent a : results ){
    System.out.println(a.getId() + ") " + a.getNom());
}
tx.commit();
s.close();
```

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 !

SQL Query
Exemples (1/2)

- Sélectionnez l'adhérent de nom donné

```
Session s = DBHelper.getFactory().openSession();
Transaction tx = s.beginTransaction();

SQLQuery q =
  s.createSQLQuery("select {adh.*} from Tadherent adh where adh.nom =
'Valron'");
q.addEntity("adh",Adherent.class);
System.out.println("\n Test Requete : sélectionnez l'adhérent de nom :
[Valron]");
System.out.println("Requete : select * from Tadherent where nom = 'Valron'");
Adherent valron = (Adherent)q.uniqueResult();
System.out.println(valron.getId() + ") "+ valron.getNom() +
         " " + valron.getTelephone().getNumero());
tx.commit();
s.close();
```

```
select adh.id as id0_, adh.nom as nom5_0_,
adh.prenom as prenom5_0_, adh.tel as tel5_0_ from Tadherent adh
where adh.nom = 'Valron'
```

SQL Query
Exemples (2/2)

- Sélectionnez l'adhérent ayant loué l'item d'id 3

```
Session s = DBHelper.getFactory().openSession();
Transaction tx = s.beginTransaction();

SQLQuery q =
s.createSQLQuery("select {adh.*} from Tadherent adh,
          TLocation loc where adh.id = loc.idadh and loc.idItem=3");
q.addEntity("adh",Adherent.class);
System.out
  .println("\n Test Requete : sélectionnez les adhérents ayant
           louer l'item d'id 3");

Adherent adh = (Adherent)q.uniqueResult();
System.out.println(adh.getId() + ") "+ adh.getNom() + " " +
                adh.getTelephone().getNumero());
tx.commit();
s.close();
```

```
select adh.id as id0_, adh.nom as nom5_0_, adh.prenom as prenom5_0_,
adh.tel as tel5_0_ from Tadherent adh, TLocation loc
where adh.id = loc.idadh and loc.idItem=3
```

Exercice

- Exercice 8 : Accès aux données