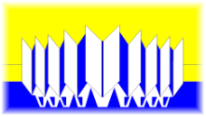
Faculté des Sciences de Tunis, Tunisie

CPOO
Framework de persistance: Hibernate



Présenté par : Nourhène ALAYA
2012-2013

Persistance des objets et bases de données relationnelles

- Majorité de bases de données relationnelles (position dominante sur le marché, théorie solide et normes reconnues)
- Nombreuses applications développées en langage de programmation orienté-objet
- Modélisation UML

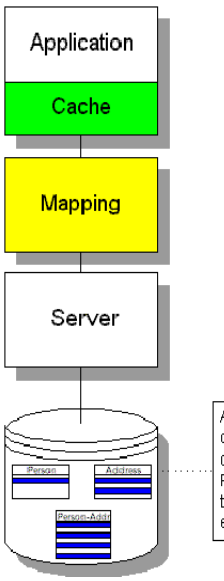
⇓

Comment effectuer la persistance des données d'une application orientée objet dans une base de données relationnelles ?

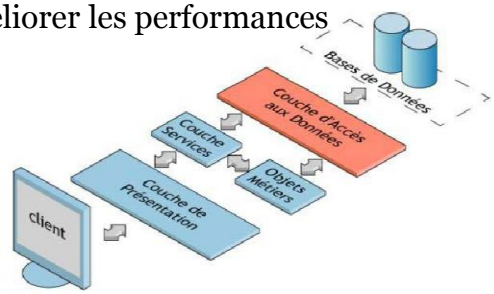
ORM : *Object/Relational Mapping*

- **La persistance**
  - Stockage, organisation et récupération des données structurées (tri, agrégation)
  - Concurrence et intégrité des données
  - Partage des données
- ORM est:
  - **Persistance automatisée et transparente d'objets métiers vers une bases de données relationnelles [BK05]**
  - Description à l'aide de **méta-données de la transformation réversible entre un modèle relationnel** et un modèle de classes [BK05, Pato5]
  - Capacité à manipuler des données stockées dans une base de données relationnelles à l'aide d'un langage de programmation orientée-objet
  - Techniques de programmation permettant de lier les bases de données relationnelles aux concepts de la programmation OO pour créer une "base de données orientées-objet virtuelle" [Wikipedia]

Couche d'accès au données
Couche de persistance



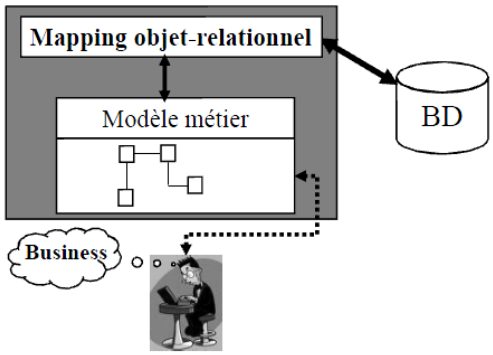
- Prise en charge de toutes les interactions entre l'application et la base de données
- Groupes de classes et de composants chargés du stockage et de la récupération des données
- Possibilité de servir de **cache** pour les objets récupérés dans la base de données pour améliorer les performances



Couche de persistance : avec correspondance objet/relationnel

- Utilisation de la couche de persistance comme un service rendant **abstraite** la représentation relationnelle indispensable au stockage final des objets
- Concentration du développeur sur les problématiques métier

Stratégie de persistance « transparente »



Exemple simple de correspondance

- Implémentation **POJO (*Plain Old Java Object*) de la classe *Departement*:**

```
public class Departement implements java.io.Serializable {
  // Fields
  private int departementId;
  private String nomDepartement;
  /** default constructor */
  public Departement() {}
  /** full constructor */
  public Departement(int departementId, String nomDepartement) {
    this.departementId = departementId;
    this.nomDepartement = nomDepartement;
  }
  // Property accessors
  ...
}
```

Relation de bases de données `Departement` :

```
CREATE TABLE Departement
(
  departement_id int4 NOT NULL,
  nom_departement varchar(25) NOT NULL
)
```

Clé primaire ??

departement_id int4	nom_departement varchar
1	MIDO
2	LSO
3	MSO
4	LANGUES

1. Identification des objets

- **Objet persistant = représentation en mémoire d'un nuplet (enregistrement)**
  Un même nuplet ne doit pas être représenté par plusieurs objets en mémoire centrale pour une même session de travail
- *Exemple :*
  - Création en mémoire d'un objet `e1` de la classe `Enseignant` (à l'occasion d'une navigation à partir d'un objet `Enseignement`) . Possibilité de retrouver le même enseignant depuis un autre objet `Enseignement` ou depuis un objet `Departement`
    - *Ne pas créer d'objet `e2` de la classe `Enseignant` en mémoire centrale indépendant de `e1`*

  =>*doit garantir l'unicité des objets en mémoire par analogie à l'unicité des enregistrements de la base de donnée.*

  ⇒ Utilisation du **cache**
  - "**Index**" des objets créés en mémoire (avec conservation de l'identité relationnelle – clé primaire)
  - Recherche dans le cache avant toute récupération dans la base.

