JPA 2 / Hibernate 5

Mohamed DERKAOUI

25/07/2017

Plan

- Introduction
- Mapping des entités
- Gestion de la concurrence
- Gestion des collections
- Gestion de l'héritage
- Interrogation des données (JP-QL/HQL, Criteria, SQL)
- Gestion du cache

Introduction

Correspondance des modèles « Relationnel - Objet »

- Le modèle objet propose plus de fonctionnalités :

  - L'héritage, le polymorphisme

- Les relations entre deux entités sont différentes

- Les objets ne possèdent pas d'identifiant unique contrairement au modèle relationnel

Accès aux Bdds en Java

- JDBC (Java DataBase Connectivity)
- Inconvénients :

  - Nécessite l'écriture de nombreuses lignes de codes répétitives

  - La liaison entre les objets et les tables est un travail de bas niveau

- Exemple de code + pattern DAO

Mapping relationnel-objet

Concept permettant de connecter un modèle objet à un modèle relationnel.

Couche qui va interagir entre l'application et la base de données.

Pourquoi utiliser ce concept?

- Pas besoin de connaître l'ensemble des tables et des champs de la base de données
- Faire abstraction de toute la partie SQL d'une application.

Mapping relationnel-objet (2)

Avantages :

- Gain de temps au niveau du développement d'une application.
- Abstraction de toute la partie SQL.
- La portabilité de l'application d'un point de vue SGBD.

Inconvénients :

- L'optimisation des frameworks/outils proposés
- La difficulté à maîtriser les frameworks/outils.

Critères de choix d'un ORM

- La facilité du mapping des tables avec les classes, des champs avec les attributs.
- Les fonctionnalités de bases des modèles relationnel et objet.
- Les performances et optimisations proposées : gestion du cache, chargement différé.
- Les fonctionnalités avancées : gestion des sessions, des transactions
- Intégration IDE : outils graphiques
- La maturité.

JPA

- Une API (Java Persistence API)
- Des implémentations


MyBatis


ORACLE TOPLINK
JPA

- Permet de définir le mapping entre des objets Java et des tables en base de données
- Remplace les appels à la base de données via JDBC

Concepts vs Classes

Concept	*JDBC*	*Hibernate*	*JPA*
Ressource	Connection	Session	EntityManager
Fabrique de ressources	DataSource	Sessionfactory	EntityManagerFactory
Exception	SQLException	HibernateException	PersistenceException

Objets Hibernate

- **SessionFactory (EntityManagerFactory) :** Un cache threadsafe (immuable) des mappings vers une (et une seule) base de données. Une factory (fabrique) de Session et un client de ConnectionProvider. Peut contenir un cache optionnel de données (de second niveau) qui est réutilisable entre les différentes transactions que cela soit au niveau processus ou au niveau cluster.
- **Session (EntityManager) :** Un objet mono-threadé, à durée de vie courte, qui représente une conversation entre l'application et l'entrepôt de persistance. Encapsule une connexion JDBC, une Factory (fabrique) des objets Transaction. Contient un cache (de premier niveau) des objets persistants, ce cache est obligatoire. Il est utilisé lors de la navigation dans le graphe d'objets ou lors de la récupération d'objets par leur identifiant.

Objets Hibernate (2)

- **Objets et Collections persistants :** Objets mono-threadés à vie courte contenant l'état de persistance et la fonction métier. Ceux-ci sont en général les objets métier ; la seule particularité est qu'ils sont associés avec une (et une seule) Session. *Dès que la Session est fermée, ils seront détachés et libres d'être utilisés par n'importe quelle couche de l'application (i.e. de et vers la présentation en tant que Data Transfer Objects - DTO : objet de transfert de données).
- **Objets et collections transitoires (Transient) :** Instances de classes persistantes qui ne sont actuellement pas associées à une Session. Elles ont pu être instanciées par l'application et ne pas avoir (encore) été persistées ou elles n'ont pu être instanciées par une Session fermée.

