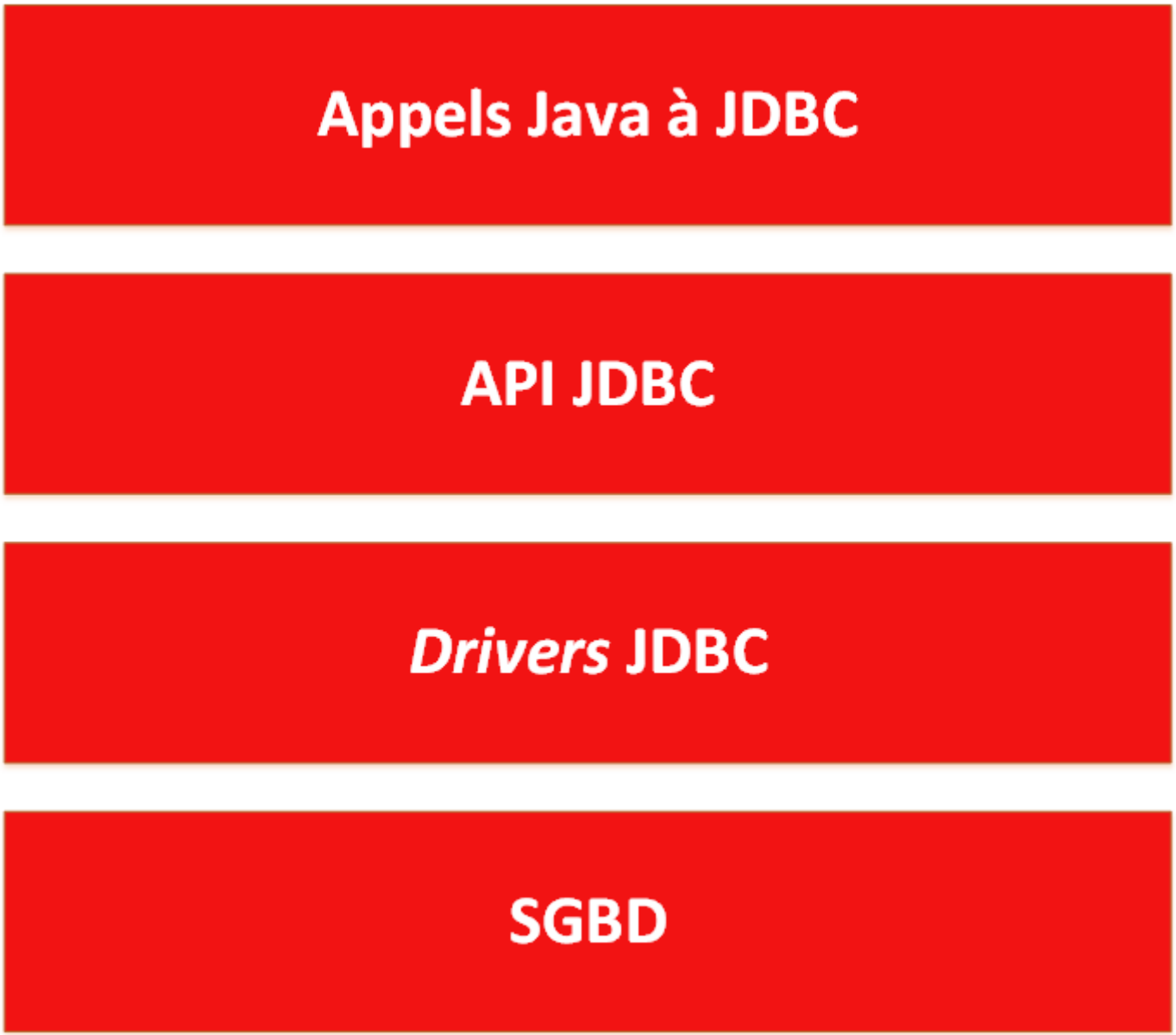
Génie Logiciel Appliqué

Gestion des bases de données

Yves Duchesne
yves@acceis.fr

- Les applications web complexes ont besoin d'utiliser un support de **stockage**
- Elles peuvent utiliser le **système de fichiers**, mais cela peut être rapidement **dangereux**, car on travaille **directement** sur le **système**
- Les bases de données fournissent un moyen plus **sécurisé**
  - **Authentification**
  - **Cloisonnement** applicatif
  - Offre un **modèle relationnel**, de **l'indexation**, etc.