Non correspondance :

- La correspondance entre le modèle objet et le modèle relationnel n'est pas une tâche facile
  - **30%** du coût de développement consacré à la mise en correspondance
  - Modélisation relationnelle tributaire de la théorie relationnelle
  - Modélisation orientée-objet sans définition mathématique rigoureuse ni partie théorique

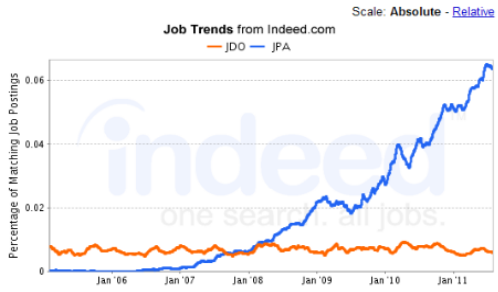
⇒**Besoin d'utiliser des outils ORM pour la réduction du code de correspondance**

Les Framework ORM

- **Normes Java :**
  - **EJB (*Entreprise Java Beans*) :**
    - Gestion de la persistance par conteneur (CMP- *Container-Managed Persistence et BMP– Beans Managed Persistence)*
    - Spécifications EJB3.0 (JSR 220 Mai 2006)
  - **JDO (*Java Data Object*) :**
    - Spécification de Sun 1999 – JDO 2.0 (JSR243 Mars 2006)
    - Abstraction du support de stockage
    - Implémentation libre : JPOX
  - **JPA (*Java Persistence API*) : *Partie des spécifications EJB 3.0 (JSR 220 en Mai 2006 – JSR*** 316 en cours) concernant la persistance des composants
- **Implémentation de JPA :**
  - **Hibernate (*JBoss*) : *Solution libre faisant partie du serveur d'appli. JBoss – version 3.3*** implémentant les spécifications JSR 220 – complète et bien documentée - plugin Eclipse - Gavin King (fondateur) membre de groupes d'expert d'EJB3
  - **TopLink (*Oracle*) : *Solution propriétaire utilisée par la serveur d'application d'Oracle***
  - **TopLink Essentials *: version libre disponible dans Netbeans 5.5 ou le serveur d'application*** (Java EE 5) *Glassfish de Sun, integrée dans le projet EclipseLink (version 1 07/2008)*
  - **OpenJPA (*Apache*), *Cayenne (Apache) ...***
- **"Comparaison " des solutions ORM :**
  - http://se.ethz.ch/projects/shinji_takasaka/master_thesis_shinji_takasaka.pdf
  - http://laurentbois.com/2006/07/29/jpa-avec-hibernate-et-toplink/
  - *http://java.dzone.com/news/hibernate-best-choice?page=2*

Pourquoi on étudiera la norme JPA ?

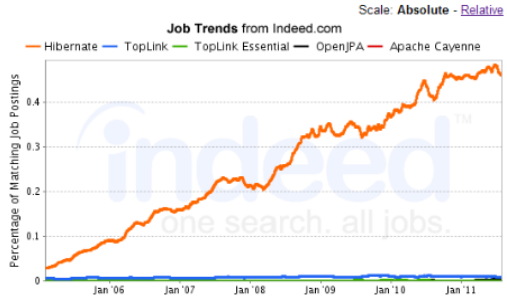
JDO, JPA Job Trends



Nb d'offres sur le site français :
- 126 pour JPA
- 35 pour JDO

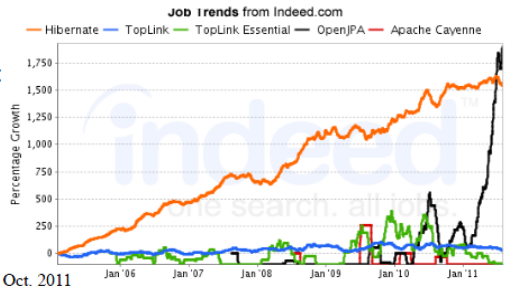
Graphiques issus de http://www.indeed.com/ - 3 Oct. 2011

Pourquoi on étudiera Hibernate ?



Nb d'offres sur le site français :

- 2041 pour Hibernate
- 31 pour TopLink
- 0 pour les autres



Graphiques issus de http://www.indeed.com/ - 3 Oct. 2011

Le Framework Hibernate

http://www.hibernate.org/

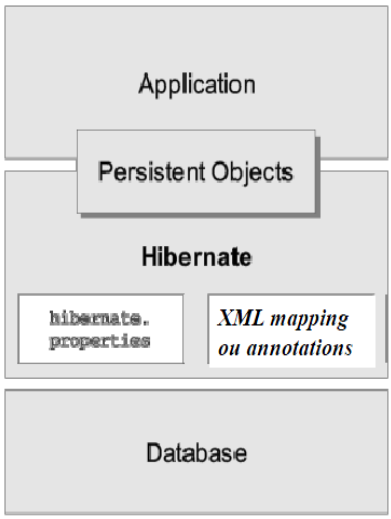
HIBERNATE

Hibernate : généralités

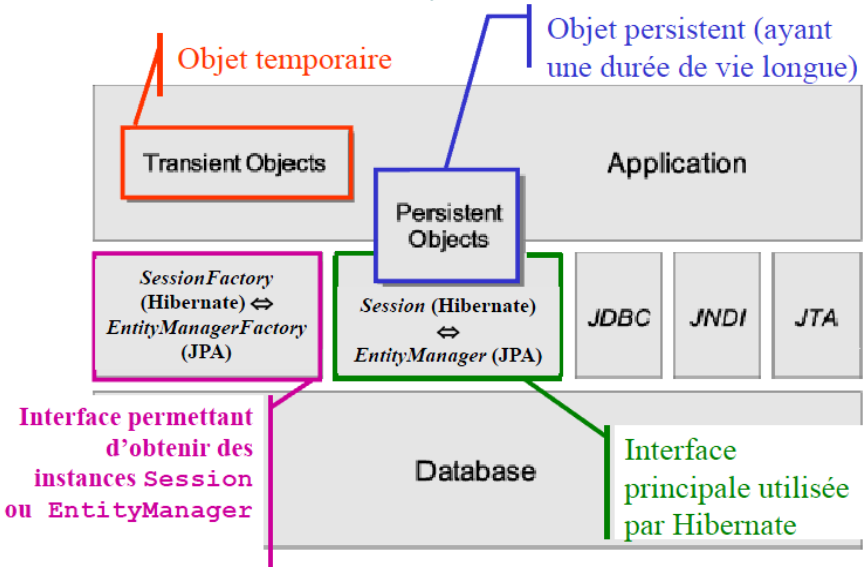
- Outil ORM ou Cadre *(Framework) de persistance* libre (*open source) gérant la persistance des objets* Java/J2EE en base de données relationnelle [Wikipédia, Pat05]
- Version 3.x (2010:3.6) : implémentation du standard de persistance EJB 3.0 *Java Persistence API (JPA)*
- Possibilité d'être utilisé aussi bien dans un développement **client lourd**, que dans un environnement **web léger** de type Apache Tomcat ou dans un environnement J2EE complet.
- **Code SQL généré à l'exécution via des informations fournies dans un document de correspondance (*mapping) XML ou des annotations***

