



Hibernate Advanced

David THIBAU – 2019

david.thibau@gmail.com



Agenda

Reviewing

- ORM and « impedance mismatch »
- Entities and Session
- Basic mapping, *GeneratedValue*
- Mapping associations, embeddeed objects

Loading associations

- Lazy and fetch attributes
- Lazy initialization Exception

Hibernate 2nd Level cache

- Entity cache configuration
- Collection and Query cache

Transactions management

- Contextual session patterns
- Concurrency control strategies
- User transactions

Miscellaneous

- Inheritance mapping
- Interceptors and Events
- Batch processing

Monitoring & performance

- Way to monitor
- Perf. recommandations



ORM and Impedence mismatch

Technical view

Object Model

- Model both the data and the treatments.
- Inheritance
- Objective : Easy to manipulate, near the business model

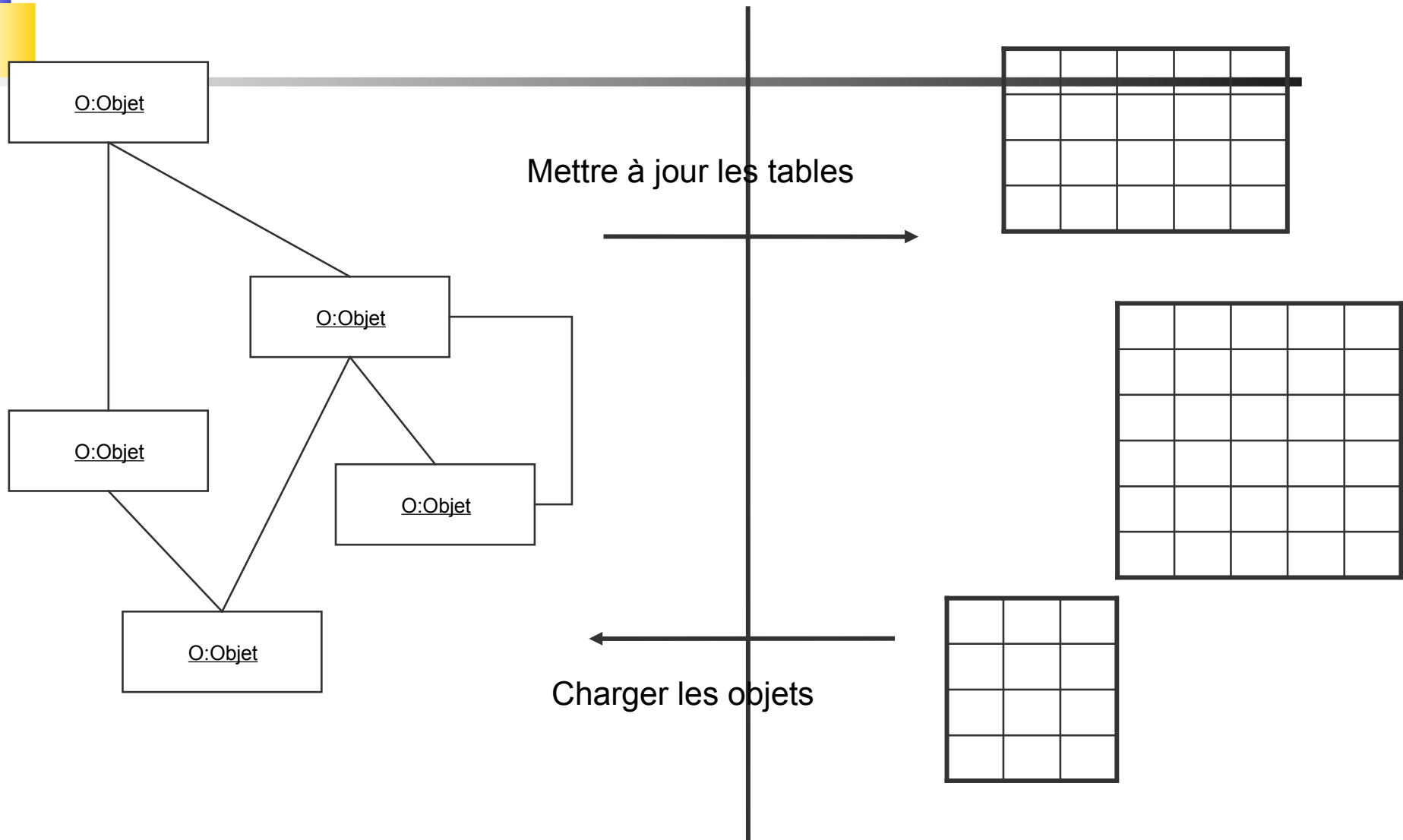
Relational model

- Only data
- Mathematical foundation : Keys and relations
- Objective : Optimize storage and queries



Impedence mismatch

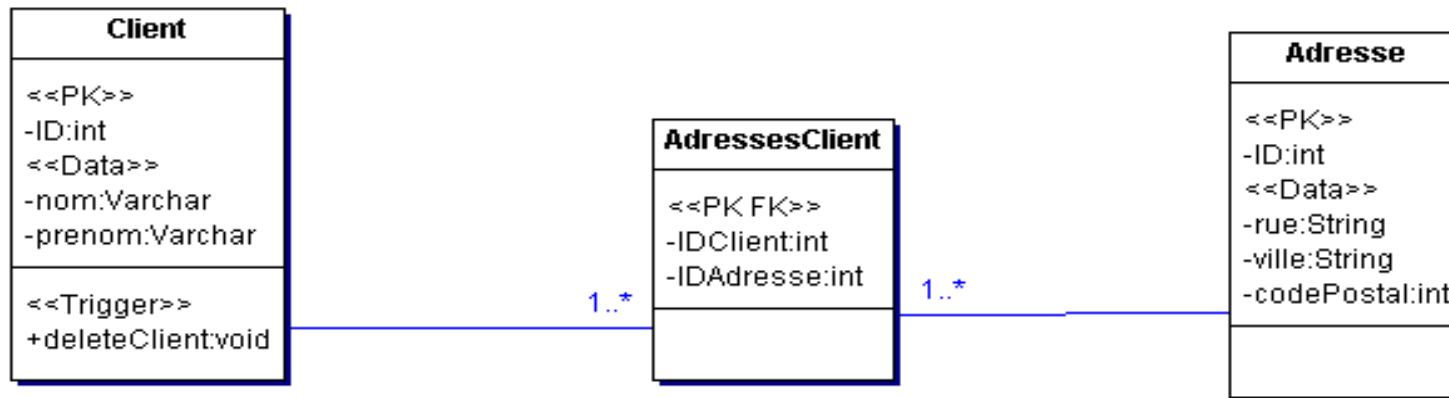
Instances persistantes vs Occurrences



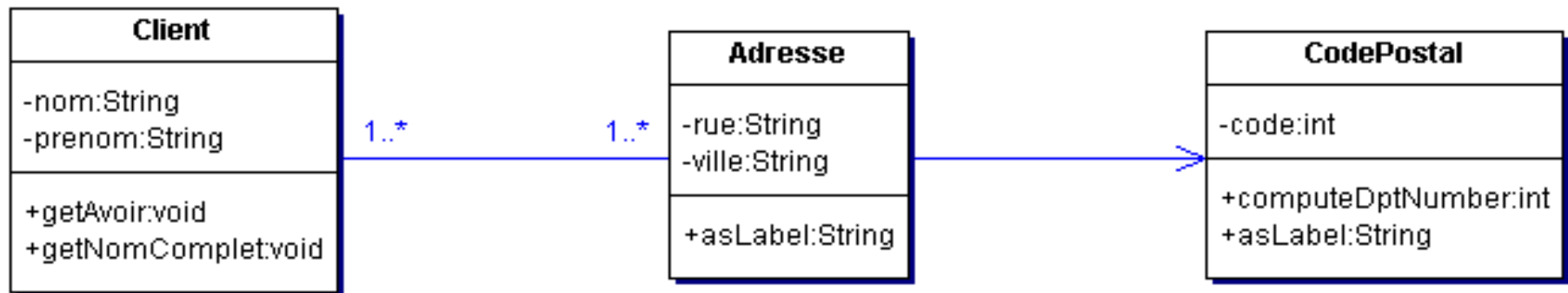
Impedence mismatch

Misleading appearance

Relational



Object



Impedence mismatch

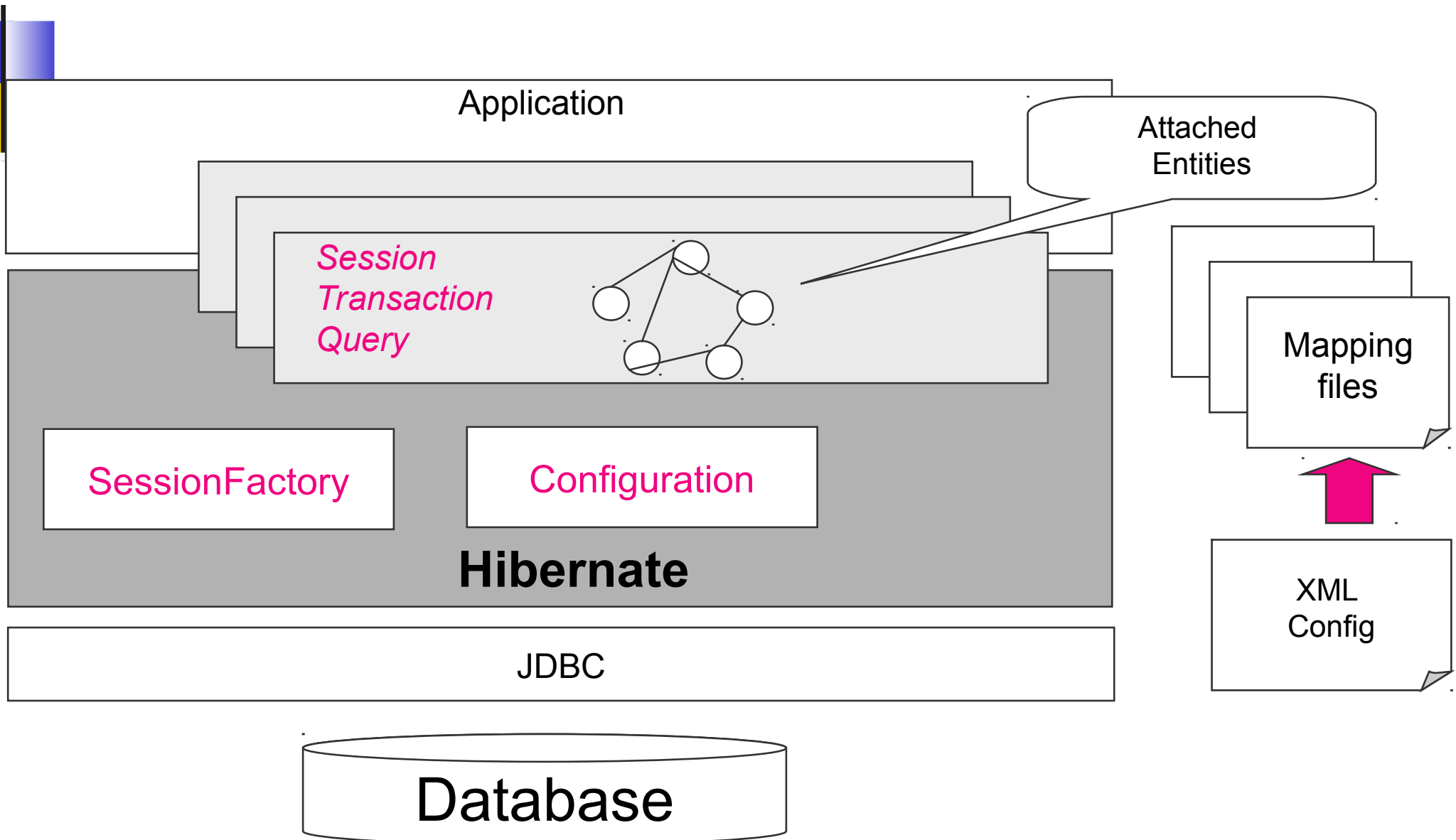
Problems to solve

- Object transient / persistent
- Identification and uniqueness of an object
- Persistence of an object tree (loading, storing, navigating), association representation
- Inheritance mapping
- User Transactions (Atomic committing, Cache)

