# Applications orientées données (NSY135)

## 7 – Introduction à JPA et Hibernate

Auteurs: Raphaël Fournier-S'niehotta et Philippe Rigaux
(philippe.rigaux@cnam.fr,fournier@cnam.fr)

Département d'informatique
Conservatoire National des Arts & Métiers, Paris, France

# Plan du cours

# Installation

- Récupérez la dernière version stable sur le site **Hibernate** : `http://hibernate.org` (4.3.7 le 11/12/14)
- Dans le répertoire **hibernate** obtenu après décompression, il y a une documentation complète et les librairies, dans **./lib**
- Il y a plusieurs sous-catégories/sous-répertoires, pour l'instant on n'a besoin que de toutes les librairies de **required**
- on les copie dans **WEB-INF/lib**
- on utilise **refresh** sous Eclipse (clic bouton droit) pour que ces nouvelles librairies soient identifiées

- Un des objets essentiels dans une application Hibernate est l'objet **Session** qui est utilisé pour communiquer avec la base de données.
- Il peut être vu comme une généralisation/extension de l'objet **Connection** de JDBC.
- Une session est créée à partir d'un ensemble de paramètres (dont les identifiants de connexion JDBC) contenus dans un fichier de configuration XML nommé **hibernate.cfg.xml**.

# Fichier de configuration initial

- Pour que ce fichier soit automatiquement déployé par Tomcat, il doit être placé dans un répertoire **WEB-INF/classes**. Créez ce répertoire s'il n'existe pas.

```
    <?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>
<!-- local connection properties -->
<property name="hibernate.connection.url">jdbc:mysql://localhost:3306/webscope</property>
<property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
<property name="hibernate.connection.username">orm</property>
<property name="hibernate.connection.password">orm</property>
<property name="hibernate.connection.pool_size">10</property>

<!-- dialect for MySQL -->
<property name="dialect">org.hibernate.dialect.MySQLInnoDBDialect</property>

<property name="hibernate.show_sql">true</property>
<property name="cache.provider_class">org.hibernate.cache.NoCacheProvider</property>
<property name="cache.use_query_cache">false</property>

  </session-factory>
    </hibernate-configuration>
```

# Fichier de configuration

- On retrouve les paramètres qui vont permettre à Hibernate d'instancier une connexion à MySQL avec JDBC (paramètres à changer).
- chaque système relationnel propose quelques variantes du langage SQL qui sont prises en compte par Hibernate: on indique le dialecte à utiliser.
  **property name=dialect**
- Finalement, les derniers paramètres ne sont pas indispendables mais vont faciliter notre découverte, notamment avec l'option **show_sql** qui affiche les requêtes SQL générées dans la console java.
- Il peut y avoir plusieurs <**session-factory**> dans un même fichier de configuration, pour plusieurs bases de données, éventuellement dans plusieurs serveurs différents.

# Test de la connexion

- Comme pour JDBC, nous créons :
  - un contrôleur **Hibernate.java** associé à l'URL **/hibernate**. Il a à peu près la même structure que celui utilisé pour JDBC (nous allons reproduire les mêmes actions)
  - une liste d'actions implantés dans une classe de tests nommée **TestsHibernate.java**;
  - des vues placées dans **WebContent/vues/hibernate**.

# Test de la connexion : TestsHibernate.java

```
  import org.hibernate.*;
  import org.hibernate.cfg.Configuration;

  public class TestsHibernate {

/**
 * Objet Session de Hibernate
 */
private Session session;

  /**
   * Constructeur établissant une connexion avec Hibernate
   */
  public TestsHibernate()  {
    Configuration configuration = new Configuration().configure();

    ServiceRegistry serviceRegistry = new StandardServiceRegistryBuilder()
      .applySettings(configuration.getProperties()).build();

    SessionFactory sessionFactory = configuration.buildSessionFactory(serviceRegistry);

    session = sessionFactory.openSession();
    }
  ...
  }
```

# Test de connexion (suite)

- Pour créer une connexion avec Hibernate nous passons par une **SessionFactory**
- Celle-ci repose sur le paramétrage de notre fichier de configuration
- ce fichier lui-même est chargé dans un objet **Configuration**

- Si tout se passe bien, on obtient un objet par lequel on peut ouvrir/fermer des sessions.
- Plusieurs raisons peuvent faire que ça ne marche pas (au moins du premier coup).
  - le fichier de configuration n'est pas trouvé (il doit être dans le **CLASSPATH** de l'application);
  - ce fichier contient des erreurs;
  - les paramètres de connexion sont faux;
  - et toutes sortes d'erreurs étranges qui peuvent nécessiter le redémarrage de Tomcat..
- En cas de problème une exception sera levée avec un message plus ou moins explicite.