Architecture du noyau Hibernate

- **`hibernate.properties`** : Fichier de configuration
  - Version XML : **hibernate.cfg.xml** permettant un paramétrage plus fin
  - Configuration par programmation
- *XML mapping* ou *Annotations* :
  - Méta-données (paramètres) décrites sous la forme de fichiers de correspondance XML ou sous forme d'annotation
  - Utilisées pour mettre en correspondance les classes Java et le modèle relation



Architecture du noyau Hibernate



Architecture du noyau Hibernate

- Différents modules :
  - ***Hibernate Core :* API native implémentant les services de base pour** la persistance Méta-données au format XML (+ annotations depuis la version 3.6) Langage HQL et interface pour écrire des requêtes
  - ***Hibernate Annotations (inclus dans Hibernate Core 3.6) :*** Remplacement des fichiers XML par des annotations JDK 5.0 implémentant les annotations du standard JPA + annotations spécifiques à Hibernate
  - ***Hibernate Entity Manager :*** *Implémentation de la partie des* spécifications **JPA** concernant
    - Les interfaces de programmation,
    - Les règles de cycle de vie des objets persistants
    - Les fonctionnalités d'interrogation *Hibernate Entity Manager = wrapper au dessus du noyau Hibernate*

Architecture du noyau Hibernate

- **SessionFactory (Core) ou EntityManagerFactory (JPA):**
  - Cache immuable (*threadsafe) des correspondances (mappings)* vers une (et une seule) base de données
  - Coûteuse à construire car implique l'analyse des fichiers de configuration
  - Pouvant contenir un cache optionnel de données (de second niveau) réutilisable entre les différentes transactions
  - Construite à partir d'un objet (Ejb3) Configuration
- **Session (Core) ou EntityManager (JPA) :**
  - Objet *mono-threadé, à durée de vie courte, représentant une* conversation entre l'application et l'entrepôt de persistance (eg. Base de Données)
  - Encapsule une connexion JDBC
  - Contient un cache de premier niveau (obligatoire) et des objets persistants

Travailler avec Hibernate Core :

- Les fichiers nécessaires sont les suivants :
- **hibernate.cfg.xml : fichier de configuration globale contenant**
  - Les paramètres de connexion à la base de données (pilote, login, mot de passe, url, etc.)
  - Le dialecte SQL de la base de données
  - La gestion de pool de connexions
  - Le niveau de détails des traces etc.
- Pour chaque classe persistante :
  - **ClassePersistante.java** : Implémentation **POJO** (*Plain Old Java Objects) de la classe similaire à l'implémentation des* ***Bean***
  - **ClassePersistante.hbm.xml :** Fichier XML de correspondance (***mapping***)
  - **ClassePersistanteHome.java ou ClassePersistanteDAO.java :** Implémentation du DAO (*Data Access Object) pour l'isolation avec la couche* de persistance – Optionnel
  - **ClassePersistante.sql :** Code SQL de création de la ou les relations – Optionnel – pouvant être généré par Hibernate

Travailler avec Hibernate (JPA) :

- Les bibliothèques supplémentaires sont nécessaires :
  - **hibernate-annotations.jar**
    - Annotations propres à Hibernate
  - **ejb3-persistence.jar**
    - Annotations du standard EJB3

Depuis la version 3.5, ces jar font partie de **hibernate3.jar**

- Fichiers nécessaires pour *Hibernate (JPA)* :
  - **hibernate.cfg.xml :** fichier de configuration
  - **ClassePersistante.java :** POJO avec annotations
  - **ClassePersistante.hbm.xml :** optionnel – possibilité de combiner annotations et méta-données XML
  - **ClassePersistanteHome.java ou ClassePersistanteDAO.java :** Implémentation du DAO (*Data Access Object) pour l'isolation avec la couche* de persistance – Optionnel

Exemple de travaille avec Hibernate Core

- Afin de gérer la configuration et les sessions hibernate, il est fortement conseillé de créer une classes **`HibernateUtil`** qui permettra de:
  - •Créer une occurrence de la classe **`Configuration`** afin de lire **`hibernate.cfg.xml`**
  - •Créer une **`SessionFactory`** afin de gérer les transactions et l'interrogation de la base de données.

```
package persistance;
import org.hibernate.*;
importorg.hibernate.cfg.*;
public class HibernateUtil {
private static SessionFactory sessionFactory;
static {
          try {
          sessionFactory = new Configuration().configure().buildSessionFactory();
          }
          catch (Throwable ex) {
          throw new ExceptionInInitializerError(ex); }
}
public static SessionFactory getSessionFactory() { return sessionFactory; }

public static void shutdown() {  getSessionFactory().close(); }

}
```

Exemple de travaille avec Hibernate Core

La **SessionFactory** nous permettra d'ouvrir des **sessions** afin d'effectuer des transactions et interroger la base de données

```
import org.hibernate.*;
import persistance.HibernateUtil;

public class Tester {

public static void main(String[] args) {
        //ouvrir une session
Session session1 =
HibernateUtil.getSessionFactory().openSession();
        //commencer une transaction
Transaction tx1 = session1.beginTransaction();

        //consulter modifier la base

        //fermer la transaction
        tx1.commit();
        //fermer la session
        session1.close();
}}
```