Objets Hibernate (3)

- **Transaction (EntityTransaction) :** (Optionnel) Un objet mono-threadé à vie courte utilisé par l'application pour définir une unité de travail atomique. Abstrait l'application des transactions sous-jacentes. Une Session peut fournir plusieurs Transactions dans certains cas.
- **TransactionFactory :** (Optionnel) Une fabrique d'instances de Transaction. Non exposé à l'application, mais peut être étendu/implémenté par le développeur.

Entity Manager

Gère toutes les interactions avec la base de données :

- Recherche : find, get
- Sauvegarde : persist, merge
- Suppression : remove
- Requêtage : langage JP-QL (HQL), createQuery,
- Transaction : begin, commit, rollback
  (support des transactions par annotations)

EntityManager et Transaction

- L'objet EntityManager (Session) possède un ensemble de méthodes permettant d'effectuer des opérations en Bdd : persist, merge, find, …
- L'objet EntityTransaction permet de :
  - Valider une action de la session : commit();
  - Annuler une action de la session : rollback();

```
EntityTransaction tx=null;
try{
     ...
    tx = em.getTransaction().begin();
    ...
    tx.Commit();

} catch(Exception ex){
    tx.Rollback();
}
em.Close();
```

Annotations JPA

- JPA est l'API de spécification de la couche de persistance au sein d'une application client lourd / léger.

  (les spécifications EJB 3 ne traitent pas directement de la couche de persistance)

- JPA 2 est la partie de la spécification EJB 3.1 qui concerne la persistance des composants :

  http://download.oracle.com/otndocs/jcp/persistence-2.0-fr-oth-JSpec/

Mapping relationnel

```
@Entity
@Table(name = "person")
public class Person implements Serializable {

    @Id
    private long id;

    @Column(name = "first_name")
    private String firstName;

    @Temporal(TemporalType.DATE)
    @Column(name = "birth_day")
    private Date birthDay;

    public Person() {
        super();
    }

    // Getters / Setters
     ...
```

Objet géré en base

Nom de la table mappant l'objet

Clé primaire

Nom de la colonne mappant l'attribut

Attribut de type date

Gestion de la concurrence

La gestion de la concurrence est essentielle dans le cas de longues transactions. Hibernate possède plusieurs modèles de concurrence : None – Optimistic (Versioned) – Pessimistic

https://docs.jboss.org/hibernate/orm/4.0/devguide/en-US/html/ch05.html

- None : la transaction concurrentielle est déléguée au SGBD. Elle peut échouer.
- Optimistic (Versioned) : si nous détectons un changement dans l'entité, nous ne pouvons pas la mettre à jour.
  @Version(Numeric, Timestamp, DB Timestamp) :
  on utilise une colonne explicite Version (meilleure stratégie).
- Pessimistic : utilisation des LockMode spécifiques à chaque SGBD.

Gestion de la concurrence
Versioned

- L'élément @Version indique que la table contient des enregistrements versionnés. La propriété est incrémentée automatiquement par Hibernate.

  Automatiquement, la requête générée incluera un test sur ce champ :

- ```
  UPDATE Player SET version = @p0, PlayerName = @p1
  WHERE PlayerId = @p2
  AND version = @p3;
  ```
  ```

Gestion de la concurrence
Pessimistic

- Nous pouvons exécuter une commande séparée pour la base de données pour obtenir un verrou sur la ligne représentant l'entité :