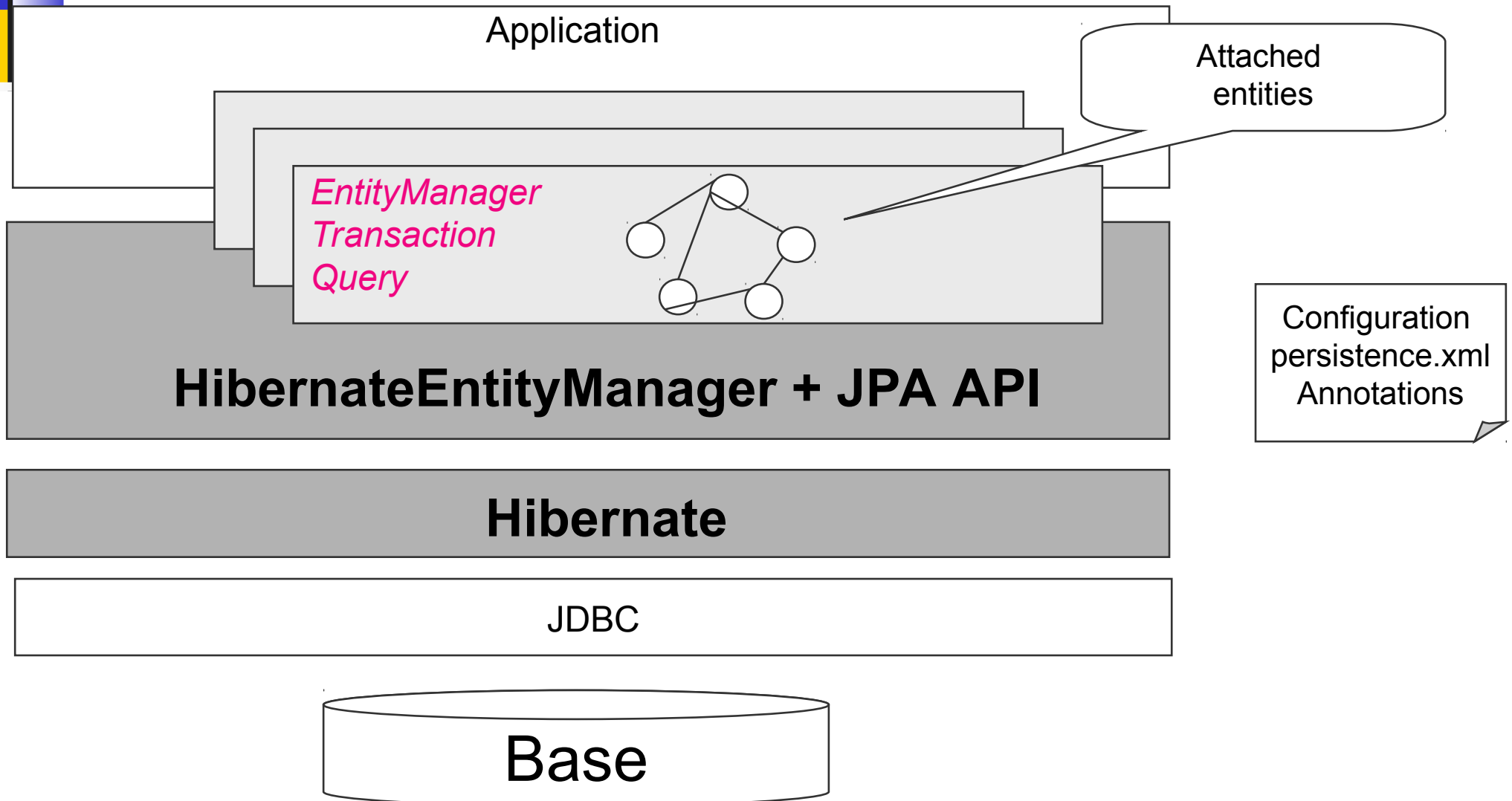


Entities and Session

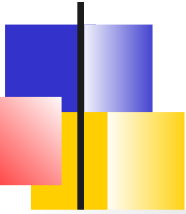
Hibernate Architecture



Hibernate/JPA Architecture



Hibernate / JPA



Nom	JPA	Hibernate	Description
Factory	<i>EntityManagerFactory</i>	<i>SessionFactory</i>	Encapsulate database configuration and allows to instantiate persistence manager
Peristance Manager	<i>EntityManager</i>	<i>Session</i>	Offer API for CRUD operations, query creation of entities. Manage a persistence context
Persistence context			1st level cache of database A set of unique persistent classes attached to the Session. Short-live object Cache is generally flushed with database transaction
Persistence Unit	<i>persistence.xml</i>	<i>hibernate.cfg.xml</i>	A set of entities mapped to a database.

Sample *hibernate.cfg.xml*

```
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
  <session-factory>
    <property name="connection.driver_class">
      org.gjt.mm.mysql.Driver</property>
    <property name="hibernate.connection.url">
      jdbc:mysql://172.16.102.42/Mediatheque</property>
    <property name="hibernate.connection.username">
      hve</property>
    <property name="hibernate.connection.password">
      pwd</property>
    <property name="show_sql">
      true</property>

    <mapping resource="com/plb/etechno/j12/exemple/metier/Theme.hbm.xml"/>
    <mapping resource="com/plb/etechno/j12/exemple/metier/MotClef.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

Database Connection
properties

Property to enable
SQL tracing

Mapping files

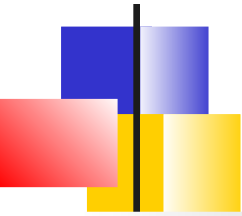


Persistence unit

- ❖ Represent a set of entities mapped to the same DB
- ❖ Defined in *persistence.xml*, it specifies:
 - How to connect to DB (JDBC or JNDI DataSource)
 - Transaction type : *RESOURCE_LOCAL* or *JTA*
 - The JPA implementation
 - Hibernate (or other ORM) specific properties

Sample *persistence.xml*

Java EE



```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0">
  <!-- Name of persistence unit -->
  <persistence-unit name="enterprise" transaction-type="JTA">
    <!-- JPA Implémentation -->
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <!-- JNDI DataSource managed by the server-->
    <jta-data-source>java:/DefaultDS</jta-data-source>

    <!-- Hibernate specific properties -->
    <properties>
      <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
    </properties>
  </persistence-unit>
</persistence>
```



Other Hibernate properties

Core configuration :

hibernate.dialect : Required for DDL generation

hibernate.show_sql, hibernate.format_sql : SQL tracing

hibernate.generate_statistics : Performance tuning

Fetching :

hibernate.jdbc.fetch_size, hibernate.jdbc.default_batch_fetch_size, hibernate.max_fetch_depth, ...

Batch (Update SQL Grouping):

hibernate.jdbc.batch_size,

2nd level-cache

*hibernate.cache.**

Intégration with JTA Transaction Manager

*hibernate.transaction.jta.platforma, hibernate.jta.**



hibernate.hbm2ddl.auto

The property ***hibernate.hbm2ddl.auto*** has 4 possible values :

- ***validate*** : Data base schema is validated before Hibernate starts. If not coherent with Object Model => Exception.
- ***update*** : If data base is not coherent, it is updated with ALTER TABLE. No data is deleted in database
- ***create*** : The database is created from the object model
- ***create-drop*** : The database is created from the object model and deleted when Hibernate stops



Sample Initialize Hibernate

```
package com.tsystems.etechno.jl2.exemple.hibernate;

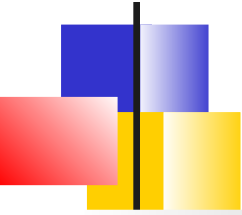
import org.hibernate.HibernateException;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class DBHelper {
    private static Configuration cfg = null;
    private static SessionFactory factory = null;

    static {
        try {
            cfg = new Configuration();
            factory = cfg.configure().buildSessionFactory();
        } catch (HibernateException e) {
            System.err.println("problème d'initialisation d'Hibernate");
            throw e;
        }
    }

    public static SessionFactory getFactory(){return factory;}
}
```


JPA bootstrapping



❖ Only once, to create the singleton *EntityManagerFactory*

– Java SE environment :

```
EntityManagerFactory emf =  
    Persistence.createEntityManagerFactory("entreprise") ;
```

– JavaEE environnement, factory is injected by the application server :