Le fichier *hibernate.cfg.xml*

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
  "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
  "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <!-- Database connection settings -->
    <property name="hibernate.connection.driver_class">org.postgresql.Driver
     </property>
    <property name="hibernate.connection.password">passwd</property>
    <property name="hibernate.connection.url">jdbc:postgresql:BDTest2</property>
    <property name="hibernate.connection.username">login</property>
   <!-- SQL dialect -->
   <property name="dialect">org.hibernate.dialect.PostgreSQLDialect</property>
   <!-- Enable Hibernate's automatic session context management -->
   <property name="current_session_context_class">thread</property>
   <!-- Disable the second-level cache  -->
   <property  name="cache.provider_class">org.hibernate.cache.NoCacheProvider
    </property>
   <!-- Echo all executed SQL to stdout -->
   <property name="show_sql">true</property>
   <!-- Drop and re-create the database schema on startup -->
   <property name="hbm2ddl.auto">create</property>
   <mapping resource="events/Event.hbm.xml"/>
   <mapping resource="events/Person.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

Le fichier *hibernate.cfg.xml*

- Déclaration du type de document utilisé par l'**analyseur syntaxique *(parseur) XML pour valider le document de*** configuration d'après la DTD de configuration d'Hibernate

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
  "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
  "http://hibernate.sourceforge.net/hibernate-configuration-
  3.0.dtd">
```

- Paramètres de configuration nécessaires pour la connexion JDBC :

```
<property
  name="hibernate.connection.driver_class">org.postgresql.Driver
 </property>
<property name="hibernate.connection.password">passwd</property>
<property name="hibernate.connection.url">jdbc:postgresql:BDTest2
 </property>
<property name="hibernate.connection.username">login
 </property>
```

- Spécification de la variante de SQL générée par Hibernate. Elle dépend du SGBD utilisé

```
<!-- SQL dialect -->
   <property
     name="dialect">org.hibernate.dialect.PostgreSQLDialect
   </property>
```


Le fichier *hibernate.cfg.xml*

- Activation de la génération automatique des schémas de base de données - directement dans la base de données (*Optionnelle, utiliser pour générer une base de données à partir des classes Java et fichiers de mapping*)

```
<!-- Drop and re-create the database schema on startup -->
   <property name="hbm2ddl.auto">create</property>
```

- Affichage de trace d'exécution de requêtes SQL

```
<!-- Echo all executed SQL to stdout -->
<property name="show_sql">true</property>
```

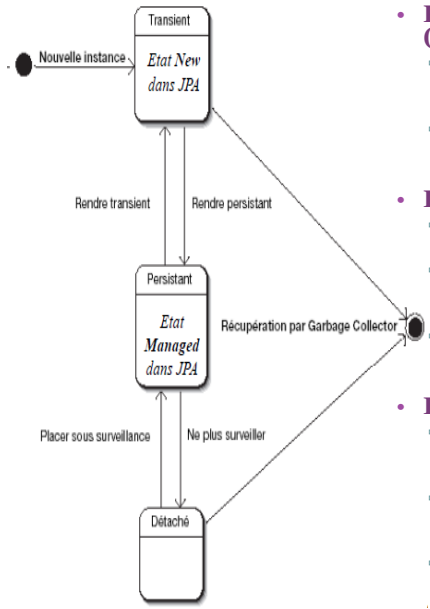
- Fichier de correspondances (de *mapping) des classes* persistantes. Il faut indiquer le path du fichier:

```
<mapping resource="events/Event.hbm.xml"/>
<mapping resource="events/Person.hbm.xml"/>
```

Classes persistantes dans Hibernate

- Classes persistantes : implémentation des entités métiers sous la forme de POJO
- Pour manipuler des objets persistants :
  1. Ouverture d'une `Session` Hibernate
  2. [Débuter une transaction] – fortement conseillé
  3. Appliquer les opérations de `Session` pour interagir avec l'environnement de persistance
  4. [Valider (`commit`()) la transaction en cours]
  5. Synchroniser avec la base de données (***`flush`***) *et fermer* la session
- Annuler (`rollback`()) la transaction et fermer la `Session` en cas d'exception soulevée par les méthodes de Session

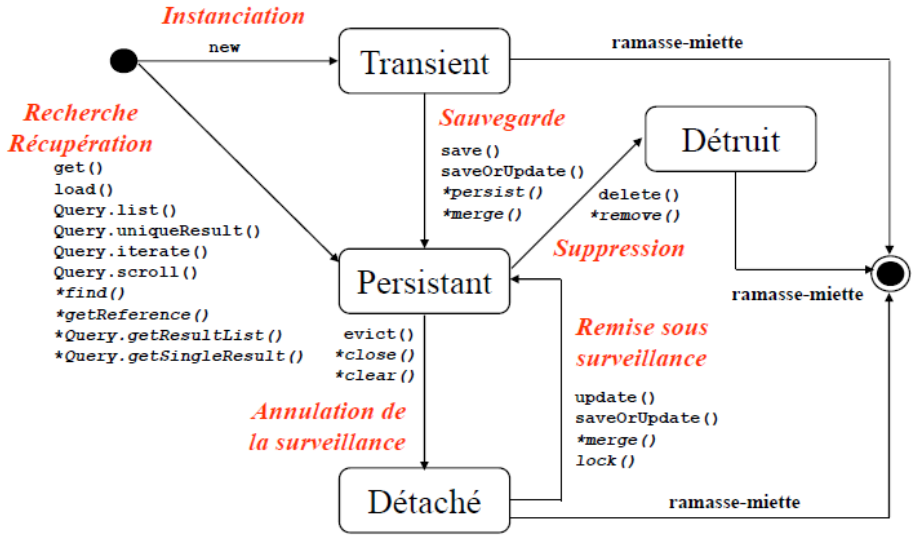
Etats des classes persistantes



- **Passager/Temporaire/Éphémère (*transient*) :**
  - Instance non associée à un contexte de persistance donc sans identité persistante (i.e. sans valeur de clé primaire)
  - Instance crée avec ***new () et non encore sauvegardé en BD.***
- **Persistant :**
  - Instance associée à un contexte de persistance (Session)
  - Instance possédant une identité persistante (i.e. valeur de clé primaire) et, peut-être, un enregistrement/nuplet correspondant dans la base
  - *Hibernate garantie* l'équivalence entre l'identité persistante et l'identité Java
- **Détaché**
  - Instance ayant été associée au contexte de persistance à présent fermé ou instance ayant été sérialisée vers un autre processus
  - Instance possèdant une identité persistante et peut-être un enregistrement/nuplet correspondant dans la base
  - Aucune garantie par Hibernate sur la relation entre l'identité persistante et l'identité Java

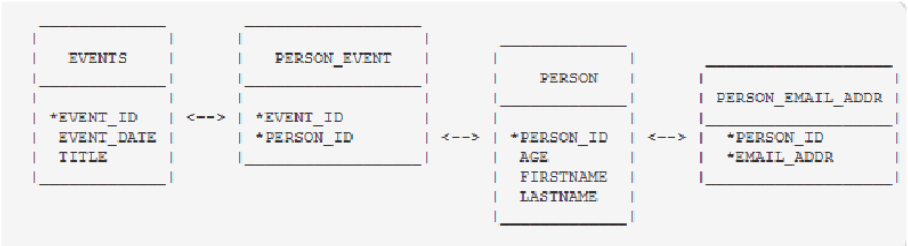
Cycle de persistance dans Hibernate

- *Diagramme d'états des objets d'une classe persistante*



Exemple de mapping d'une table

Modèle relationnel :



Modèle objet :