```
Player player = entityManager.find(Player.class,1);
entityManager.lock(player, LockModeType.WRITE);
 player.playerName = "other";
 tx.commit();
```

- SELECT PlayerId FROM Player **with (updlock, rowlock)** WHERE PlayerId = @p0;
- UPDATE Player SET PlayerName = @p1
  WHERE PlayerId = @p2 AND PlayerName = @p3;

- Inconvénient : l'attente pour l'obtention du verrou (pour la modification si la ligne est verrouillée).
  Une exception est déclenchée après le Timeout parce que nous ne pouvions pas obtenir le verrou :
  <property name="javax.persistence.lock.timeout" value="1000" />

Relations entre Entity Beans

- **One-To-One**
  @OneToOne
  @PrimaryKeyJoinColumn
  @JoinColumn
- **Many-To-One**
  @ManyToOne
  @JoinColumn
- **One-To-Many**
  @OneToMany
  (pas de @JoinColumn)
- **Many-To-Many**
  @ManyToMany
  @JoinTable

Stratégies de chargement des relations

- Chargement tardif : **LAZY**
  Les entités en relation ne sont chargées qu'au moment de l'accès

- Chargement immédiat : **EAGER**
  Les entités en relation sont chargées dès le load de l'objet @OneToMany(mappedBy="customer", **fetch=FetchType.EAGER**)

Traitement en cascade

- Les annotations `@OneToOne`, `@OneToMany`, `@ManyToOne` et `@ManyToMany` possèdent l'attribut cascade
- Une opération appliquée à une entité est propagée aux relations de celle-ci :

  par exemple, lorsqu'un utilisateur est supprimé, son compte l'est également
- 4 Types : `PERSIST` , `MERGE` , `REMOVE` , `REFRESH`
- `CascadeType.ALL` : cumule les 4

Héritage

Il existe trois façons d'organiser l'héritage :

- **SINGLE_TABLE**
  @Inheritance
  @DiscriminatorColumn
  @DiscriminatorValue

- **TABLE_PER_CLASS**
  @Inheritance

- **JOINED**
  @Inheritance

La différence entre elles se situe au niveau de l'optimisation du stockage et des performances

Héritage : SINGLE_TABLE

- Tout est dans la même table
- Une colonne, appelée “Discriminator” définit le type de la classe enregistrée.
- De nombreuses colonnes inutilisées

```
@Entity(name="COMPTE")
@Inheritance (strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="compte_discriminator",
 discriminatorType=DiscriminatorType.STRING,length=15)
public abstract class Compte implements Serializable{…}

@Entity
@DiscriminatorValue("COMPTE_EPARGNE")
public class CompteEpargne extends Compte
       implements Serializable {…}
```

Héritage : TABLE_PER_CLASS

- Chaque Entity Bean fils a sa propre table
- Lourd à gérer pour le polymorphisme

```
@Entity
@Inheritance (strategy=InheritanceType.TABLE_PER_CLASS)
public abstract class Compte implements Serializable
{…}
La clé @Id ne peut pas être @GeneratedValue (Identity)
(mettre la génération en TABLE)


@Entity
public class CompteEpargne extends Compte
                    implements Serializable {…}
```

Héritage : JOINED

- Chaque Entity Bean a sa propre table
- Beaucoup de jointures

```
@Entity
@Inheritance (strategy=InheritanceType.JOINED)
public abstract class Compte implements Serializable
{…}


@Entity
public class CompteEpargne extends Compte
                    implements Serializable {…}
```

Héritage : récapitulatif

Stratégie	SINGLE_TABLE	TABLE_PER_CLASS	JOINED
Avantages	**Aucune jointure, donc très performant**	**Performant en insertion**	**Intégration des données proche du modèle objet**
Inconvénients	**Organisation des données non optimale**	**Polymorphisme lourd à gérer**	**Utilisation intensive des jointures, donc baisse des performances**

Requêtes

- JPA-QL (HQL)
  https://docs.jboss.org/hibernate/orm/3.3/reference/fr-FR/html/queryhql.html
- Criteria

  https://docs.jboss.org/hibernate/orm/3.3/reference/fr-FR/html/querycriteria.html
- SQL
  https://docs.jboss.org/hibernate/orm/3.6/reference/fr-FR/html/querysql.html
- Requêtes nommées

Gérer les exceptions

- **Récupérer les DataAccessException**

La cause de l'exception est l'erreur SQL.