```
@PersistenceUnit
```

```
private EntityManagerFactory emf
```



Session and first level cache

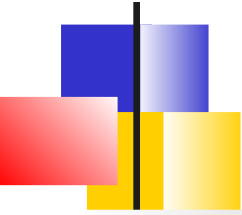
- The Hibernate session can be viewed as a short time **cache** of the database
 - Throughout the lifetime of the session, entity objects are attached to the cache.
 - Updates (creation, modification, deletion) are performed on the objects in memory
 - Sometimes, the cache is synchronized with the database and Hibernate generates the necessary SQL statements
 - The default behavior is to synchronize the cache only when the session is closed
 - We can force the cache to synchronize with *flush()*



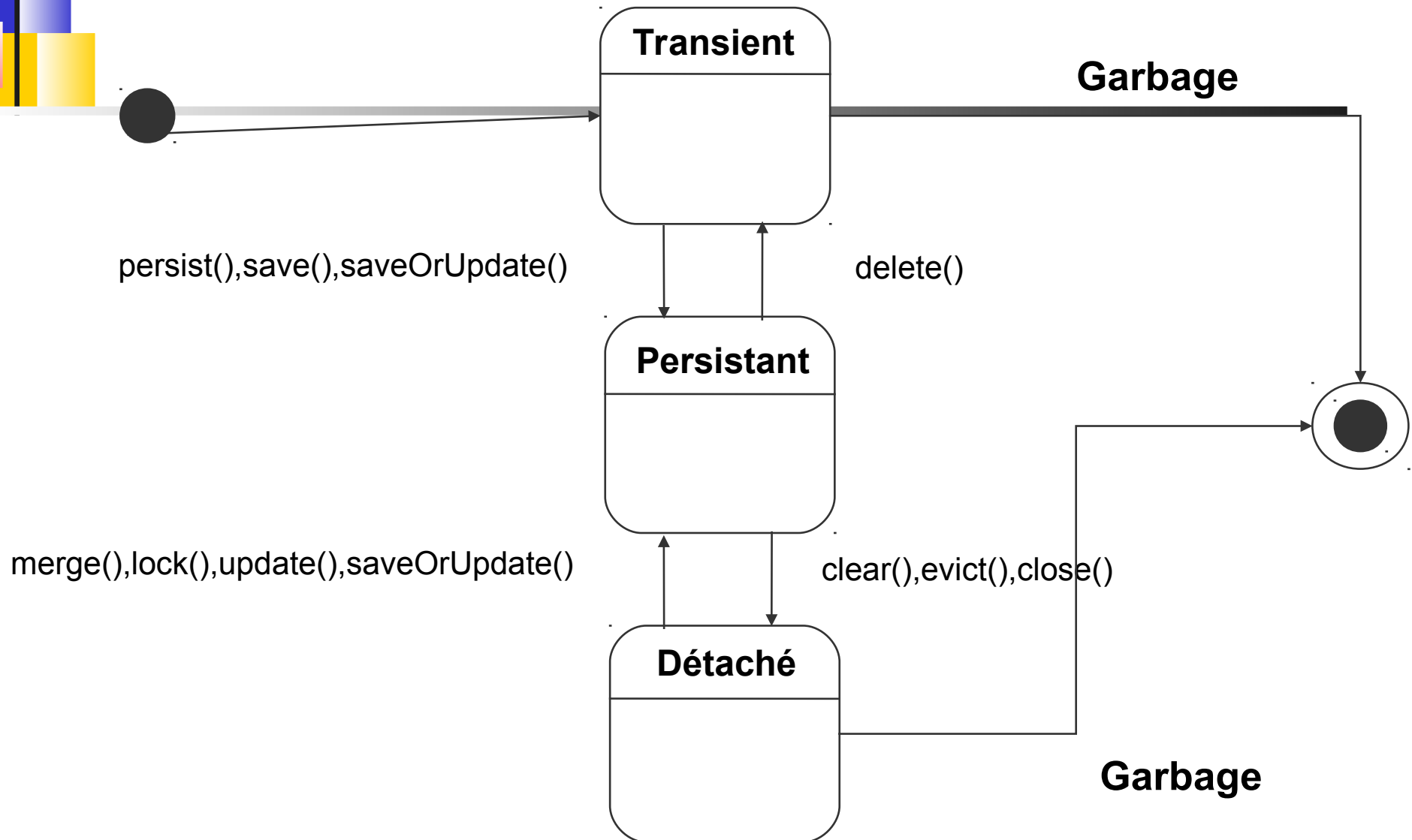
Object identity

- Object identity (`==` operator)
 - Same heap location in the JVM
 - Objects equality (*`equals()`* and *`hashCode()`* methods)
 - Same values but not necessary the same location in heap
 - Relational identity
 - Based upon primary key
- => Hibernate guarantees that there are never 2 separate instances of the same entity inside the same session
- => In general, this means that *`equals()`* and *`hashCode()`* rely on the relational identity

Status of an entity instance vs a session

- 
-
- Transient
 - Has no image in the database
 - If it is garbaged, data are lost
 - Persistent
 - Has an image in the database
 - Attached to a Session which guarantees the synchronization between the memory state and the database state
 - Detached
 - Has an image in the database
 - But is not associated to a session

State diagram





FlushMode Hibernate

- By default, explicit calls to *flush()* are not needed. The flush is performed when the session is closed.
- In certain cases, we want to override this default behavior by defining the flushMode via :
void setHibernateFlushMode(FlushMode flushMode) ;
- Hibernate provides 4 choices :
 - ALWAYS
 - AUTO (default)
 - COMMIT
 - MANUAL



Entity Classes

- ❖ *Serializable* Java Beans with a primary key (simple basic attribute or class or composite)
- ❖ Mapping meta-data concerns :
 - *Id* definition
 - Data Type precision
 - Physical model adaptation (optional)
 - Association between entities



Session API

- Non Thread-safe implementation

Get by Id

get/load

Insertion

persist, save, saveOrUpdate

Update database

update, saveOrUpdate, merge

Cache management

refresh, clear, evict, contains, flush, close

Query

CreateQuery

Transaction

beginTransaction



EntityManager API

Get by identity

```
<T> T find(Class<T> entityClass, Object primaryKey)
```

```
<T> T getReference(Class<T> entityClass, Object primaryKey)
```

Insertion, deletion

```
void persist(Object entity);
```

```
void remove(Object entity);
```

Update cache with a detached object

```
<T> T merge(T entity);
```

Cache management

```
void refresh(Object entity);
```

```
boolean contains(Object entity);
```

```
void clear();
```

Query, Transaction

```
Query createQuery, createNamedQuery,  
getTransaction, joinTransaction
```



API Hibernate

Query and Transaction interfaces

- Query
 - Represent a query of DB
 - Associated to a session
 - *mySession.createQuery(« from Theme »)*
 - Allow parameters
 - *list()* , *uniqueResult()*
- Transaction
 - Represents a unit of work
 - Associated to a session
 - For one session, only one transaction at one time.
 - *commit()* / *rollback()*



Sample Mapping : Theme.hbm.xml

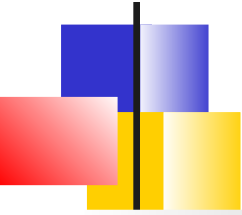
```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping SYSTEM
"C:\users\hve\Veille_technologique\cours_hibernate\
hibernate-3.0\hibernate-mapping-3.0.dtd">

<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" lazy="false" cascade="all">
      <key column="IDTheme"/>
      <one-to-many class="MotClef"/>
    </set>
  </class>
</hibernate-mapping>
```

Example Query a Theme



```
public Theme getThemeCompleet(String label){  
    Session session = DBHelper.getFactory().openSession();  
    Transaction tx = null;  
    Theme ret = null;  
    try {  
        tx = session.beginTransaction();  
        Query hqlQuery =  
            session.createQuery("from Theme t where t.label like :labelSearch ");  
        hqlQuery.setString("labelSearch", "%" + label + "%");  
        ret = (Theme)hqlQuery.uniqueResult();  
        tx.commit();  
    }..... // gestion des exception  
        return ret;  
    }  
}
```

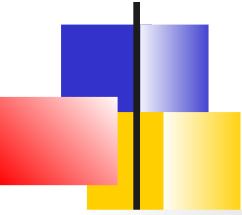
1

2

Hibernate log

```
Hibernate: select theme0_.id as id, theme0_.label as label0_ from TTheme theme0_ where theme0_.label like ?  
Hibernate: select motclefs0_.IDTheme as IDTheme1_, motclefs0_.ID as ID1_, motclefs0_.ID as ID0_,  
        motclefs0_.mot_clef as mot2_1_0_ from tmotclef motclefs0_ where motclefs0_.IDTheme=?
```

Sample Persist and cascading



```
public void createTheme(Theme t) {  
    Session session = DBHelper.getFactory().openSession();  
    Transaction tx = null;  
    Theme ret = null;  
    try {  
        tx = session.beginTransaction();  
  
        session.persist(t);  
  
        tx.commit();  
        ... // gestion des exceptions  
    }  
}
```

1

2

3

4

Hibernate log

```
Hibernate: insert into TTheme (label) values (?)  
Hibernate: insert into tmotclef (mot_clef) values (?)  
Hibernate: insert into tmotclef (mot_clef) values (?)  
Hibernate: insert into tmotclef (mot_clef) values (?)  
Hibernate: update tmotclef set IDTheme=? where ID=?  
Hibernate: update tmotclef set IDTheme=? where ID=?  
Hibernate: update tmotclef set IDTheme=? where ID=?
```

Example JPA : Theme Entity

@Entity

```
public class Theme implements Serializable {
    @Id @GeneratedValue
    private Long id;
    private String label;
    private Set<MotClef> motclefs = new HashSet<MotClef>();
    @OneToMany(cascade=CascadeType.ALL, fetch=fetchType.LAZY)
    public Set<MotClef> getMotclefs() {
        return motclefs;
    }
    public void setMotclefs(Set<MotClef> motclefs) {
        this.motclefs = motclefs;
    }
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public String getLabel() { return label ; }
    public void setLabel(String label) { this.label = label; }
    ... etc
}
```



Sample *getResultList()*

```
public List<Theme> getAllThemes() {
    List<Theme> ret = null;
    EntityManager em = emf.createEntityManager();
    EntityTransaction tx = em.getTransaction();
    try {
        tx.begin();
        Query jqlQuery = em.createQuery("from Theme");
        ret = (List<Theme>) jqlQuery.getResultList();
        tx.commit();
    }
    // Gestion des exceptions
    return ret;
}

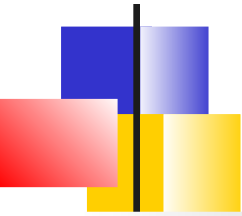
em.close()
```



Example

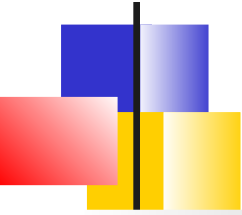
Update from detached entity

```
public void updateTheme(Theme t) {  
    EntityManager em = emf.createEntityManager();  
    EntityTransaction tx = em.getTransaction();  
    Theme ret = null;  
    try {  
        tx.begin();  
        em.merge(t);  
        tx.commit();  
    }  
    ...// gestion des exceptions  
    em.close();  
}
```

Basic Mapping

Approaches

- 
-
- ❖ 3 approaches to mapping :
 - Annotations
 - XML descriptor JPA 2
 - *hbm.xml* file
 - ❖ Metadata can be classified :
 - Logical (Object Model, associations, ...)
 - Physical (Tables, columns, foreign key, association table, index, ...)
 - ❖ Hibernate supports JPA annotation from *javax.persistence.**, and specific annotations from *org.hibernate.annotations.**



Entity class

- The 2 required annotations are
 - **@Entity** on the class
 - **@Id** on an attribute or a getter

Key generation



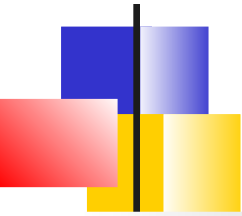
@GeneratedValue specifies that primary must be generated automatically:

@Id

@GeneratedValue