```
public class Person {

    private Long id;
    private int age;
    private String firstname;
    private String lastname;
    private Set events ;
    private Set emailAddresses ;
    …
}
```

```
public class Event {

    private Long id;
    private String title;
    private Date date;
    private Set participants;
    …
}
```

Fichier du mapping XML

- Exemple de fichier hbm de correspondance pour la classe **`Person`**,

`Person.hbm.xml`

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <class name="events.Person" table="PERSON">
        <id name="id" column="PERSON_ID">
          <generator class="native"/>
        </id>
        <property name="age"/>
        <property name="firstname"/>
        <property name="lastname"/>
        <set name="events" table="PERSON_EVENT">
          <key column="PERSON_ID"/>
          <many-to-many column="EVENT_ID" class="events.Event"/>
        </set>
        <set name="emailAddresses" table="PERSON_EMAIL_ADDR">
          <key column="PERSON_ID"/>
          <element type="string" column="EMAIL_ADDR"/>
        </set>
    </class>
</hibernate-mapping>
```

Déclaration de la DTD

Déclaration du mapping

Classe `event.Person` « mappée » en relation `PERSON`

Mapping de l'identifiant

Mapping de collection

Attribut de jointure entre PERSON et PERSON_EVENT

Pour préciser qu'il y a une « table association »

Pour définir une collection de valeurs

Fichiers de *mapping* des classes métiers

- Éléments des fichiers de *mapping XML :*
  - Déclaration de la **DTD (entête du fichier)**
  - Élément racine : **<hibernate-mapping>** Possibilité d'y définir l'attribut package pour éviter de le spécifier à chaque déclaration de classe et d'association
  - **<class>:** pour spécifier la correspondance entre une relation de base de données et une classe
  - **<property> :** pour spécifier la correspondance entre une colonne de relation et une propriété de classe
  - **<many-to-one> et <one-to-one>** : pour spécifier la correspondance d'une association vers une entité
  - **<component> :** pour spécifier la correspondance d'une association vers un composant
  - **<one-to-many> et <many-to-many> :** pour spécifier la correspondance d'une association vers une collection d'entités

31

Description de la balise <class>

- **name :** nom Java complet de la classe (ou interface) persistante
- **table :** nom de la relation en base de données – par défaut nom (non-qualifié) de la classe
- **schema :** Surcharge le nom de schéma spécifié par l'élément racine (optionnel)

Balise XML
```
<class name="package.Departement" table="departement"
schema="public">
```

Annotations
```
@Entity
@Table(name="departement",schema="public")
public class Departement { … }
```

- + d'autres pour gérer la génération des requêtes SQL à l'exécution, la stratégie de verrouillage, les cas où la classe appartient à une hiérarchie etc.

32

Description de la balise <class>

- **L'attribut <id>**
  - **identité relationnelle de l'entité persistante**
    - Pour définir la correspondance entre la propriété identificateur de la classe et la *clé primaire* de la relation
    - Obligatoire pour toute classe représentée par une relation de base de données
    - **name :** nom de la propriété identifiant – si non référencé Hibernate considère que la classe n'a pas d'identifiant
    - **column :** nom de la colonne clé primaire – par défaut de même nom que la propriété (attribut **name)**
    - **generator :** génération automatique d'id par un cretain algorithme comme suit :
      - **increment** : génère des identifiants de type long, short ou int qui ne sont uniques que si aucun autre processus n'insère de données dans la même table
      - **native** : choisit le générateur en fonction de la base de données (sequence pour Oracle ou PostgreSQL, identity pour MYSQL par ex.)
      - **assigned** : laisse l'application affecter un identifiant à l'objet avant que la métode **save()** ou **persist()** ne soit appelée.- Stratégie par défaut si aucun <generator> n'est spécifié

```
<id name="id" column="ID">
    <generator class="sequence">
        <param name="sequence">SEQ_REF_ARTICLE</param>
    </generator>
</id>
```

Description de la balise <composite-id>

- **Clé composite :**

```
<class name="test1_package.Enseignement" table="enseignement" schema="public">
        <composite-id name="id" class="test1_package.EnseignementId">
            <key-property name="enseignementId" type="int">
                <column name="enseignement_id" />
            </key-property>
            <key-property name="departementId" type="int">
                <column name="departement_id" />
            </key-property>
        </composite-id>
...
</class>
```

```
public class EnseignementId implements java.io.Serializable {
      // Fields
      private int enseignementId;
      private int departementId;
      public EnseignementId() {} /** default constructor */
      public boolean equals(Object other) { …}
      public int hashCode() {…}
     // Property accessors
  …
}
```

```
public class Enseignement implements java.io.Serializable {
     // Fields
     private EnseignementId id;
…}
```