- Java propose une API **native** pour interfacer une application avec une base de données
- Il s'agit de **JDBC** (*Java DataBase Connectivity*)
- JDBC est conçue de façon à pouvoir s'interfacer avec **tous les types** de bases de données (SGBD)
- Elle propose une interface **commune** et **agnostique** au SGBD, qui s'appuira sur des ***drivers*** pour communiquer avec le SGBD


Appels Java à JDBC
API JDBC
Drivers JDBC
SGBD

Gestion des bases de données

- Le *driver* est une couche chargée de **traduire** les appels **génériques** de JDBC dans le langage **spécifique** du SGBD
- Cette architecture permet **d'abstraire** le code produit d'une technologie particulière
- Le *driver* est spécifié à la **création** de la connexion à la base de données et il est possible, en spécifiant un autre *driver*, de rendre l'application compatible avec un nouveau SGBD
- C'est l'intérêt des **l'abstraction**

- La connexion à la base de données est effectuée en instanciant un objet de type **`java.sql.Connection`**
- Cette instance est obtenue en appelant la méthode **`getConnection()`** de la classe **`java.sql.DriverManager`**
- Il faut fournir à cette méthode l'URL qui devra être utilisée pour accéder à la base, sous la forme d'une chaîne de caractère au format précis, qui précise le **type** de SGBD et les **informations** utiles à la connexion

- Le format de cette URL est le suivant

```
jdbc:<type de SGBD>:<URL d'accès>
```

- Exemple : accès à une base de données avec les caractéristiques suivantes :
  - SGBD : MySQL
  - Adresse IP du serveur : 192.168.0.20
  - Port du service : 3306
  - Nom de la base : test
- L'URL sera la suivante

```
jdbc:mysql://192.168.0.20:3306/test
```

- Pour se connecter à cette base en utilisant JDBC, on doit alors spécifier les appels suivants :

```
import java.sql.Connection;
import java.sql.DriverManager;
String url = "jdbc:mysql://192.168.0.20:3306/test";
Connection conn = DriverManager.getConnection(url);
```

- L'object conn peut ensuite être utilisé pour interagir avec la base de données

- Pour exécuter une requête sur la base de données il faut utiliser des ***statements***
- Ils sont représentés par la classe **`java.sql.Statement`**
- Il s'agit d'une **interface** que l'on ne peut instancier à la volée
- Pour obtenir un objet de ce type, il faut utiliser la méthode **`createStatement()`** sur l'objet de type **`java.sql.Connection`**

- Il est ensuite possible d'exécuter des requêtes grâce à cet objet **`java.sql.Statement`**
- Plusieurs méthodes existent pour exécuter des requêtes selon leurs types
- Il y a principalement **deux types** de requêtes que l'on souhaitera effectuer sur une base de données
  - **Sélectionner des enregistrement** grâce à la méthode **`executeQuery()`**
  - **Effectuer une modification** grâce à la méthode **`executeUpdate()`**

- Ces deux méthodes utilisent des requêtes au format **`java.lang.String`**, c'est à dire sous forme de **chaînes de caractères** simples
- Les requêtes de sélection d'enregistrements dans la base de données retournent un objet de type **`java.sql.ResultSet`**, qui contient l'ensemble des enregistrements sélectionnés
- Les requêtes de mise à jour retournent simplement un **entier** qui correspond au nombre d'enregistrement modifiés

- L'objet **`java.sql.ResultSet`** représente un résultat de sélection d'enregistrements qui comporte plusieurs **lignes**, chacun représentant un **enregistrement.** Chaque enregistrement comporte plusieurs **colonnes**
- On itère grâce à la méthode **`next()`**, qui renvoie **`true`** tant qu'il reste des lignes à parcourir
- Des méthodes **typées** permettent d'obtenir les colonnes
  - **`getString(colonne)`**
  - **`getInt(colonne)`**