```
public int getId() {  
    return id;  
}
```

GenerationType



The strategy to generate the key is specified with the *strategy* attribute :

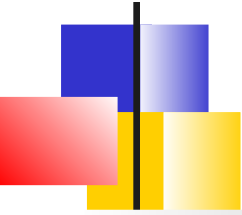
- *GenerationType.IDENTITY*: Use an identity column from the database
- *GenerationType.TABLE*: Requires a table specified with *@TableGenerator*
- *GenerationType.SEQUENCE*: Specify a sequence in the database (Default : *hibernate_sequence*)
- *GenerationType.AUTO*: Hibernate chooses in function of the features supported by the database



JavaBean Properties

- When using annotations, each property of the entity is persistent \Leftrightarrow **@Basic** annotation
- Use **@Transient** if a property is not persistent
- With *hbm.xml*, each property must be explicitly specified
- **@Basic** allow to precise attributes :
 - **fetch** : The fetching strategy. A column may be lazy.
 - **optional** : true/false (But better is **@NotNull**)

@Temporal



@Temporal annotation is suited for *java.util.Date* or *java.util.Calendar* properties. It precises the SQL type in database

- DATE, TIME or TIMESTAMP (default)

@Lob

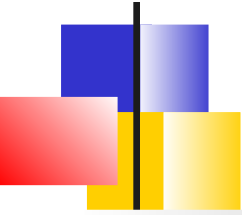


@Lob represents binary or character data which are not bounded (in size)

The SQL type is then BLOB or CLOB:

- BLOB if *byte[], Byte[], Serializable*
- CLOB if *char[], Character[], String*

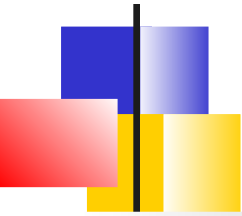
@Enumerated



@Enumerated is suited for *enum* properties. It specifies the SQL type of the associated column via the *EnumType* attribute

- *ORDINAL* (default) or *STRING*

Sample



```
@Entity
public class Personne implements Serializable{
    @Id private int id;
    private String nom;
    private String prenom;

    @Temporal(TemporalType.DATE)
    private Date naissance;

    @Lob private Byte[] photo;

    @Enumerated(EnumType.STRING)
    private Genre sexe;
```



@Table

- The physical annotation *@Table* allow to specify :
 - ***name*** : A different name for the table associated with the entity
 - ***catalog*** : The catalog
 - ***schema*** : The schema
 - ***uniqueConstraints*** : An array of unique constraints



Sample

```
@Entity
@Table(name="TBL_FLIGHT", schema="AIR_COMMAND",
uniqueConstraints=
@UniqueConstraint(name="flight_number",
columnNames={"comp_prefix", "flight_number"} ) )
public class Flight implements Serializable {
    @Column(name="comp_prefix")
    public String getCompagnyPrefix() {return companyPrefix;}

    @Column(name="flight_number")
    public String getNumber() { return number; }
}
```



@Column

- **@Column** allow to control the column associated to a property :
 - **name** : The name of the column
 - **unique** (true/false): Unique constraint
 - **nullable** (true/false): Required constraint
 - **insertable** (true/false): included in INSERT order
 - **updatable** (true/false): included in UPDATE order
 - **columnDefinition** (optional): override the DDL for this column
 - **table** : Target table (default to primary table)
 - **length** : Size of the column (default 255)
 - **precision** : Decimal precision (default 0)
 - **scale** (optional): Decimale scale (default 0)



Mapping associations



Aspects of associations

- Structural
 - Foreign key, Shared primary, association table
 - Cardinality and navigation
 - Hibernate collection and Java collection
- Fetching
 - When : Lazy-loading / eager
 - How : Fetch mode N+1, Join
- Lifecycle
 - Cascade
 - Composition and embeddeed objects



Combination

With cardinality and navigation, we can have 7 cases :

- Uni-directional association
 - One-to-One
 - One-to-Many
 - Many-to-One
 - Many-to-Many
- Bi-directional associations
 - One-to-one
 - One-to-Many / Many-to-One
 - Many-To-Many

Most of these cases can be mapped using foreign key or association table or shared primary key.

Some combination, if possible, are not recommended.

Ex : Uni-directional *OneToMany* with foreign key

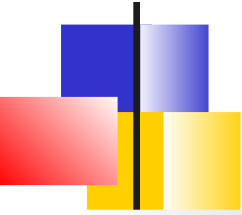


Many-to-One

```
<class name="Person">
  <id name="id" column="personId">
    <generator class="native"/>
  </id>
  <many-to-one name="address"
    column="addressId"
    not-null="true"/>
</class>
```

```
<class name="Address">
  <id name="id" column="addressId">
    <generator class="native"/>
  </id>
</class>
```

```
create table Person ( personId bigint not null primary key, addressId bigint not null )
create table Address ( addressId bigint not null primary key )
```

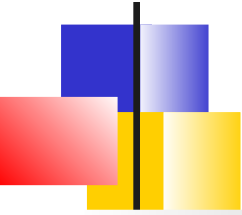


One-to-One (foreign key)

```
<class name="Person">
  <id name="id" column="personId">
    <generator class="native"/>
  </id>
  <many-to-one name="address"
    column="addressId"
    unique="true"
    not-null="true"/>
</class>
```

```
<class name="Address">
  <id name="id" column="addressId">
    <generator class="native"/>
  </id>
</class>
```

```
create table Person ( personId bigint not null primary key, addressId bigint not null unique )
create table Address ( addressId bigint not null primary key )
```



One-to-One (primary shared key)

```
<class name="Person">
  <id name="id" column="personId">
    <generator class="native"/>
  </id>
</class>
```

```
<class name="Address">
  <id name="id" column="personId">
    <generator class="foreign">
      <param name="property">person</param>
    </generator>
  </id>
  <one-to-one name="person" constrained="true"/>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table Address ( personId bigint not null primary key )
```



One-to-Many (join table)

```
<class name="Person">
  <id name="id" column="personId">
    <generator class="native"/>
  </id>
  <set name="addresses" table="PersonAddress">
    <key column="personId"/>
    <many-to-many column="addressId"
      unique="true"
      class="Address"/>
  </set>
</class>
```

```
<class name="Address">
  <id name="id" column="addressId">
    <generator class="native"/>
  </id>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null, addressId bigint not null primary key )
create table Address ( addressId bigint not null primary key )
```



Many-to-One (join table)

```
<class name="Person">
  <id name="id" column="personId">
    <generator class="native"/>
  </id>
  <join table="PersonAddress"
    optional="true">
    <key column="personId" unique="true"/>
    <many-to-one name="address"
      column="addressId"
      not-null="true"/>
  </join>
</class>

<class name="Address">
  <id name="id" column="addressId">
    <generator class="native"/>
  </id>
</class>

create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null primary key, addressId bigint not null )
create table Address ( addressId bigint not null primary key
```



Many-to-Many

```
<class name="Person">
  <id name="id" column="personId">
    <generator class="native"/>
  </id>
  <set name="addresses" table="PersonAddress">
    <key column="personId"/>
    <many-to-many column="addressId"
      class="Address"/>
  </set>
</class>
```

```
<class name="Address">
  <id name="id" column="addressId">
    <generator class="native"/>
  </id>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null, addressId bigint not null, primary key
  (personId, addressId) )
create table Address ( addressId bigint not null primary key )
```



Bi

One-to-Many / Many-to-One

```
<class name="Person">
  <id name="id" column="personId">
    <generator class="native"/>
  </id>
  <many-to-one name="address"
    column="addressId"
    not-null="true"/>
</class>
```

```
<class name="Address">
  <id name="id" column="addressId">
    <generator class="native"/>
  </id>
  <set name="people" inverse="true">
    <key column="addressId"/>
    <one-to-many class="Person"/>
  </set>
</class>
```

```
create table Person ( personId bigint not null primary key, addressId bigint not null )
create table Address ( addressId bigint not null primary key )
```



Bi One-to-One foreign key

```
<class name="Person">
  <id name="id" column="personId">
    <generator class="native"/>
  </id>
  <many-to-one name="address"
    column="addressId"
    unique="true"
    not-null="true"/>
</class>
```

```
<class name="Address">
  <id name="id" column="addressId">
    <generator class="native"/>
  </id>
  <one-to-one name="person"
    property-ref="address"/>
</class>
```

```
create table Person ( personId bigint not null primary key, addressId bigint not null unique )
create table Address ( addressId bigint not null primary key )
```




Bi One-To-One Primary shared key

```
<class name="Person">
  <id name="id" column="personId">
    <generator class="native"/>
  </id>
  <one-to-one name="address"/>
</class>

<class name="Address">
  <id name="id" column="personId">
    <generator class="foreign">
      <param name="property">person</param>
    </generator>
  </id>
  <one-to-one name="person"
    constrained="true"/>
</class>

create table Person ( personId bigint not null primary key )
create table Address ( personId bigint not null primary key )
```



Bi One-to-Many/Many-to-One Join tables

```
<class name="Person">
  <id name="id" column="personId">
    <generator class="native"/>
  </id>
  <set name="addresses"
    table="PersonAddress">
    <key column="personId"/>
    <many-to-many column="addressId"
      unique="true"
      class="Address"/>
  </set>
</class>

<class name="Address">
  <id name="id" column="addressId">
    <generator class="native"/>
  </id>
  <join table="PersonAddress"
    inverse="true"
    optional="true">
    <key column="addressId"/>
    <many-to-one name="person"
      column="personId"
      not-null="true"/>
  </join>
</class>

create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null, addressId bigint not null primary key )
create table Address ( addressId bigint not null primary key )
```



Bi Many-to-Many

```
<class name="Person">
  <id name="id" column="personId">
    <generator class="native"/>
  </id>
  <set name="addresses" table="PersonAddress">
    <key column="personId"/>
    <many-to-many column="addressId"
      class="Address"/>
  </set>
</class>

<class name="Address">
  <id name="id" column="addressId">
    <generator class="native"/>
  </id>
  <set name="people" inverse="true" table="PersonAddress">
    <key column="addressId"/>
    <many-to-many column="personId"
      class="Person"/>
  </set>
</class>

create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null, addressId bigint not null, primary key (personId, addressId) )
create table Address ( addressId bigint not null primary key )
```



Composition

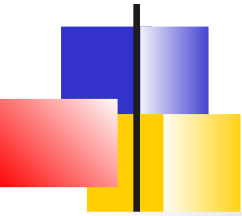
- A composition relation between 2 objects means that the life cycle of one object follow its parent
- With Hibernate it can be implemented :
 - With 2 distinct entities
 - Each have a primary key and a primary table
 - With embeddeed objects
 - The component object has no primary key and is generally stored in the same table as its enclosing entity

Transitive persistence cascade attribute

- ***save-update*** : propage la sauvegarde (*save()* et *update()*)
- ***delete*** : propage la suppression aux exemplaires.
- ***delete-orphan*** : supprime aussi les exemplaires que l'on enlève du DVD.