```
try {
    dao.delete(delete1);
} catch (DataAccessException e) {
    System.out.println(e.getMessage());
    System.out.println(e.getCause());
    SQLException sqlex = (SQLException) e.getCause();
    System.out.println(sqlex.getErrorCode());
    System.out.println(sqlex.getSQLState());
}
```

Intercepteurs

- Hibernate fournit des intercepteurs (callbacks) au niveau de l'entité permettant d'effectuer des traitements.
- Interfaces à implémenter :
  - **Lifecycle :** traitement sur la sauvegarde, mise à jour, suppression ou le chargement d'un objet (**propre à l'entité**)
  Peut être utilisée pour gérer la cascade au lieu du mapping.
  On peut utiliser des annotations JPA au lieu de cette interface :
  @PrePersist, @PostPersist, @PostUpdate,...

  - **Interceptor :** callback de la session à l'application pour introspecter les propriétés d'une entité. A déclarer dans les propriétés d'Hibernate :

```
<property name="hibernate.ejb.interceptor"
        value="nompackage.NomInterceptor" />
```

  ou utiliser les mêmes annotations JPA + @EntityListeners sur l'entité
  https://docs.jboss.org/hibernate/orm/4.0/hem/en-US/html/listeners.html

Intercepteurs Lifecycle

- public interface Lifecycle

```
{
        boolean onSave(Session s);              (1)
        boolean onUpdate(Session s);            (2)
        boolean onDelete(Session s);            (3)
        void onLoad(Session s, object id);      (4)
}
```

- Le retour est boolean : VETO (true), NO_VETO (false)

(1) **OnSave** - called just before the object is saved or inserted
OnSave() is called after an identifier is assigned to the object, except when native key generation is used.

(2) **OnUpdate** - called just before an object is updated
(when the object is passed to Isession.Update())
OnUpdate() is not called every time the object's persistent state is updated. It is called only when a transient object is passed to ISession.Update().

(3) **OnDelete** - called just before an object is deleted

(4) **OnLoad** - called just after an object is loaded

Intercepteurs
JPA Callbacks

Définir des méthodes dans l'entité concernée :

- **@PrePersist** - before a new entity is persisted (added to the EntityManager).
- **@PostPersist** - after storing a new entity in the database (during commit or flush).
- **@PostLoad** - after an entity has been retrieved from the database.
- **@PreUpdate** - when an entity is identified as modified by the EntityManager.
- **@PostUpdate** - after updating an entity in the database (during commit or flush).
- **@PreRemove** - when an entity is marked for removal in the EntityManager.
- **@PostRemove** - after deleting an entity from the database (during commit or flush).

Intercepteurs
Interceptor

On peut implémenter l'interface Interceptor ou mieux, hériter de **EmptyInterceptor**.

```
public class AuditInterceptor extends EmptyInterceptor {

    private EntityManager em;

    public AuditInterceptor() {
        System.out.println("AuditInterceptor constructed");
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("PU_2");
        em = emf.createEntityManager();
    }
    @Override
    public boolean onSave(Object entity, Serializable id, Object[] state, String[] propertyNames, Type[] types) {
        System.out.println("onSave called");
        em.getTransaction().begin();
        AuditLog a = new AuditLog();
        a.setEntry("object with ID: " + id + " saved");
        em.persist(a);
        em.getTransaction().commit();
        return false;
    }

    @Override
    public void afterTransactionCompletion(Transaction tx) {
        System.out.println("afterTransactionCompletion called");
        em.getTransaction().begin();
        AuditLog a = new AuditLog();
        a.setEntry("transaction: " + tx + " completed");
        em.persist(a);
        em.getTransaction().commit();
    }
}
```

Listeners

- Méthodes de callbacks à appliquer sur une entité :

```
public class MyListener {
    @PrePersist void onPrePersist(Object o) {}
    @PostPersist void onPostPersist(Object o) {}
    @PostLoad void onPostLoad(Object o) {}
    @PreUpdate void onPreUpdate(Object o) {}
    @PostUpdate void onPostUpdate(Object o) {}
    @PreRemove void onPreRemove(Object o) {}
    @PostRemove void onPostRemove(Object o) {}
}
```

- Application sur l'entité d'un ou plusieurs listeners ; on peut gérer également l'héritage :