- Exemple de code de sélection d'enregistrements

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;
String url = "jdbc:mysql://192.168.0.20:3306/test";
Connection conn = DriverManager.getConnection(url);
Statement stmt = conn.createStatement();
String query = "SELECT * FROM users WHERE login='pierre'";
ResultSet resultat = stmt.executeQuery(query);
while (resultat.next()) {
      System.out.println(resultat.getString("PASSWORD"));
}
```

- Exemple de code de suppression d'enregistrements

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;
String url = "jdbc:mysql://192.168.0.20:3306/test";
Connection conn = DriverManager.getConnection(url);
Statement stmt = conn.createStatement();
String query = "DELETE FROM users WHERE login='pierre'";
int nb = stmt.executeUpdate(query);
System.out.println(nb + " enregistrement supprimés");
```

- Cette méthode est la plus **simple**, la plus **triviale** pour interagir avec la base de données
- Sur le plan de la sécurité, elle est aussi la plus **dangereuse**
- Si des données fournies par l'utilisateur sont insérées dans les requêtes (ce qui arrive **immanquablement**), il y a un risque fort d'**injection SQL**
- Cette utilisation est donc à **proscrire**

- Exemple de code vulnérable au sein d'une *servlet*

```
String login = request.getParameter("login");
String url = "jdbc:mysql://192.168.0.20:3306/test";
Connection conn = DriverManager.getConnection(url);
Statement stmt = conn.createStatement();
String query = "SELECT * FROM users WHERE login='"+login+"'";
ResultSet resultat = stmt.executeQuery(query);
while (resultat.next()) {
      System.out.println(resultat.getString("PASSWORD"));
}
```

  - Un attaquant peut injecter du SQL dans le paramètre **`login`**

- Il existe un moyen d'interagir avec la base de données de façon **plus sécurisée** en utilisant JDBC
- Il s'agit de l'utilisation des **requêtes préparées** (*prepared statements*)
- Ces requêtes sont exécutées en **deux temps**
  - Elles sont d'abord **envoyées** au SGBD pour être **compilées**
  - Les **paramètres** de la requête sont passées dans un **deuxième appel**
  - Cela évite les **injections**

- Pour utiliser des requêtes préparées avec JDBC, il faut utiliser la classe **`java.sql.PreparedStatement`**
- Elle est obtenue en appelant la méthode **`prepareStatement()`** de la classe **`java.sql.Connection`**
  - Cette méthode prend la requête préparée en paramètre, c'est à dire la requête **sans les paramètres valués**
  - Les valeurs des paramètres sont ensuite spécifiées
    - **`setInt()`**
    - **`setString()`**

- Exemple de code de sélection d'enregistrements

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
String url = "jdbc:mysql://192.168.0.20:3306/test";
Connection conn = DriverManager.getConnection(url);
String query = "SELECT * FROM users WHERE login=?";
PreparedStatement stmt = conn.prepareStatement(query);
stmt.setString(1, "pierre");
ResultSet resultat = stmt.executeQuery();
while (resultat.next()) {
      System.out.println(resultat.getString("PASSWORD"));
}
```

- Exemple de code de suppression d'enregistrements

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
String url = "jdbc:mysql://192.168.0.20:3306/test";
Connection conn = DriverManager.getConnection(url);
String query = "DELETE FROM users WHERE login=?";
PreparedStatement stmt = conn.prepareStatement(query);
stmt.setString(1, "pierre");
int nb = stmt.executeUpdate(query);
System.out.println(nb + " enregistrement supprimés");
```

- Il existe un **troisième moyen** d'interagir avec la base de données, également de **façon sécurisée**
- L'utilisation de **procédures stockées**
- Il s'agit de procédures **enregistrées en base** de données, qui correspondent à un **ensemble de requêtes**
- Ce sont des traitement **stéréotypés** enregistrés et **disponibles**
  - C'est comparable à une **méthode** dans une librairie