```
<set name="exemplaires"  cascade="save-update,delete-orphan" >  
    <key column="IDItem"/>  
    <one-to-many class="Exemplaire"/>  
</set>
```

Transitive persistence cascade attribute



For each persistence operation, there is a cascade attribute :

- *create* → `persist()`
- *merge* → `merge()`
- *save-update* → `saveOrUpdate()`
- *delete* → `delete()`
- *lock* → `lock()`
- *refresh* → `refresh()`
- *evict* → `evict()`
- *replicate* → `replicate()`
- *all*
- *all-delete-orphan*

Persistence Transitive JPA



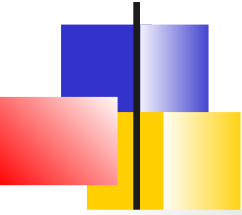
@OneToOne(cascade={CascadeType.ALL})

– *ALL* means *persist, merge, remove, refresh, ...*

The equivalent to *delete-orphan*, with JPA 2.0 is
orphanRemoval=true

**@OneToOne(cascade={CascadeType.ALL},orphan
Removal=true)**

Strict composition mapping



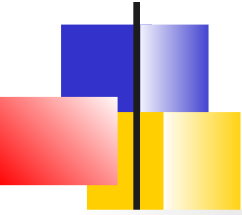
```
<hibernate-mapping package="com.plb.etechno.jl2.exemple.metier">
  <class name="Adherent" table="Tadherent">
    <id name="id" column="id" access="field">
      <generator class="native"/>
    </id>
    <property name="nom" column="nom"/>
    <property name="prenom" column="prenom"/>

    <component name="telephone" class="Telephone">
      <property name="numero" type="string" column="tel"/>
    </component>

    <set name="locations" cascade="all-delete-orphan"
      inverse="true" lazy="false">
      <key column="IDAdh"/>
      <one-to-many class="Location"/>
    </set>

  </class>
</hibernate-mapping>
```


Strict composition JPA



```
@Embeddable public class Adresse implements Serializable
private String voie;
private String codePostal;
private String ville;
public Adresse () {}
```

```
@Column(name="VOIE")
```

```
public String getVoie() { return voie;}
public void setVoie(String voie) { this.voie=voie; }
```

```
@Column(name="CODE")
```

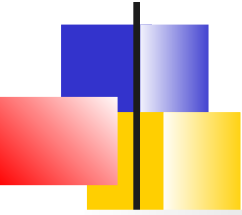
```
public long getCodePostal() { return codePostal; }
public void setCodePostal(String code) { codePostal=code; }
```

```
@Column(name="VILLE")
```

```
public long getVille() { return ville; }
public void setVille(String ville) {ville=ville; }
```

```
}
```

Strict Composition JPA



```
@Entity public class Client implements Serializable {
private ClientPK pk;
private String nom;
private Adresse adresse;

@Id @GeneratedValue
public ClientPK getPk() { return pk; }
public void setPk(ClientPK pk) { this.pk = pk; }

public String getNom() { return pk.getNom(); }
public void setNom(String nom) { pk.setNom(nom); }

@Embedded
@AttributeOverride(name="codePostal",
column=@Column(name="CODE_POSTAL"))
public Adresse getAdresse() { return adresse; }
public void setAdresse(Adresse ad) { adresse=ad; }
}
```



Loading association

Lazy and Fetch attributes
LazyInitializationException



Loading associations

Lazy and Fetch attributes

- 2 aspects to consider when loading associations :
 - **When** it is loaded: *lazy*
 - **How** it is loaded: *fetch*
- These aspects are configured via annotations or *hbm* file
- The default behaviour can be overridden via
 - HQL or Criteria queries,
 - programmatically
 - or usage of configuration profile

When it is loaded

Lazy attribute

- Configuration
 - XML Configuration defines the lazy boolean attribute
 - JPA defines the fetch attribute of @Basic and Association annotations which may be equal to :
 - ***FetchType.LAZY*** ou ***FetchType.EAGER***.
- Default Values
 - *-to-Many : Lazy
 - *-toOne, Basic : Eager
- Basic
 - Can optimize performance of an application
 - Needs instrumentation of classes during build



How it is loaded fetch attribute or @Fetch

- 4 distinct strategies :
 - **Join** : Associated entities are loaded via a single SELECT with an OUTER JOIN.
Number of associations are limited via configuration
It disables Lazy-loading
 - **Select** : Separated *select* to load the association for each parent in the session either lazily either eagerly. (N+1 Strategy)
 - **Batch** : Optimization of the select mode, group several parent entities in one select
 - **SubSelect** : Use a second separated *select* to load the association somehow the number of parents in the session

LazyInitializationException



With lazy associations, developers may face
LazyInitializationException

Which means, that some code tried to access an unloaded
association outside the scope of a session

```
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") ;
```



Solutions to *LazyInitializationException*

To solve *LazyInitializationException*, several options :

Change the configuration *lazy* à *false*
=> Generally a bad idea !!

Overriding lazy configuration for the specific usecase

- Access the association during the session
- Perform a Query specifying eager fetching
- Call *Hibernate.initialize()*
- Use Fetch profiles

JOIN FETCH



JOIN FETCH HQL queries allow the immediate loading of association which has a *LAZY configuration*

- It precises also the use of a join to have a single query

```
s = sessions.openSession();
Transaction tx = s.beginTransaction() ;
// Immediate loading with a join
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");
```



Sample Criteria

```
CriteriaBuilder criteriaBuilder =
    entityManager.getCriteriaBuilder();

CriteriaQuery<Employee> query =
    criteriaBuilder.createQuery(Employee.class);

Root<Employee> employee = query.from(Employee.class);

employee.fetch(Employee_.tasks, JoinType.INNER);

query.select(employee)

    .distinct(true);

TypedQuery<Employee> typedQuery =
    entityManager.createQuery(query);

List<Employee> resultList = typedQuery.getResultList();
```



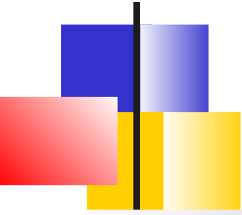
Example

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() ;
// Force the loading of the collection
Hibernate.initialize(u.getPermissions()) ;
tx.commit();
s.close() ;

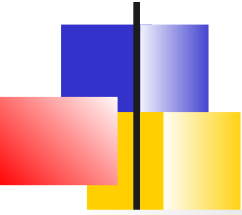
Integer accessLevel = (Integer) permissions.get("accounts") ;
```

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>
```



Example

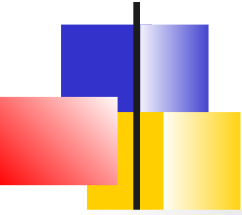
Activation of a profile

```
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") ;
```

Web MVC Application

Open Session in View



In some MVC frameworks, the pattern « **Open Session in View** » allow to avoid *LazyInitializationException*

A filter is used to Open and Close the Hibernate Session which is opened during all the request

View Templates (JSP, JSF, Thymeleaf, ...) can access association even if they have not been loaded



2nd Level Cache

Entity cache configuration
Collection and query cache



2nd level cache

An Hibernate session is a cache which has a short lifetime and which is associated only to its thread

Hibernate allow to configure a 2nd level cache which is shared between all threads. This cache has a long lifetime

Cache may contain :

- Entities

- Collections

- Query results



Configuration

Configuration consist of :

- Specifying the cache provider implementation
- What must be cached :
 - At global level, (for all entities, ...)
Not recommended to change default values
 - At entity, association or query level
- Defines cache regions which can have different eviction/management strategies



Cache provider

The configuration of the cache provider is generally performed as follow :

Activate Caching

```
hibernate.cache.use_second_level_cache=true
```

Specify implementation

```
hibernate.cache.region.factory_class=org.hibernate.cache.ehcache.EhCacheRegionFactory
```

Hibernate supports :

- JCache : Interface standard Java, an implementation must be provided (see *HazelCast* for example)
- Legacy solutions
 - EHCache
 - InfiniSpan



Entity Configuration

To configure caching, use the **<cache>** element or the **@Cache** annotation

```
@Entity
@Cache(usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
public class Forest { ... }
----
```

At first loading, Hibernate store the entity in the cache (its database representation not the full object).

If it needs to load the entity again, it will use the cache

If it is aware of updates, it will invalidate the region associated to the entity



@Cache attributes

@Cache has 3 attributes :

usage (required) : Control the concurrency access of the cache

region (optional) : The cache region associated with the entity, if we want to rename the default one

include (optional) : all (default) or non-lazy (lazy properties are not cached)



Concurrency strategy

usage attribute set the isolation level of transaction of R/W operations.

Implementations provide default values, so configuration is rarely required :

NONE :

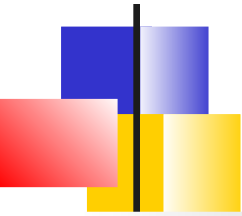
READ_ONLY : Best choice if only read operations are required

READ_WRITE : Application performs write operations. Tx has no repeatble read

NONSTRICT_READ_WRITE : Simultaneous Write operation on the same data are rare

TRANSACTIONAL : No concurrency problems but low throughput

Session interactions with the 2nd level cache



The **CacheMode** attribute of a session specifies how this session interacts with the 2nd level cache :

CacheMode.NORMAL (default) : Session reads and writes to the cache

CacheMode.GET : Session reads but do not write. An update invalidate the cache

CacheMode.PUT : Session do not read from cache, but write after loading from database

CacheMode.IGNORE : Never interacts except for invalidating data



Eviction

The *SessionFactory* interface allows to evict an entity or a collection from the cache :

```
// An entity
sessionFactory.evict(Cat.class, catId);
// All entities of one type
sessionFactory.evict(Cat.class);
// A collection
sessionFactory.evictCollection("Cat.kittens", catId);
// All the collections of an entity type
sessionFactory.evictCollection("Cat.kittens");
```




Collection Configuration

@Cache annotation can added to the collection property.