Description de la balise <property>

- **Déclaration de propriété persistante :**
  - **name :** nom de la propriété, avec une lettre initiale en minuscule **(cf.** convention *JavaBean)*
  - **column :** nom de la colonne de base de données correspondante – par défaut de même nom que la propriété
  - **type** : nom indiquant le type Hibernate - déterminé par défaut par introspection de la classe
  - **update, insert :** indique que les colonnes *mappées doivent être* incluses dans les **UPDATE** et/ou les **INSERT -** par défaut à **true.** Mettre les deux à **false** empêche la propagation en base de données (utile si vous savez qu'un trigger affectera la valeur à la colonne)
  - **unique :** Génère le DDL d'une contrainte d'unicité pour les colonnes- optionnel
  - **not-null :** Génère le DDL d'une contrainte de non nullité pour les colonnes – optionnel

Description de la balise <property>

```
<hibernate-mapping>
        <class name="events.Event" table="EVENTS">
            <id name="id" column="EVENT_ID">
                <generator class="native"/>
            </id>
            <property name="date" type="timestamp"
                 column="EVENT_DATE"/>
            <property name="title"/>
    …
    </class>
    </hibernate-mapping>

    <property name="nomDepartement" type="string">
          <column name="nom_departement" length="25"
                  not-null="true" unique="true" />
    </property>
```


Les associations en objet

- Dans le code objet une association peut être représentée par
  - **Association (1:1 ou N:1)** une variable d'instance référençant l'objet associé
  - **Association (1:N ou M:N)** une variable d'instance de type **Set** ou ***Map*** *contenant les objets associés*
  - **Association (M:N)** : Si l'association (M:N) contient des propriétés, utiliser une classe d'association devient obligatoire.

Traduction des associations 1:1

- **<one-to-one> :**

Dans **Employee.hbm.xml** :

```
<one-to-one name="person" class="Person"/>
```

Dans **Person.hbm.xml** :

```
<one-to-one name="employee" class="Employee"
            constrained="true"/>
```

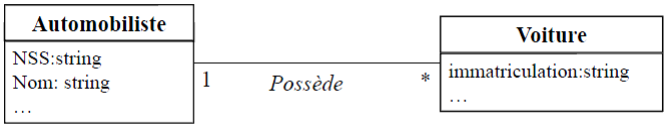
La clé primaire de person fait référence à la clé primaire de employee

▪ Stratégie Hibernate spéciale de génération d'identifiants par **foreign**

⇒ Attribution de la *même valeur* de clé primaire à une instance fraîchement enregistrée de **Person** et l'instance de **Employee** référencée par la propriété **employee** de **Person**

```
<class name="person" table="PERSON">
    <id name="id" column="PERSON_ID">
        <generator class="foreign">
            <param name="property">employee</param>
        </generator>
    </id>
    ...
    <one-to-one name="employee"
        class="Employee"
        constrained="true"/>
</class>
```


Traduction des associations 1:N (ou N:1)




```
CREATE TABLE Automobilsite
( Automobiliste_ID  SERIAL,
  NSS varchar(10) NOT NULL,
  Nom varchar(20) NOT NULL,
  …
  CONSTRAINT PK_Automobiliste PRIMARY KEY (Automobiliste_ID),
) ;

CREATE TABLE Voiture
( Voiture_ID  SERIAL,
  Immatriculation varchar(10) NOT NULL,
  …
  Proprietaire_ID int NOT NULL,
  CONSTRAINT PK_Automobiliste PRIMARY KEY (Voiture_ID),
  CONSTRAINT PK_Automobiliste_Voiture
  FOREIGN KEY (Proprietaire_ID) REFERENCES Automobiliste (Automobiliste_ID)
);
```

De Automobiliste vers Voiture

```
public class Automobiliste {
  // Fields
  private String NSS;
  private String nom;
  private Collection<Voiture> parc_automobile;
  …
}
```

De Voiture vers Automobiliste

```
public class Voiture {
  // Fields
  private String immatriculation;
  private Automobiliste proprietaire;
  …
}
```

Traduction des associations 1:N (ou N:1)

Exemple d'association uni-directionnelle :

```
<class name="test1_package.Enseignement" table="enseignement"
       schema="public">
        ...
        <many-to-one name="departement"
                 class="test1_package.Departement">
            <column name="departement_id" not-null="true" />
        </many-to-one>
        ...
</class>
```


```
/*** Enseignement generated by hbm2java*/
public class Enseignement implements java.io.Serializable {
     // Fields
     private EnseignementId id;
     private Departement departement;
     private String intitule;
     private String description;
     ...
}
```


Traduction des associations 1:N (ou N:1)

- Par analogie dans le mapping de `Departement`, nous allons créer un association `<one-to-many>` vers `Enseignements`
- L'association `<one-to-many>` de traduit par un `Set<Enseignement>` au niveau de la classe Departement

```
<set name="enseignements" inverse="true">
  <key>
    <column name="departement_id" not-null="true" />
  </key>
  <one-to-many class="package.Enseignement" />
</set>
```

2. Traduction des association binaire M:N - Objet

- **Correspondance des collections :**
- Implémentations des collections **Set, List et Map** propres à Hibernate
- **Balise <key column= … > :**
  - Pour spécifier comment effectuer la jointure entre les deux relations entrant en jeu dans l'association **to-many**
  - **column :** nom de la clé étrangère