- Ces procédure stockées peuvent prendre des paramètres spécifiés par l'utilisateur appelant
- Exemple

```
CREATE PROCEDURE select_user
(IN nom CHAR(255))
BEGIN
    SELECT * FROM users WHERE login=nom;
END
```

- Pour appeler cette procédure, il faut utiliser la classe **`java.sql.CallableStatement`**
- Elle est obtenue en appelant la méthode **`prepareCall()`** de la classe **`java.sql.Connection`**
- Cette méthode prend directement **l'appel à la procédure stockée**, et les paramètres sont passés dans un second temps grâce à des méthode dédiées
  - **`setString()`**
  - **`setInt()`**

- Exemple : appel de **`select_user()`**

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.CallableStatement;
String url = "jdbc:mysql://192.168.0.20:3306/test";
Connection conn = DriverManager.getConnection(url);
String query = "{CALL select_user(?)}";
CallableStatement stmt = conn.prepareCall(query);
stmt.setString(1, "pierre");
ResultSet resultat = stmt.executeQuery();
while (resultat.next()) {
      System.out.println(resultat.getString("PASSWORD"));
}
```

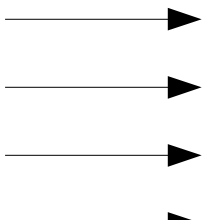
- En résumé :
- **Trois façons** d'exécuter des requêtes
- La méthode **naïve**, avec des requêtes simples envoyées sous forme de **chaînes de caractères**
  - **Pas sécurisées**
- Les **requêtes préparées**, qui permettent de spécifier les paramètres en **deux appels**
  - **Sécurisées**
- Les appels à **procédures stockées**
  - **Sécurisées**

- Ces méthodes permettent d'apporter de la **sécurité** à l'application
  - En évitant de former les requêtes SQL à l'aide **chaînes de caractères concaténées**
- Il existe cependant des outils apportant du **confort** au développeur et incluent des **fonctionnalités de sécurité**
- **Les Object Relational Mapper**
  - **ORM**

- La problématique de travail avec les bases de données en Java peut être **simplifié** en partant d'un constat simple
- Java : modèle **objet**
  - **Classes**
  - Subdivisées en **champs**
- Base de données : modèle **relationnel**
  - **Tables**
  - Subdivisées en **colonnes**
- Les ORM **exploitent** cette similitude