```
@OneToMany(cascade=CascadeType.ALL)
@Cache(usage =
    CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
public SortedSet<Ticket> getTickets() {
    return tickets;
}
```

The collection cache entry will store the entity identifiers only.



Query caching

Query results can also be cached even if use cases are rare

Query caching is disabled by default, to enable :

hibernate.cache.use_query_cache true

A query can be then be cached :

➤ programmatically :

```
org.hibernate.Query.setCacheable(true)
```

➤ Or via annotation :

```
@NamedQuery(name = "products.getProducts",  
            query = "SELECT product FROM Product product",  
            hints = { @QueryHint(name = "org.hibernate.cacheable", value =  
"true"), @QueryHint(name="org.hibernate.cacheMode", value="NORMAL") })
```



Transaction management

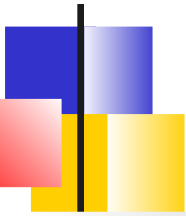
Contextual session pattern
Strategies of Concurrency control
User transactions



Introduction

Transaction can have different meanings

- DB transaction. In Java : JDBC or JTA
- Logical transaction related to a persistent context
- User-transaction, i.e A unit of work

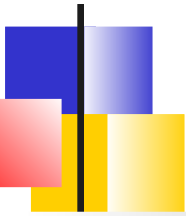


DB Transaction configuration

By default, the transaction coordinator used is :

- If JPA
The one indicated with the transaction type for the persistence unit.
- If No-JPA : JDBC

This default behaviour can be overridden with the `hibernate.transaction.coordinator_class` property



Contextual Session patterns

One-session-per-operation : Anti-pattern

One-session-per-request : Default pattern
=> session is then bound to a DB transaction

- Either manually
- Either via framework (EJB, Spring). Can use `SessionFactory.getCurrentSession()` to retrieve the session associated to the thread

One-session-per-conversation : Pattern to use for user transaction

One-session-per-application : Anti-Pattern



Concurrency control

Hibernate uses JDBC connexion or JTA ressources without adding specific locking

Thanks to the session (1st level cache), Hibernate provides a « repeatable reads » isolation level

2 other mechanisms are provided for concurrency control :

- Optimistic strategy via Versionning or timestamp
- Pessimistic strategies vis SELECT FOR UPDATE



Strategies for concurrency control

- Optimistic-locking
 - Use of time-stamp or versionnning.
 - Conflicts are resolved when committing
- Pessimistic-locking
 - Immediate locking.
 - Conflicts are avoided



Lock and Isolation level

Typically, you only need to specify an isolation level for the JDBC connections and let the database handle locking issues.

Pessimistic locking is only used when you do need to obtain exclusive pessimistic locks or re-obtain locks at the start of a new transaction.



Pessimistic vs Optimistic

- Pessimistic-locking

- Conflicts are frequent and dangerous.
- Reactivity is not a concern
- Concurrent transactions involved separate domain models
- Conflicts resolution are automatic

- Optimistic-locking

- Conflicts are rare and not critical
- High reactivity
- Conflict resolution must be managed by users.



Optimistic

- Add a technical column
 - Version : Integer incremented at each update
 - Timestamp : Last modification timestamp.
- Each update contains the restriction
WHERE version = X
- If a conflict is detected, the user is warned



Set up

- For the class Theme
 - Update the table with the technical column
 - Add a new property to the class
 - Declare it as version property

```
<class name="Theme" table="TTheme">
  <id name="id" column="id" access="field">
    <generator class="native"/>
  </id>
  <version name="version" column="version"/>
  <property name="label" column="label"/>
  ...
</class>
```

```
public class Theme {
    private Long id;
    private String label;
    private int version;
    ....
}
```

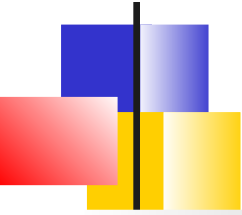


Set up with JPA

- For class Theme
 - Add the column
 - Add the property
 - Annotate with *@Version*

```
@Entity
public class Theme {
    @Id @GeneratedValue
    private Long id;
    private String label ;
    @Version
    private int version;
    ....
}
```

Optimistic Usage



```
Session s1 = DBHelper.getFactory().openSession();
Session s2 = DBHelper.getFactory().openSession();
Transaction tx1 = s1.beginTransaction();
Transaction tx2 = s2.beginTransaction();
Theme info1 = (Theme)s1.get(Theme.class, new Long(3));
Theme info2 = (Theme)s2.get(Theme.class, new Long(3));
System.out.println("égalité en base ? : " +(info1.getId().equals(info2.getId())));
info1.setLabel("INFORMATIQUE");
info2.setLabel("INFO");
tx2.commit();
tx1.commit();
```



```
Equality in base ? :true
Hibernate: update TTheme set version=?, label=? where id=? and version=?
Hibernate: update TTheme set version=?, label=? where id=? and version=?
SEVERE: Could not synchronize database state with session
org.hibernate.StaleObjectStateException: Row was updated or deleted by another transaction
(or unsaved-value mapping was incorrect): [com.tsystems.etechno.j12.exemple.metier.Theme#3]
```

Pessimistic strategy

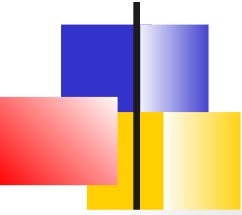
explicit locking

- Session API which use LockMode
 - get(), load(), refresh()
 - lock(), setLockMode()

```
Session s = DBHelper.getFactory().openSession();  
Transaction tx = s.beginTransaction();  
Theme info = (Theme)s.get(Theme.class, new Long(3),LockMode.UPGRADE);  
info.setLabel("INFORMATIQUE");  
tx1.commit();
```

Pessimistic strategy

Lock types

- 
- ***LockMode.NONE*** : No Lock (default).
 - ***LockMode.READ*** : Optimistic, check the version.
 - ***LockMode.WRITE*** : Force the version to be incremented, even if no update
 - ***LockMode.PESSIMISTIC_FORCE_INCREMENT*** : Pessimistic lock with version incremented
 - ***LockMode.PESSIMISTIC_READ*** : Pessimistic shared lock (if database support)
 - ***LockMode.UPGRADE*** : Wait for a lock. May timeout
 - ***LockMode.UPGRADE_NOWAIT*** : Fail-fast, if lock cannot be obtained



Sample

```
try {  
    EntityManager entityManager = getEntityManagerWithOpenTransaction();  
    PessimisticLockingStudent resultStudent =  
entityManager.find(PessimisticLockingStudent.class, 1L);  
    entityManager.refresh(resultStudent, LockModeType.PESSIMISTIC_READ);  
  
    EntityManager entityManager2 = getEntityManagerWithOpenTransaction();  
    PessimisticLockingStudent resultStudent2 =  
entityManager2.find(PessimisticLockingStudent.class, 1L);  
    resultStudent2.setName("Change");  
    entityManager2.persist(resultStudent2);  
    entityManager2.getTransaction()  
        .commit();  
  
    entityManager.close();  
    entityManager2.close();  
} catch (Exception e) {  
    // Which line throws the exception ?  
    Assert.isTrue(e instanceof PessimisticLockException);  
}
```

User Transaction

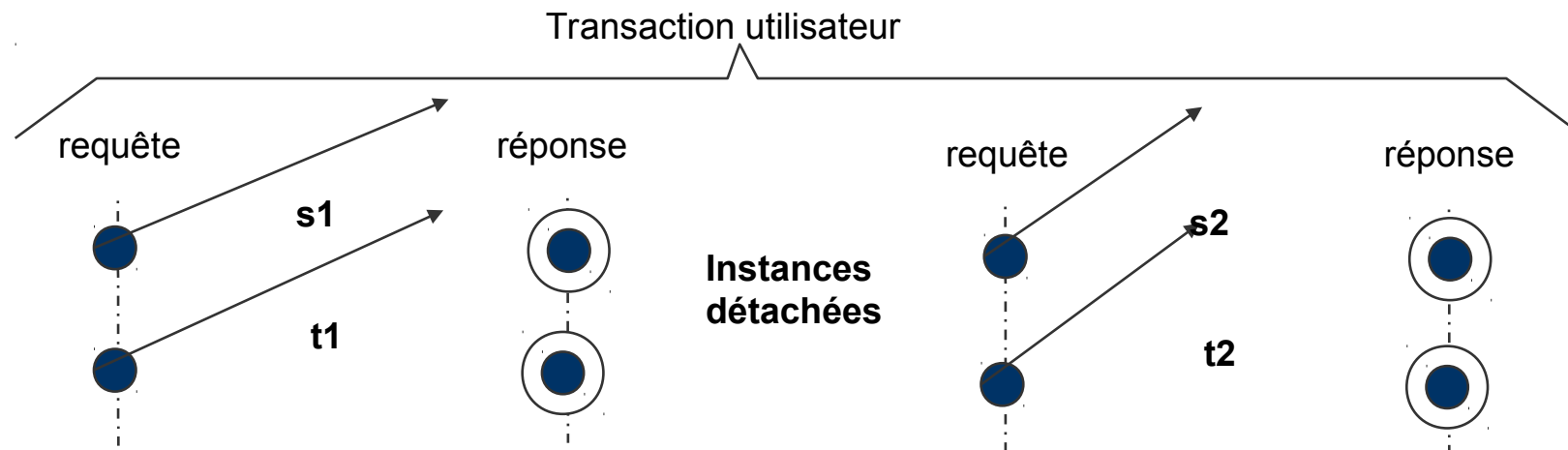
Définitions

- Database Transaction
 - Managed by database.
- User Transaction
 - User-machine interaction.
 - Similar to use case.
 - Cannot lock resources
 - A transaction from the user point of view.

User transaction

One session for each DB transaction

- Detached instances
 - 1 : Getting persistent objects for one session.
 - 2 : Entities are detached during user manipulation
 - 3 : Entities are reattached to another session for update





Sample

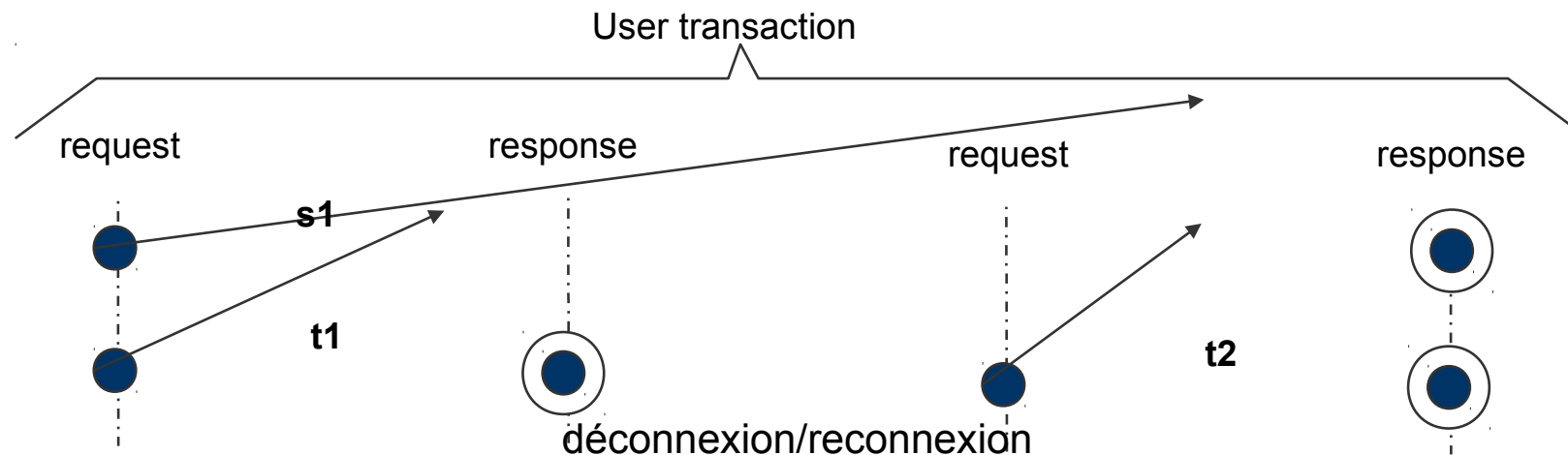
```
public class UseCaseManager {
    MyEntity myEntity;
    public void request1(long id) {
        Session sess = factory.openSession() ; Transaction tx = null;
        try {
            tx = sess.beginTransaction();
            this.myEntity = sess.load(MyEntity.class,id)
            tx.commit(); // Flush of the session and DB update
        } catch (RuntimeException e) {    if (tx != null) tx.rollback() ; throw e; // or display error message
        } finally { sess.close() ; }
    }

    public void request2(String newValue) {
        Session sess = factory.openSession() ; Transaction tx = null;
        try {
            tx = sess.beginTransaction();
            myEntity = sess.load(MyEntity.class,myEntity.getId()); // Re-attachment
            myEntity.setValue(newValue);
            tx.commit(); // Flush de la session et Mise à jour BD
        } catch (RuntimeException e) {
            if (tx != null) tx.rollback() ; throw e; // or display error message
        } finally { sess.close() ; }
    }
}
```

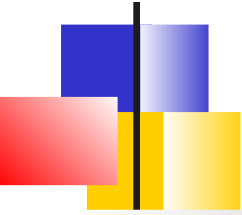
User Transaction

A single session for a user transaction

- Session is opened with *FlushMode.MANUAL*
- Session is closed at the last DB transaction
- `org.hibernate.Transaction.commit()` : Release the JDBC connexion
- Entities stay attached
- Last request call explicitly `n.flush()`
- An optimistic strategy can be applied



Example Session Hibernate