```
@Entity @EntityListeners(MyListener.class)
public class MyEntityWithListener {
}
@Entity @EntityListeners({MyListener1.class, MyListener2.class})
public class MyEntityWithTwoListeners {
}
```

- On peut exclure des listeners hérités d'une super classe :

```
@Entity @ExcludeSuperclassListeners
public class EntityWithNoListener extends EntityWithListener {
}
```

Listeners par défaut

- On peut définir des listeners par défaut (pas d'annotation, fichier META-INF/orm.xml) :

```
<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
 http://java.sun.com/xml/ns/persistence/orm_1_0.xsd" version="1.0">
  <persistence-unit-metadata>
    <persistence-unit-defaults>
      <entity-listeners>
        <entity-listener class="fr.dawan.JpaHibernateTraining.tools.MyDefaultListener" />
      </entity-listeners>
    </persistence-unit-defaults>
  </persistence-unit-metadata>
</entity-mappings>
```

- On peut exclure les default listeners pour une entité :

```
@Entity @ExcludeDefaultListeners
public class NoDefaultListenersForThisEntity {
}

@Entity
public class NoDefaultListenersForThisEntityEither extends NoDefaultListenersForThisEntity {
}
```

Stratégies de chargement (fetching)

- Une stratégie de fetch définit le comportement de Hibernate pour récupérer les objets associés si l'application a besoin de naviguer dans les associations. Définie par déclaration ou dans la requête.
- Hibernate définit plusieurs stratégies :

  - **Join fetching** : Hibernate récupère les instances associées ou les collections dans le même SELECT (en utilisant un OUTER JOIN).

  - **Select fetching :** un second SELECT est utilisé pour récupérer les associations ou les collections. Sauf si vous désactivez explicitement le lazy fetching (lazy="false"), le second SELECT sera exécuté uniquement à l'accès de l'association.

  - **Subselect fetching :** un second SELECT est utilisé pour récupérer toutes les collections associées récupérées par une précédente requête ou un fetch. Sauf si vous désactivez explicitement le lazy fetching (lazy="false"), le second SELECT sera exécuté uniquement à l'accès de l'association.

  - **Extra-lazy collection fetching :** des éléments individuels de la collection sont accessibles à partir de la BDD selon les besoins. NHibernate essaie de ne pas charger toute la collection en mémoire sauf si c'est absolument nécessaire (adapté pour les très grandes collections).

  - **Batch fetching :** une stratégie d'optimisation du select fetching. NHibernate récupère un lot d'instances ou de collections en un seul SELECT en spécifiant une liste de clés primaires ou de clés étrangères.

Stratégies de chargement (fetching)

- Hibernate marque également une distinction entre :

  - Immediate fetching : une association, une collection ou un attribut est chargé immédiatement lorsque le propriétaire est chargé (Eager).

  - Lazy collection fetching : une collection est chargée lorsque l'application invoque une méthode sur cette collection (ceci est la valeur par défaut pour les collections).

  - Proxy fetching : une association unique est chargée lorsqu'une méthode autre que le getter sur l'identifiant est appelée sur l'objet associé.

- Mise en place :

  - par code (dans la requête)

Batch fetching

Batch fetching est **une optimisation du lazy select fetching**. 2 manières pour le tuning :

- **Au niveau de la classe/entités :**
  Scénario 1 : plusieurs Adresses pour 1 Client. Adresse a une référence vers le Client.
  - 25 instances de « Adresse » chargées en session.
  - La classe Client est mappé avec un proxy, lazy="true".
  - Si on boucle sur les adresses et on appelle le client associé, Hibernate exécutera 25 SELECT pour récupérer les propriétaires proxifiés.
  On peut optimiser ce comportement en spécifiant un batch-size dans le mapping de Client :
  <class name="Client" batch-size="10">...</class> ou **@BatchSize(size=30)**
  Hibernate exécutera seulement 3 requêtes. Le pattern sera 10, 10, 5.

  sinon au niveau de la requête setFetchSize(30)

- **Au niveau de la collection :**
  Scénario 2 : 1 Client possède une lazy collection d'objets Adresse - OneToMany
  - 10 instances « Client » chargées en session.
  - Si on boucle sur les clients, Hibernate générera 10 SELECT, 1 pour chaque appel de client.Adresses. On peut pré-charger la collection :

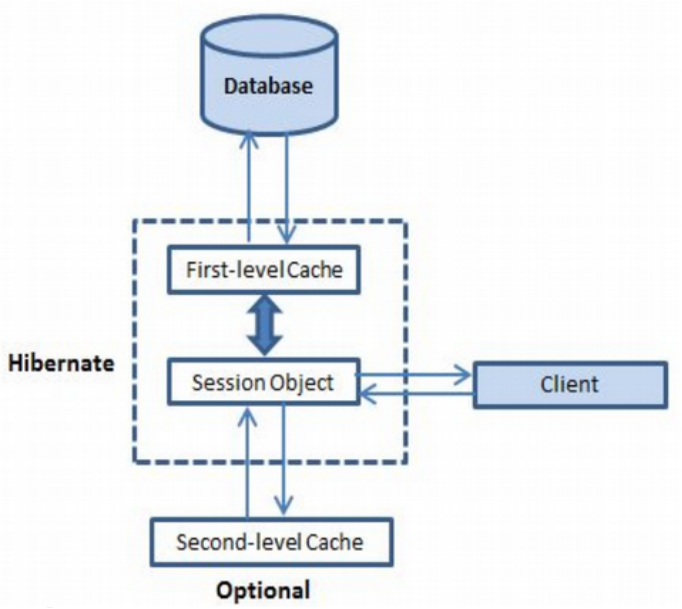
```
<class name="Client">
    <set name="Adresses" batch-size="3">...</set>
</class>
```

  Avec un batch-size à 3, Hibernate chargera 3, 3, 3, 1 collections en 4 requêtes SELECT.
  La valeur de l'attribut dépend du nombre de collections non initialisées dans une Session particulière .
  Usage du batch fetching : un arbre imbriqué d'objets

Cache de niveau 1

Hibernate possède 2 niveaux de cache.

- **Cache de niveau 1 :**
  - Chaque fois que vous passez un objet à persist(), merge() ou que vous récupérez un objet avec find(), cet objet est ajouté au cache interne de la session (EntityMana
  - Quand flush() est ensuite appelée, l'état de cet objet va être synchronisé avec la base de données. Si vous ne voule que cette synchronisation se produit ou si vous traitez un grand nombre d'objets et la nécessité de gérer efficacement la mémoire :
  - La méthode **detach()** peut être utilisée pour supprimer l'objet et ses collections dépendantes du cache de premier niveau.
  - La méthode **clear()** permet de vider complètement le cache de la session.



Cache de niveau 1 (suppression)