```
Class User {

...
    String nom;
    String prenom;
    Date naissance;
    int taille;
}
```



```
+----------------------+
|          User        |
+----------+-----------+
|nom       | char(255) |
|prenom    | char(255) |
|naissance | date      |
|taille    | int(11)   |
+----------------------+
```

- L'objectif est d'abord d'apporter une **fonctionnalité** au développeur
  - Travailler avec des **objets**
  - Plus **pratique** avec un langage fortement **orienté objet**, comme Java
  - Plus **propre** que de travailler avec des chaînes da caractères
- Il est donc préférable d'utiliser un ORM lorsqu'on travaille avec un SGBD
  - Il faut cependant en avoir **l'utilité**, ie : ne pas avoir un usage « marginal » du SGBD

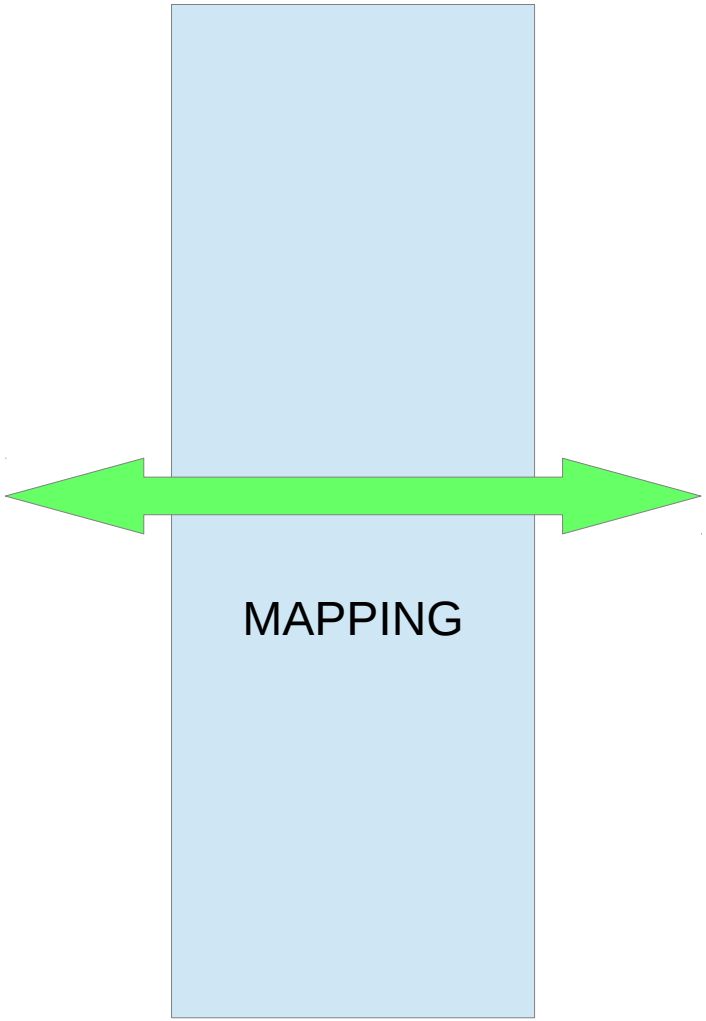
- En plus de cette **fonctionnalité** de « confort » pour le développeur, l'utilisation d'un ORM apporte de la **sécurité**
- Car les **requêtes** sont générées de façon sécurisées
  - Les paramètres sont bien « **nettoyés** »
- Car c'est **plus propre** de travailler avec des objets que des chaînes de caractères
- Car ça rend le code plus **lisible**, plus **compréhensible**, donc plus facile à **maintenir**

- En Java/J2EE, l'ORM historique, très utilisé, est **Hibernate**
- Il offre un e**nvironnement complet** de travail avec le SGBD
- Il s'appuie sur **JDBC**
  - Il est donc **compatible** avec tous les SGBD
- Il propose toutes les fonctionnalités sous forme **d'objets**

- La base de la configuration est le **mapping**

```
Class User {

...
    String nom;
    String prenom;
    Date naissance;
    int taille;
}
```




```
+----------------------+
|          User        |
+----------+-----------+
|nom       | char(255) |
|prenom    | char(255) |
|naissance | date      |
|taille    | int(11)   |
+----------------------+
```

- Le mapping permet de créer des **correspondances** entre le modèle **objet** et le modèle **relationnel**
  - Entre les tables et les objets
  - Entre les champs et les colonnes
- Il est déclaré au sein de **fichiers XML**
- On définit un fichier de mapping par classe
  - Ils portent l'extension **.hbm.xml**
- Hibernate se chargera de mettre en œuvre ces correspondances

- La balise de plus haut niveau est **`<hibernate-mapping>`**
- Elle permet de déclarer plusieurs classes grâce à des balises **`<class>`**
  - Attribut "**name**"
  - Attribut "**table**"
- Les champs de la classe sont ensuite déclarés au sein de balises **`<property>`**
  - Name
  - Column
  - Type

- Exemple de mapping

```
<hibernate-mapping>
    <class name="Utilisateur" table="User">
        <property name="nom" column="name" type="string">
        <property name="prenom" column="firstname" type="string">
        <property name="naissance" column="name" type="date">
        <property name="taille" column="height" type="int">
    </class>
<hibernate-mapping>
```

- Afin de fonctionner, en plus du *mapping*, Hibernate a besoin d'une **configuration générale**, afin de préciser plusieurs choses
  - **Nature** du SGBD
    - MySQL ?
    - Oracle ?
    - Etc.
    - + driver associé
  - **URL** de connexion à la base
  - **Identifiants** de connexion
    - Login
    - Mot de passe

- Cette configuration figure au sein du fichier **`hibernate.cfg.xml`**
- La balise de plus haut niveau est **`<hibernate-configuration>`**
- Balise **`<session-factory>`**
- Elle contient un ensemble de propriétés, présentées sous la forme de balises **`<property>`**
- Les fichiers de *mapping* y sont référencés grace à la balise **`<mapping>`**
  - Champ "**resource**"

- Chacune de ces propriétés va définir la configuration d'Hibernate
  - **hibernate.dialect**
    - Ex : org.hibernate.dialect.MySQLDialect
  - **hibernate.connections.driver_class**
    - Ex : com.mysql.jdbc.Driver
  - **hibernate.connection.url**
    - Ex : jdbc://mysql://10.0.16.32:3306/test
  - **hibernate.connection.username**
  - **hibernate.connection.password**

- Exemple de configuration