```
public class UseCaseManager {
    MyEntity myEntity ; Session sess;

    public void request1(long id, String newValue1) {
        Session sess = factory.openSession() ; Transaction tx = null ;
        sess.setFlushMode(FlushMode.MANUAL) ;
        try {
            tx = sess.getTransaction() ; tx.begin();
            this.myEntity = sess.load(MyEntity.class,id)
            myEntity.setValue1(newValue1) ;
            tx.commit(); // Release connexion
        } catch (RuntimeException e) { if (tx != null) tx.rollback() ; throw e;}
        // session is not closed
    }

    // May throw an OptimisticLockException if versionning is implemented
    public void request2(String newValue2) {
        try {
            tx = sess.beginTransaction() ; // Obtient une nouvelle connexion JDBC
            // Entity is already attached
            myEntity.setValue2(newValue2) ;
            sess.flush() ;
            tx.commit();
        } catch (RuntimeException e) { if (tx != null) tx.rollback() ; throw e;
        } finally { sess.close() ; }
    }
}
```



Miscellaneous

Inheritance mapping
Interceptors and events
Batch



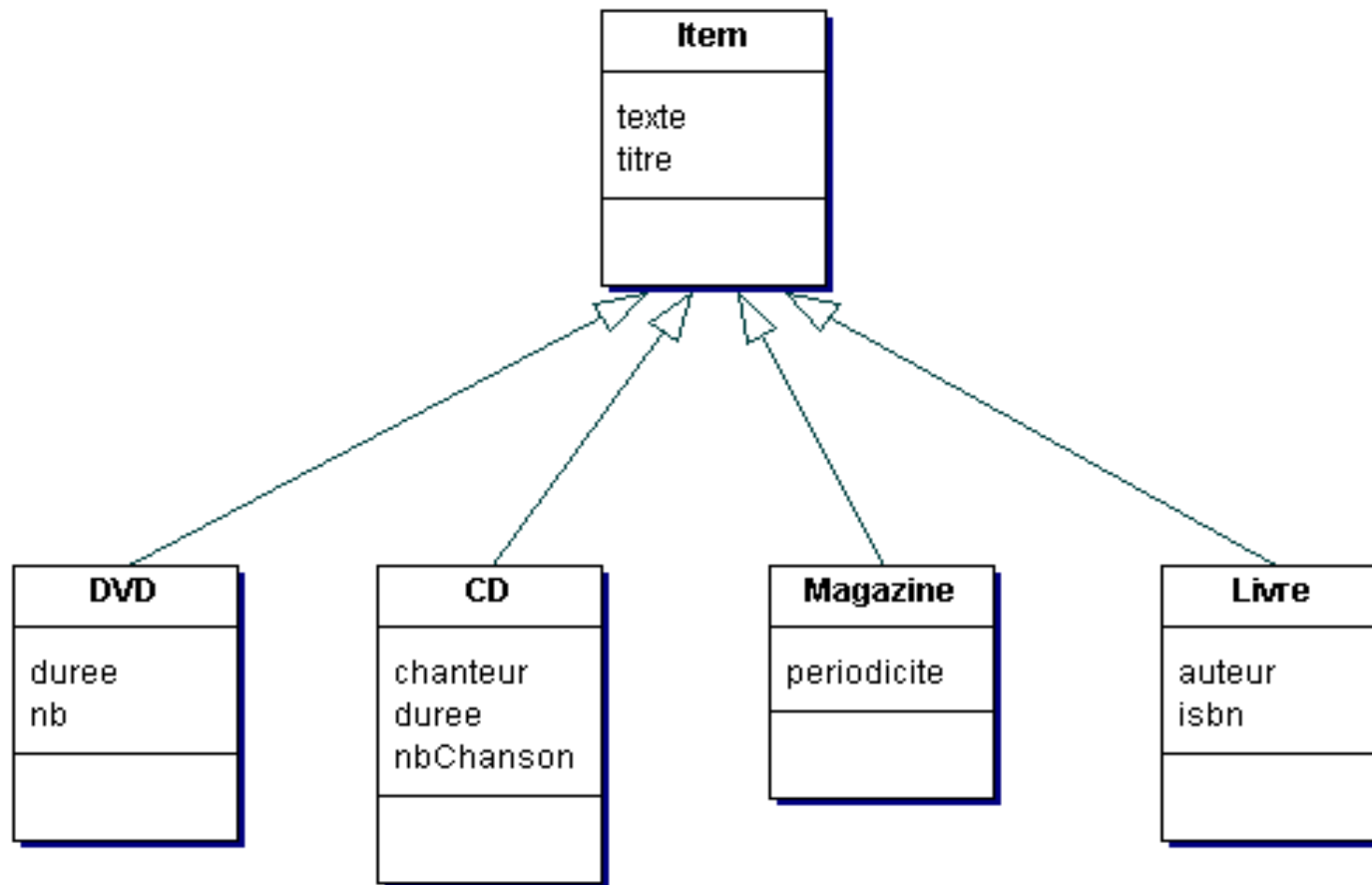
Inheritance Mapping



3 approaches

- One table for each concrete class
 - No inheritance relation in the relational model
- A single table for the hierarchy of classes
 - Use of a discriminator column
- One table for each class
 - Use of foreign key (or shared primary key)

Sample



One table by concrete class

- Each table of the concrete classes includes the common attributes .

Table DvD

id	Titre	Texte	Durée
1	Troie	...	2h20
2	Paris	...	1h40

Table CD

id	Titre	Texte	Chanteur	Nb
1	Live U2	...	U2	22
2	No rest	..	Rivers	10

Mapping

```
<hibernate-mapping package="com.plb.etechno.j12.exemple.metier">
  <class name="Item">
    <id name="id" column="id" access="field">
      <generator class="native"/>
    </id>
    <property name="titre" column="titre"/>
    <property name="texte" column="texte"/>
    <set name="exemplaires" lazy="false" >
      <key column="IDItem"/>
      <one-to-many class="Exemplaire"/>
    </set>
    <union-subclass name="DVD" table="TDvD">
      <property name="duree" column="duree"/>
      <property name="nbDvd" column="nb"/>
    </union-subclass>
    <union-subclass name="CD" table="TCd">
      <property name="chanteur" column="chant"/>
      <property name="nbDvd" column="nb"/>
    </union-subclass>
  </class>
</hibernate-mapping>
```

Partie Item

Partie DvD

Partie CD



One table for the whole hierarchy

Table Item

id	titre	texte	duree	nbDvd	chanteur	auteur	isbn	periodic.	type	
01	Troie	Fil.	2h20	2	null	null	null	null	DvD	
02	Supe	Dis.	1h04	null	SuperT.	null	null	null	CD	
03	Tele7	Je..	null	null	null	null	null	hebdo	Mag	

Mapping

```
<hibernate-mapping package="com.plb.etechno.j12.exemple.metier">
  <class name="Item" table="TItem" discriminator-value="IT">
    <id name="id" column="ID"><generator class="native"/></id>
    <discriminator column="discriminator" type="string"/>
    <property name="titre" column="titre" />
    <property name="texte" column="texte" />
    <set name="exemplaires" inverse="true" cascade="all-delete-orphan" lazy="false" >
      <key column="IDItem"/>
      <one-to-many class="Exemplaire"/>
    </set>

    <subclass name="DvD" discriminator-value="DVD">
      <property name="nbDvd" column="nbdvd"/>
      <property name="duree" column="duree"/>
    </subclass>

    <subclass name="CD" discriminator-value="CD">
      <property name="chanteur" column="chanteur"/>
    </subclass>

  </class>
</hibernate-mapping>
```

Item

DvD

CD

One table for each class

- A shared primary key is applicable on all the tables

table Item

id	titre	texte
01	Troie	Film de...
02	Collatéral	Thriller...
03	Crisis	Premier...

table TDvD

IdDVD	duree	nbDvD
001	2h20	2
002	2h	1

table TCD

idCD	chanteur
003	Supertramp

Mapping

```
<hibernate-mapping package="com.plb.etechno.j12.exemple.metier">
  <class name="Item" table="TItem" >
    <id name="id" column="ID"><generator class="native"/></id>
    <property name="titre" column="titre" />
    <property name="texte" column="texte" />
    <set name="exemplaires" inverse="true" cascade="all-delete-orphan" lazy="
false" >
      <key column="IDItem"/>
      <one-to-many class="Exemplaire"/>
    </set>

    <joined-subclass name="Dvd" table="TDVD">
      <key column="IDDVD"/>
      <property name="nbDvd" column="nbdvd"/>
      <property name="duree" column="duree"/>
    </joined-subclass>

    <joined-subclass name="CD" table="TCD">
      <key column="IDCD"/>
      <property name="chanteur" column="chanteur"/>
    </joined-subclass>

  </class>
</hibernate-mapping>
```


JPA



With JPA, the strategy is specified on the base class

@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)

@Inheritance(strategy=InheritanceType.SINGLE_TABLE)

@Inheritance(strategy=InheritanceType.JOINED)

With SINGLE_TABLE, the discriminant column is indicated via

@DiscriminatorColumn

Each subclass is annotated with ***@DiscriminatorValue*** which
precises the value which identifies it



Polymorphic queries