Dans le fichier Person.hbm.xml

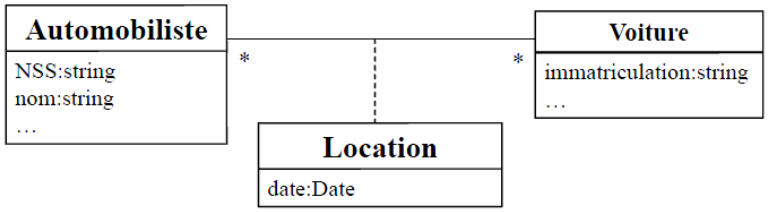
```
<set name="events" table="PERSON_EVENT">
    <key column="PERSON_ID"/>
    <many-to-many column="EVENT_ID" class="events.Event"/>
</set>
```

Dans le fichier Event.hbm.xml

```
<set name="participants" table="PERSON_EVENT"
    inverse="true">
    <key column="EVENT_ID"/>
    <many-to-many column="PERSON_ID" class="events.Person"/>
</set>
```


2. Traduction des association binaire M:N - Objet

- **2ème Solution avec classe d'association**




```
public class Location {
  private Automobiliste loueur;
  private Voiture vehicule;
  private Date date;
  …
}
```

```
CREATE TABLE Location
( …
 CONSTRAINT PK_Location PRIMARY KEY (Automobiliste_ID,Voiture_ID, Date),
 CONSTRAINT FK_Location_Voiture
   FOREIGN KEY (Voiture_ID) REFERENCES Voiture (Voiture_ID),
 CONSTRAINT FK_Location_Automobiliste
   FOREIGN KEY (Automobiliste_ID) REFERENCES Automobiliste (Automobiliste_ID),
);
```

Opérations du gestionnaire de persistance

- **Récupérer** une instance persistante dans *Hibernate Core* : **`load()`** *vs* **`get()`**

```
Session session = HibernateUtil.getSessionFactory().getCurrentSession();
session.beginTransaction();
Person aPerson1 = (Person) session.load(Person.class, new Long(1));
if (aPerson1!=null) System.out.println(aPerson1.getFirstname());
session.getTransaction().commit();
```

```
⇒Hibernate: select person0_.PERSON_ID as PERSON1_2_0_, person0_.age as
age2_0_, person0_.firstname as firstname2_0_, person0_.lastname as
lastname2_0_ from PERSON person0_ where person0_.PERSON_ID=?

Exception in thread "main" org.hibernate.ObjectNotFoundException: No row
with the given identifier exists: [events.Person#1]
```

```
Session session =HibernateUtil.getSessionFactory().getCurrentSession();
session.beginTransaction();
Person aPerson1 = (Person) session.get(Person.class, new Long(1));
if (aPerson1!=null) System.out.println(aPerson1.getFirstname());
session.getTransaction().commit();
```

```
⇒Hibernate: select person0_.PERSON_ID as PERSON1_2_0_, person0_.age as
age2_0_, person0_.firstname as firstname2_0_, person0_.lastname as
lastname2_0_ from PERSON person0_ where person0_.PERSON_ID=?
Mais pas d'exception!! (sauf si on n'oublie de tester aPerson2)
```


Opérations du gestionnaire de persistance

- Rendre une instance persistante :
  - session.save(objet)
  - session.persist(objet)
  - session.merge(objet)

```
Session session = HibernateUtil.getSessionFactory().getCurrentSession();
session.beginTransaction();

Event theEvent = new Event();
theEvent.setTitle(title);
theEvent.setDate(theDate);

session.save(theEvent); // Même chose avec session.persist(theEvent);
                        // ou avec entityManager.persist(theEvent);
session.getTransaction().commit();
```

Car nous avons utilisé <generator> dans le fichier hbm

```
⇒ select nextval ('hibernate_sequence')
⇒ insert into EVENTS (EVENT_DATE, title, EVENT_ID) values (?, ?, ?)
```

SQL généré

Opérations du gestionnaire de persistance **merge**

```
// Après le code du transparent précédent
theEvent.setTitle("Test2"); // Mise à jour d'une instance détachée
Session session2 = HibernateUtil.getSessionFactory().getCurrentSession();
session2.beginTransaction();
Event theEvent2 = (Event)session2.merge(theEvent);
// TheEvent est toujours détaché, theEvent2 est persistant
session2.getTransaction().commit();
```

Suite au code du transparent précédent :

⇒ select nextval ('hibernate_sequence')

⇒ insert into EVENTS (EVENT_DATE, title, EVENT_ID) values (?, ?, ?)

Suite au merge :

⇒ **select … from EVENTS event0_ where event0_.EVENT_ID=?**

⇒ **update EVENTS set EVENT_DATE=?, title=? where EVENT_ID=?**

ou

insert into EVENTS (EVENT_DATE, title, EVENT_ID) values (?, ?, ?)

si pas d'utilisation de la session dans le transparent précédent

Opérations du gestionnaire de persistance

automatic dirty checking

```
// Après le code du transparent précédent
Session session = HibernateUtil.getSessionFactory().getCurrentSession();
session.beginTransaction();
Person aPerson1 = (Person) session.load(Person.class, new Long(2));
aPerson1.setAge(32);
session.getTransaction().commit();
```

⇒ **select … PERSON person0_ where person0_.PERSON_ID=?**

⇒ **update PERSON set age=?, firstname=?, lastname=? where PERSON_ID=?**

Il faut appeler `save` ou `persist` pour rendre un objet transient persistant (pas de persistance automatique)

```
Session session = HibernateUtil.getSessionFactory().getCurrentSession();
session.beginTransaction();
Event theEvent = new Event();
theEvent.setTitle("Test3");
theEvent.setDate(new Date());
session.getTransaction().commit();
```