```
List<Book> books = em.createQuery("from Book").list();
//un grand résultat

for (Book b : books)

{

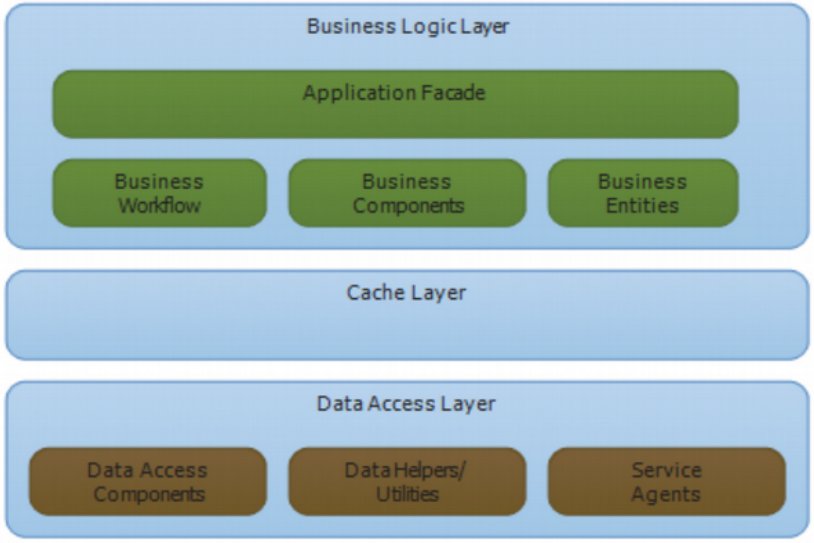
    DoSomethingWithABook(b);