```
public List<Item> getAllItems(){  
    List<Item> ret = new ArrayList<Item>();  
    Session session = DBHelper.getFactory().openSession();  
    Transaction tx = null;  
    try {  
        tx = session.beginTransaction();  
        Query hqlQuery = session.createQuery("from Item");  
        ret = (List<Item>)hqlQuery.list();  
        tx.commit();  
    }  
    ...  
}
```

CU : Consulter la liste des items

- 1 - L'utilisateur demande à consulter la liste complète des items
- 2 - Le système affiche la liste des items
 - 1) Troie nb Ex -> 3
 - 2) Collateral nb Ex -> 0
 - 3) Crisis nb Ex -> 0
- 3 - L'utilisateur sélectionne un item [le 3 ème]
- 4 - Le système affiche les détails :
Crisis chanteur : SuperTramp



Comparison

SINGLE_TABLE performs the best in terms of executed SQL statements. However, you cannot use NOT NULL constraints on the column-level. You can still use triggers and rules to enforce such constraints, but it's not as straightforward.

JOINED addresses the data integrity concerns because every subclass is associated with a different table. Polymorphic queries or *@OneToMany* base class associations don't perform very well with this strategy. However, polymorphic *@ManyToOne* associations are fine, and they can provide a lot of value.

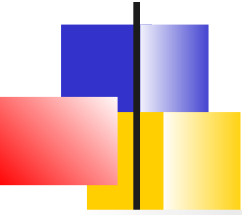
TABLE_PER_CLASS should be avoided since it does not render efficient SQL statements.



Interceptors and events

Interceptors and events

Introduction

- 
-
- Application can listen to session events .
 - The ***Interceptor*** interface can implement callback methods called when entities are inserted, updated or deleted.
 - ***Listeners*** are configured via XML or annotations.



Interceptors

Interceptors may be set :

- On the *session* :

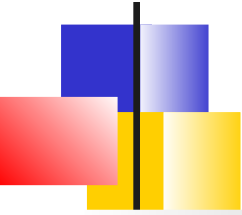
```
Session session =  
sf.openSession( new AuditInterceptor() );
```

- On the *session-factory* (applicable for all sessions) :

```
new Configuration().  
setInterceptor( new AuditInterceptor() );
```

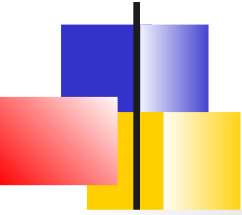
Interceptors

Callback methods



void afterTransactionBegin(Transaction tx) : Beginning of a transaction
void afterTransactionCompletion(Transaction tx) : After a commit or rollback
void beforeTransactionCompletion(Transaction tx) : Before a commit
void onCollectionRecreate(Object collection, Serializable key) : Loading a collection
void onCollectionRemove(Object collection, Serializable key) : Removing a collection
void onCollectionUpdate(Object collection, Serializable key) : Update of a collection
void onDelete(Object entity, Serializable id, Object[] state, String[] propertyNames, Type[] types) : Deletion of an entity
boolean onFlushDirty(...) : Entity is detected dirty when flushing
boolean onLoad(...) : Loading an entity
String onPrepareStatement(String sql) : An SQL statement is built
boolean onSave(..) : Insertion of an entity
void postFlush(Iterator entities) : Before a flush
void preFlush(Iterator entities) : After a flush

Interceptors Example



```
public class AuditInterceptor extends EmptyInterceptor {
    private int updates, creates ;

    public boolean onFlushDirty(Object entity, Serializable id, Object[] newState, Object[]
oldState,String[] propertyNames,Type[] types) {
        updates++;
        for ( int i=0; i < propertyNames.length; i++ ) {
            if ( "lastUpdateTimestamp".equals( propertyNames[i] ) ) { newState[i] = new
Date() ; return true ; }
        }
    public boolean onSave(Object entity, Serializable id, Object[] state, String[]
propertyNames,Type[] types) {
        creates++;
        for ( int i=0; i<propertyNames.length; i++ ) {
            if ( "createTimestamp".equals( propertyNames[i] ) ) { state[i] = new Date() ;
return true ; }
        }
    }
    public void afterTransactionCompletion(Transaction tx) {
        if ( tx.wasCommitted() ) { System.out.println("Creations: " + creates + ", Updates: " +
updates) ; }
        updates=0; creates=0 ;
    }
}
```




Events

org.hibernate.event

All the methods of the *Session* interface are associated to events

When a method is called, Hibernate broadcast an event to all registered listeners. Some Listeners are configured by default

A listener is a singleton and must be thread safe

Registration is made by

- Configuration (XML or annotation)
- Programmatically, via the *Configuration* object



Example

```
<hibernate-configuration>
```

```
    <session-factory>
```

```
        ...
```

```
        <event type="load">
```

```
            <listener class="com.eg.MyLoadListener"/>
```

```
            <listener  
class="org.hibernate.event.def.DefaultLoadEventListener"/>
```

```
        </event>
```

```
    </session-factory>
```

```
</hibernate-configuration>
```



Example

```
public class MyLoadListener implements LoadEventListener {  
  
    // this is the single method defined by the LoadEventListener interface  
  
    public void onLoad(LoadEvent event, LoadEventListener.LoadType loadType)  
  
        throws HibernateException {  
  
        if ( !MySecurity.isAuthorized( event.getEntityClassName(),  
event.getEntityId() ) ) {  
  
            throw MySecurityException("Unauthorized access");  
  
        }  
  
    }  
  
}
```



JPA Call-back method

These methods may be defined in the entity class or in a listener class which may be specified in the *@EntityListener* class annotation

Available annotations are:

@PrePersist : Before persist

@PostPersist : After DB insertion

@PreUpdate : Before merge

@PostUpdate : After update of entity

@PreRemove : Before call to remove

@PostRemove : After DB deletion

@PostLoad : After loading



Example

```
@Entity
public class Client implements
    java.io.Serializable{
    .
    .
    .
@PostPersist
    public void insertion() { ... }
@PostLoad
    public void chargement() { ... }
}
```



Example : *EntityListener*

```
public class Logging {
```

```
    @PostPersist
```

```
    public void insertion(Object entite) {
```

```
        ...
```

```
    }
```

```
    @PostLoad
```

```
    public void chargement(Object entite)
```

```
        ...
```

```
    }
```

```
}
```

```
-----
```

```
@Entity
```

```
@EntityListeners(Logging.class)
```

```
public class Personne implements java.io.Serializable {
```



Default *EntityListeners*

Default entity listeners can be specified in *orm.xml*

→ they are applied for all entities

```
<entity-mappings>
```

```
...
```

```
  <entity-listeners>
```

```
    <entity-listener class="com.sample.Logging">
```

```
      <post-persist method-name="insertion"/>
```

```
      <post-load method-name="chargement"/>
```

```
    </entity-listener>
```

```
    <entity-listener class="com.sample.Perform"/>
```

```
  </entity-listeners>
```

```
...
```

```
</entity-mappings>
```

Lab : Listener



Batch



JDBC Batching

JDBC offers support for batching :
drivers will send the batched operation
to the server in one call, which can
save on network calls to the database.

hibernate.jdbc.batch_size : Maximum
number of statements Hibernate will
batch together



Session Batching

The following code may throw an *OutOfMemoryException* as the session contains all *Customer* entities

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
for ( int i=0; i<100000; i++ ) {
    Customer customer = new Customer(.....);
    session.save(customer);
}
tx.commit();
session.close();
```



Solutions to avoid *OutOfMemory*

When you make new objects persistent, employ methods *flush()* and *clear()* to the session regularly, to control the size of the first-level cache.

When you retrieve and update data, *flush()* and *clear()* the session regularly.
In addition, do not use 2nd level cache and use method *scroll()* to take advantage of server-side cursors for queries that return many rows of data.



Sample : Batch insert

```
txn.begin();

int batchSize = 25;

for ( int i = 0; i < entityCount; i++ ) {
    if ( i > 0 && i % batchSize == 0 ) {
        //flush a batch of inserts and release memory
        entityManager.flush();
        entityManager.clear();
    }

    Person Person = new Person( String.format( "Person %d", i ) );
    entityManager.persist( Person );
}

txn.commit();
```



Sample : Use of *scroll()*

```
int batchSize = 25;

Session session = entityManager.unwrap( Session.class );

scrollableResults = session
    .createQuery( "select p from Person p" )
    .setCacheMode( CacheMode.IGNORE )
    .scroll( ScrollMode.FORWARD_ONLY );

int count = 0;
while ( scrollableResults.next() ) {
    Person Person = (Person) scrollableResults.get( 0 );
    processPerson(Person);
    if ( ++count % batchSize == 0 ) {
        //flush a batch of updates and release memory:
        entityManager.flush();
        entityManager.clear();
    }
}

txn.commit();
```



Monitoring & Performance

Way to monitor
Recommendations



Log

Hibernate can log :

- The SQL statement executed
- Which bind parameter values it used,
- How many records the query returned
- How long each execution took

To activate :

org.hibernate.SQL to DEBUG.

But it is often better to use tools like
datasource-proxy or *p6spy*



Statistics

By default, statistics are not collected because this add processing and memory overhead.

To enable statistics :

hibernate.generate_statistics=true

To display stats on the logs

- *org.hibernate.stat* = DEBUG
- Or use API in interceptor



Log sample

```
14:37:30,715  INFO StatisticalLoggingSessionEventListener:258 - Session
Metrics {
    48986 nanoseconds spent acquiring 1 JDBC connections;
    23326 nanoseconds spent releasing 1 JDBC connections;
    259859 nanoseconds spent preparing 1 JDBC statements;
    1092619 nanoseconds spent executing 1 JDBC statements;
    0 nanoseconds spent executing 0 JDBC batches;
    0 nanoseconds spent performing 0 L2C puts;
    0 nanoseconds spent performing 0 L2C hits;
    0 nanoseconds spent performing 0 L2C misses;
    22383767 nanoseconds spent executing 1 flushes (flushing a total of 1
entities and 1 collections);
    72779 nanoseconds spent executing 1 partial-flushes (flushing a total
of 0 entities and 0 collections)
}
```



Statistics API

An instance of the Statistics interface can be obtained by
SessionFactory.getStatistics()

It provides :

- Agregate statistics on queries, entities, collections
- SessionFactory statistics : Number of entities and collections
- Session statistics : Number of open, close, flush
- JDBC statistics : Number of PS
- Transaction statistics : Number of (succesful) transactions
- Natural Ids statistics
- 2nd Level Cache



Schema management

Although, Hibernate provides the update option for the *hibernate.hbm2ddl.auto* configuration property, this feature is not suitable for a production environment

Use an automated schema migration tool like ***Flyway*** or ***Liquibase***



Mapping

- ✓ Identifier : SEQUENCE is the best choice
- ✓ Bi-directional are generally best than uni-directional
- ✓ The *@ManyToOne* and the *@OneToOne* child-side association are best to represent a FOREIGN KEY relationship.
- ✓ It's best to map *@OneToOne* association using *@MapsId* so that the PRIMARY KEY is shared between the child and the parent entities
- ✓ For unidirectional collections, Sets are the best choice because they generate the most efficient SQL statements.
- ✓ Bidirectional associations are usually a better choice because the *@ManyToOne* side controls the association.
- ✓ The *@ManyToMany* annotation is rarely a good choice because it treats both sides as unidirectional associations. You can replace it with a LinkEntity and 2 ManyToOne relationship



Caching

Although the second-level cache can reduce transaction response time since entities are retrieved from the cache rather than from the database, there are other options to achieve the same goal, and you should consider these alternatives prior to jumping to a second-level cache layer:

- tuning the underlying database cache so that the working set fits into memory, therefore reducing Disk I/O traffic.
- optimizing database statements through JDBC batching, statement caching, indexing can reduce the average response time, therefore increasing throughput as well.
- database replication is also a very valuable option to increase read-only transaction throughput