```
<hibernate-configuration>
    <session-factory>
        <property name="hibernate.dialect">
            org.hibernate.dialect.MySQLDialect
        </property>
        <property name="hibernate.connection.driver_class">
            com.mysql.jdbc.Driver
        </property>
        <property name="hibernate.connection.url">
            jdbc:mysql://10.0.16.32:3306/test
        </property>
        <property name="hibernate.connection.username">yves</property>
        <property name="hibernate.connection.password">topkek</property>
        <mapping resource="User.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

- Une fois la configuration effectuée, on peut **utiliser** Hibernate
- Il faut **récupérer** la "session factory" déclarée dans le fichier de configuration
  - **`org.hibernate.cfg.Configuration()`**
  - **`buildSessionFactory()`**
- On peut ensuite **ouvrir une session** sur le SGBD grâce à l'objet obtenu
  - **`openSession()`**
  - **`org.hibernate.Session`**

- Toutes les opérations effectuées par Hibernate se déroulent au sein d'une **Transaction**
  - C'est un concept qui permet de gérer les **accès concurrents** et **l'atomicité** des opérations
    - On effectue les opérations sur la transaction
    - Elles sont conservées en mémoire
    - Elles sont envoyées par l'appel à **`commit()`**
    - Elles sont annulées par l'appel à **`rollBack()`**
- Une transaction est créé sur la session
  - **beginTransaction()**
  - **org.hibernate.Transaction**

- Une fois que l'on dispose d'une transaction, on peut commencer à effectuer des **opérations**
  - Création
  - Suppression
  - Consultation
  - Modification
- On va travailler avec des objets Java
  - C'est l'intérêt d'un ORM !
- Ces objets **doivent** faire l'objet d'un *mapping* pour pouvoir être utilisés

- L'objet org.hibernate.Session met à disposition des **méthodes dédiées** aux opérations sur le SGBD
  - `save()`
  - `update()`
  - `delete()`
  - `list()`
- Attention : pour les opérations de modification, les changements ne **seront pas pris en compte** avant l'appel à Transaction.commit()

- Hibernate permet évidemment de créer des critères de recherche, de mise à jour, etc.
  - **`org.hibernate.Criteria`**
  - **`session.createCriteria(Class class)`**
- L'objet Criteria permet d'ajouter des critères de recherche
  - **`org.hibernate.Restrictions`**
    - eq
    - gt
    - lt
    - etc.

- Exemple : recherche de tous les utilisateurs

```
SessionFactory factory = new
Configuration().configure().buildSessionFactory();

Session session = factory.openSession();

Transaction tx = session.beginTransaction();

Criteria criteria =
session.createCriteria(User.class);

List<User> users = criteria.list();

for (User user : users) {

    System.out.println(user.getNom());

}

tx.commit();
```

- Exemple : création d'un utilisateur

```
SessionFactory factory = new
Configuration().configure().buildSessionFactory();

Session session = factory.openSession();

Transaction tx = session.beginTransaction();

User user = new User();

user.setNom("Duchesne");

user.setPrenom("Yves");

user.setTaille(187);

try {

    session.save(user);

} catch (Exception e) {

    tx.rollBack();

}

tx.commit();
```

- Attention : utiliser Hibernate ne garantit **pas** la sécurisation de votre application
- Il faut l'utiliser **correctement**
- En particulier, il propose le **HQL**, langage intermédiaire entre l'ORM et le SQL
- Cette utilisation n'apporte **pas** le même niveau de sécurité
  - **Injections HQL**
  - Voire **injections SQL** (cf. SSTIC)