    Session.evict(b); //em.detach(b);

}
```

Cache de niveau 2 Introduction

- La mise en cache des entités est une technique très importante pour améliorer les performances de l'application.
  Généralement, on introduit une couche de mise en cache dans une architecture multi-couches **avant la couche d'accès aux données**.
  Parfois, on utilise les composants web côté présentation pour mettre en cache les entités :
  - Session
  - Application (servletContext)



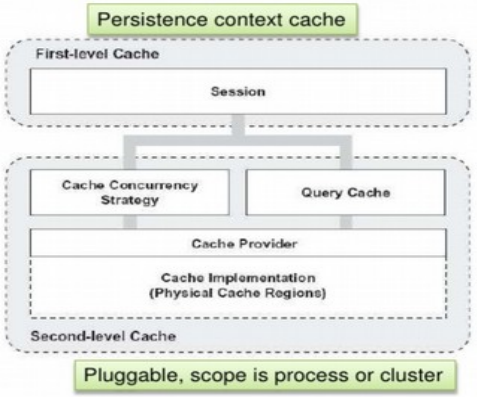
Hibernate fournit un cache de niveau 2 (couche accès aux données) qui permet de s'abstraire de l'utilisation des composants de la couche présentation ou métier.

Cache de niveau 2 Configuration

```
<property name="cache.use_second_level_cache">true</property>
<property name="cache.provider_class">...</property>
<property name="cache.use_query_cache">true</property>
<property name="prepare_sql">true</property>
```

- De multiples cache providers sont disponibles :
  https://docs.jboss.org/hibernate/stable/core.old/reference/fr/html/performance-cache.html

  Exemple **ehCache** :
  http://www.baeldung.com/hibernate-second-level-cache



Cache de niveau 2
Caching des entités

- Pour spécifier la mise en cache d'une entité ou d'une collection, on ajoute l'annotation @Cache dans le mapping de la classe ou de la collection ou une conf. Xml :
- L'attribut **usage** spécifie la stratégie de gestion de la concurrence :
  - read only : cache en lecture seule, pas de modification d'instances persistantes (manière la plus simple et la plus performante).
  - read/write : si l'application doit mettre à jour des données. On doit s'assurer que la transaction est terminée et que la session est fermée (strict).
  - nonstrict read/write : si l'application doit occasionnellement mettre à jour des données (multiples transactions simultanées).

Cache de niveau 2
Suppression d'entités

- **Suppression d'un book spécifique :**
```
entityManagerFactory.getCache().evict(Book.class, bookId);
```
- **Suppression de toutes les instances Book :**
```
entityManagerFactory.getCache().evict(Book.class);
```
- **Suppression d'une collection Chapters d'un objet Book :**
```
entityManagerFactory.getCache()
        .evictCollection("Book.Chapters", bookId);
```
- **Suppression de toutes les collections Chapters :**
```
entityManagerFactory.getCache()
        .evictCollection("Book.Chapters");
```

Cache de niveau 2
Caching de requêtes

- Les **résultats** d'une requête peuvent être mis en cache.
  Utile uniquement pour les requêtes exécutées fréquemment avec les mêmes paramètres.
- Configuration (persistence.xml ou hibernate.cfg.xml) :
  `cache.use_query_cache = true`
- Utilisation : `Query.setCacheable(true)`

```
List<Book> booksList2 = em.createQuery("FROM Book b")
                          .setCacheable(true).setCacheRegion("ShortTermCacheRegion")
                          .list();
// ……
```

- Forcer le rafraîchissement :
  entityManagerFactory.getCache().`evictQueries(regionName).`