# Testons la connexion

```
TestsHibernate tstHiber = new TestsHibernate();
    // Action + vue test de connexion
    maVue = VUES + "connexion.jsp";
```

# Une première entité

- Dans JPA, on associe une table à une classe Java annotée par **@Entity**.
- Chaque ligne de la table devient une **entité**, instance de cette classe.
- La classe doit obéir à certaines contraintes qui sont à peu près comparables à celle d'un JavaBean.
- Les propriétés sont privées, et des accesseurs **set** et **get** sont définis pour chaque propriété.
- L'association d'une propriété à une colonne de la table est indiquée par des annotations.

# Exemple

```java
package modeles.webscope;

import javax.persistence.*;

@Entity
public class Pays  {
@Id
String code;
public void setCode(String c) {code = c;}
public String getCode() {return code ;}

@Column
String nom;
public void setNom(String n) {nom = n;}
  public String getNom() {return nom;}

@Column
String langue;
public void setLangue(String l) {langue = l;}
   public String getLangue() {return langue;}
}
```

# Première entité (suite)

- Remarquez l'import du **package javax.persistence**, et les annotations suivante :
  - **@Entity** : indique que les instances de la classe sont **persistantes** (stockées en BDD);
  - **@Id** : indique que cette propriété est la clé primaire;
  - **@Column** : indique que la propriété est associée à une colonne de la table.
- Bien entendu les annotations peuvent être beaucoup plus complexes.
- Ici, on s'appuie sur des choix par défaut : le nom de la table est le même que le nom de la classe; idem pour les propriétés, et l'identifiant n'est pas auto-généré.
- Déclarons dans le fichier de configuration que cette entité persistante est dans notre base :

```
<session-factory>
    (...)
  <mapping class="modeles.webscope.Pays"/>
</session-factory>
```

- Hibernate ne connaitra le **mapping** entre une classe et une table que si elle est déclarée dans la configuration.

# Insertion

Définissons une action **insertion** dans le contrôleur, avec le code suivant.

```
 TestsHibernate tstHiber = new TestsHibernate();
Pays monPays = new Pays();
monPays.setCode("is");
monPays.setNom("Islande");
monPays.setLangue("islandais");
tstHiber.insertPays(monPays);
maVue = VUES + "insertion.jsp";
```

# Sauvegarde

- On instancie et utilise un objet **Pays** comme n'importe quel **bean**.
- La capacité de l'objet est devenir persistant n'apparaît pas du tout.
- On peut l'utiliser comme un objet "métier", **transient** (donc non sauvegardé).
- Pour le sauvergarder on appelle la méthode **insertPays** de notre classe utilitaire :

```
session.beginTransaction();
session.save(pays);
session.getTransaction().commit();
```

- Il suffit donc de demander à la session de sauvegarder l'objet (dans le cadre d'une transaction) et le tour est joué.
- Hibernate se charge de tout : génération de l'ordre SQL correct, exécution de cet ordre, validation.
- Avec ce code on doit obtenir dans la console Java l'affichage de la requête SQL.
- On peut vérifier avec phpMyAdmin que l'insertion s'est bien faite.

# Lecture de données

Voyons maintenant comment lire des données de la base. Nous avons plusieurs possibilités.

- **Transmettre une requête SQL via Hibernate.** C'est la moins portable des solutions car on ne peut pas être sûr à 100% que la syntaxe est compatible d'un SGBD à un autre; il est vrai qu'on ne change pas de SGBD tous les jours.
- **Transmettre une requête HQL**. Hibernate propose un langage d'interrogation proche de SQL qui est transcrit ensuite dans le dialecte du SGBD utilisé.
- **Utiliser l'API Criteria**. Plus de langage externe (SQL) passé plus ou moins comme une chaîne de caractères : on construit une requête comme un objet. Cette interface a également l'avantage d'être normalisée en JPA.