⇒ **aucune requête générée**

Outils de récupération des instances persistantes

- Hibernate supporte ces différents sortes de requêtes :
  - HQL (*Hibernate Query Language)*
  - EJB-QL(EJB*Query Language)*
  - API Criteria
  - Requêtes natives en SQL (*Native Query)*
- Stratégies de chargement d'Hibernate3 :
  - **Chargement par jointure :** Récupération de l'instance ou la collection dans un même SELECT, en utilisant un OUTER JOIN
  - **Chargement par select** : Utilisation d'un second SELECT pour récupérer l'instance ou la collection**, Chargement tardif** ⇒ second SELECT exécuté uniquement lors de l'accès réel à l'association
  - **Chargement par lot :** stratégie d'optimisation pour le chargement par SELECT = récupèration d'un lot d'instances ou de collections en un seul SELECT en spécifiant une liste de clé primaire ou de clé étrangère

Exemples de stratégies de chargement

- **Chargement tardif (`lazy`) par défaut pour les deux collections `events` et `participants` :**

```
Person aPerson1 = (Person) session.get(Person.class, new
Long(4));
```

```
⇒ select … from PERSON person0_ where person0_.PERSON_ID=?
```

- **Chargement agressif (`eager`) pour la collection `events` avec stratégie par défaut (`select`):**

```
Person aPerson1 = (Person) session.get(Person.class, new
Long(4));
```

```
⇒ select … from PERSON person0_ where person0_.PERSON_ID=?
⇒ select … from PERSON_EVENT events0_ inner join EVENTS
    event1_ on events0_.EVENT_ID=event1_.EVENT_ID
    where events0_.PERSON_ID=?
```

- **Chargement agressif pour la collection `events` avec stratégie charg. par jointure (`join`) :**

```
Person aPerson1 = (Person) session.get(Person.class, new
Long(4));
```

```
⇒ select … from PERSON person0_ left outer join PERSON_EVENT
    events1_ on person0_.PERSON_ID=events1_.PERSON_ID
    left outer join EVENTS event2_
    on events1_.EVENT_ID=event2_.EVENT_ID
    where person0_.PERSON_ID=?
```

49

Requête HQL

- "Langage de requêtes orientées objet" ou encapsulation du SQL selon une logique orientée objet
- Requêtes HQL (et SQL natives) représentées avec une instance de **org.hibernate.Query**
- Obtention d'une **Query en utilisant la Session** courante : **session.createQuery (string)**
- Clauses : **from, select, where**
- Invocation de la méthode **list() ⇒ retour du** ré

```
List result = session.createQuery("from Event").list();
```

50

Requête HQL

- **from :** Clause suivie d'un nom de classe et non de la table de BD : **from Event**
  - Possibilité d'appliquer la clause sur tout type de classe (abstraite ou concrète, interface)
- **select :** Non obligatoire
  - Raisonnement objet ⇒ possibilité de naviguer à travers le graphe d'objets
- **join :** Pour exécuter des jointures **(inner join)** Exemple :

  select p from Person p
  join p.events where p.id = 4
- **left join :** Jointure externe **(left outer join)**

  select p from Person p
  left join p.events where p.id = 4
- **Lier les paramètres :**

```
Long personId ;
Person aPerson = (Person)session.createQuery("select
p from Person p left join fetch p.events where p.id
= :pid").setParameter("pid", personId)
        .uniqueResult();
```

API Criteria

- API d'interrogation par critères dite intuitive et extensible – appartenant au noyau
- Obtention des instances persistances d'une classe

```
Criteria crit = session.createCriteria(Person.class);
List resultat = crit.list();
```

Exécution de la requête

- **Définition de critères de recherche**

```
List resultat = session.createCriteria(Person.class)
    .add( Restrictions.like("name", "Manou%") )
    .add( Restrictions.or(
        Restrictions.eq( "age", new Integer(0) ),
        Restrictions.isNull("age")
    ) )
    .list();
```


API Criteria

- Peuplement d'associations de manière dynamique

```
List resultat = session.createCriteria(Person.class)
    .add( Restrictions.like("name", "Manou%") )
    .setFetchMode("events", FetchMode.JOIN)
    .list();
```

pour choisir le mode de chagement
Valeurs: `FetchMode.JOIN` (pour imposer une jointure externe), `FetchMode.SELECT` (pour charger l'association par un SELECT supplémentaire)

- Requêtes par l'exemple

```
Person p = new Person();
p.setName("Manouvrier");
List results = session.createCriteria(Person.class)
                .add( Example.create(p) ) .list();
```

53

Requêtes SQL natives

- Pour utiliser des requêtes optimisées et tirer partie des spécificités du SGBD utilisé
- **Requêtes natives du noyau**

```
session.createSQLQuery("SELECT * FROM PERSON").list();
```

Retourne une liste d'`Object[]` avec des valeurs scalaires pour chaque colonne de la table PERSON (i.e. retourne une table comme pour les requêtes classiques JDBC)

```
session.createSQLQuery("SELECT * FROM PERSON")
        .addEntity(Person.class);
```

Retourne une liste d'objets de la classe `Person`

54

Transaction

- Pour le noyau ou *Hibernate Entity Manager :*

```
Session sess = factory.openSession(); // Même chose en utilisant
Transaction tx = null;                // EntityManager
try {
    tx = sess.beginTransaction();
    // do some work
    ...
    tx.commit();
}
catch (RuntimeException e) {
    if (tx != null) tx.rollback();
    throw e; // or display error message
}
finally {
    sess.close();
}
```

Déclenchement automatique de la synchronisation avec la BD

Toutes exceptions soulevées ne sont pas récupérables et doivent être considérées comme fatales pour le gestionnaire de persistance en cours (il faut annuler la transaction et fermer l'unité de travail)

Transaction

- Définition du niveau d'isolation :
- En SQL :
  - ***SET TRANSACTION ISOLATION LEVEL SERIALIZABLE READ ONLY***
- Dans le fichier de config (`hibernate.hbm.xml` ou `persistence.xml`) :

<property name="hibernate.connection.isolation" value="x"/>

  - avec x = 1 pour **read uncommitted**
  - 2 pour **read committed**
  - 4 pour **read-uncommitted, committed, repeatable read**
  - 8 pour **serializabe**