JPA définit un langage de requête, JPQL (**Java Persistence Query Language**) qui est un sous-ensemble de HQL. Toute requête JPQL est une requête HQL, l'inverse n'est pas vrai.

# Table des pays

- Voici la requête qui parcours la table des pays, avec l'API **Criteria** :

```
public List<Pays> lecturePays() {
  Criteria criteria = session.createCriteria(Pays.class);
  return criteria.list();
  }
```

- Difficile de faire plus simple, on aurait même pu l'écrire en une seule ligne...
- Bien sûr pour des requêtes plus complexes, la construction est plus longue.
- Voici la méthode équivalente en HQL :

```
public List<Pays> lecturePaysHQL() {
  Query query = session.createQuery("from Pays");
  return query.list();
}
```

- C'est presque la même chose, mais on introduit une chaîne de caractères avec une expression qui n'est pas du java (et qui n'est donc pas contrôlée à la compilation).

# Gérer les associations

- un des aspects les plus importants du **mapping** entre la base de données et les objets java : la représentation des associations.
- Pour l'instant nous nous contentons de l'association (plusieurs à un) entre les films et les pays ("**plusieurs** films peuvent avoir été tournés dans **un seul** pays").
- En relationnel, nous avons donc dans la table **Film** un attribut **code_pays** qui sert de clé étrangère.

# Associations

```java
package modeles.webscope;

import javax.persistence.*;

@Entity
public class Film {

@Id
private Integer id;
public void setId(Integer i) {id = i;}

@Column
String titre;
public void setTitre(String t) {titre= t;}
public String getTitre() {return titre;}

@Column
Integer annee;
public void setAnnee(Integer a) {annee = a;}
public Integer getAnnee() {return annee;}

@ManyToOne
@JoinColumn (name="code_pays")
Pays pays;
public void setPays(Pays p) {pays = p;}
public Pays getPays() {return pays;}
}
```

# Associations (suite)

- Cette classe est incomplète : il manque le genre, le réalisateur, etc.
- Ce qui nous intéresse c'est le lien avec **Pays** qui est représenté ici :

```
@ManyToOne
@JoinColumn (name="code_pays")
Pays pays;
public void setPays(Pays p) {pays = p;}
public Pays getPays() {return pays;}
```

- On découvre une nouvelle annotation, **@ManyToOne**, qui indique à Hibernate que la propriété **pays**, instance de la classe **Pays**, encode un des côtés d'une association plusieurs-à-un entre les films et les pays.
- Pour instancier le pays dans lequel un film a été tourné, Hibernate devra donc exécuter la requête SQL qui, connaissant un film, permet de trouver le pays associé. Cette requête est :

```
select * from Pays where code = :film.code-pays
```

- La **clé étrangère** qui permet de trouver le pays est **code_pays** dans la table **Film**. C'est ce qu'indique la seconde annotation, **@JoinColumn**.

# Associations (suite)

- Ne pas oublier de modifier le fichier de configuration pour indiquer qu'une nouvelle classe est "mappée".
- toutes les informations permettant à Hibernate d'engendrer la requête qui précède ont bien été spécifiées par les annotations.
- Vous noterez que nous avons défini l'association du côté **Film** mais **pas** du côté **Pays**.
- Etant donné un objet pays, nous de pouvons pas accéder à la liste des films qui y ont été tournés.
- **L'association est uni-directionnelle**.

# Notions à retenir

- une application objet représente les données comme un **graphe d'objets**, liés par des références;
- une couche ORM transforme une base relationnelle en graphe d'objets et permet à l'application de **naviguer** dans ce graphe en suivant les références entre objets;
- la transformation est effectuée à la volée en fonction des navigations (accès) effectués par l'application; elle repose sur la génération de requêtes SQL;
- Hibernate est une implantation de la spécification JPA, et un peu plus que cela.