

Hibernate

Cours n°3

Architecture Java EE - ENSG - M2 TSI

Clément BOIN
clement.boin@gmail.com

Persistance des objets

Problématique

- Comment faire en sorte que les **données** stockées en **base de données** correspondent exactement aux **objets** qui sont situés dans une **application** ?
 - persistance des objets

1ère solution : JDBC

```
public class MonExempleJDBC {  
  
    public static void main(String[] args) {  
        String url="jdbc:postgresql://localhost/Votrebase";  
        String user = "root";  
        String passwd = "password";  
        Class.forName("org.postgresql.Driver");  
  
        // Etape 2 Connexion à la base de données  
        Connection con=DriverManager.getConnection(url,user,passwd);  
  
        // Etape 3 - Création du curseur Statement  
        Statement stmt = con.createStatement();  
  
        // Etape 4 - Exécution de la requête  
        ResultSet rs = stmt.executeQuery("Select * from personne ");  
  
        // Etape 5 - Exploitation des résultats  
        while (rs.next()) {  
            int resultat1 = rs.getInt("num");  
            String resultat2 = rs.getString("nom");  
            System.out.println(resultat1 + " " + resultat2);  
        }  
        // Etape 6- Fermeture des flots mémoire  
        rs.close();  
        stmt.close();  
        con.close();  
    }  
}
```

Si l'on fait tout
avec JDBC, le code
deviendra vite très
long et répétitif.

Les solutions pour simplifier

Différentes solutions pour assurer la persistance :

- API standards :
 - ▢ JDO : Java Data Objects
 - ▢ EJB entity
 - ▢ JPA (Java Persistence API)
- designs patterns
 - ▢ DAO (Data Access Object)
- frameworks open source :
 - ▢ EclipseLink (ancien)
 - ▢ **Hibernate** (développé par JBoss)

Mapping objet-relationnel

- Mécanisme permettant de faire correspondre les attributs d'une relation (dans une base de données) avec les attributs d'un objet (en Java).
- Des **outils** permettent de faciliter la mise en œuvre du mapping Objet/Relationnel.
- Mise en œuvre plus ou moins complexe.
- **Réduire la quantité de code** à produire

La solution Hibernate ?

- permet de palier certaines faiblesses des API standards
- offre un cadre de travail assisté et plus rigoureux
- très populaire actuellement
- Hibernate a sa propre API et il implémente aussi celle de JPA (elles sont très proches)
 - ▢ On restera sur Hibernate dans ce cours
- Pour l'instant, pas de Spring !

Hibernate

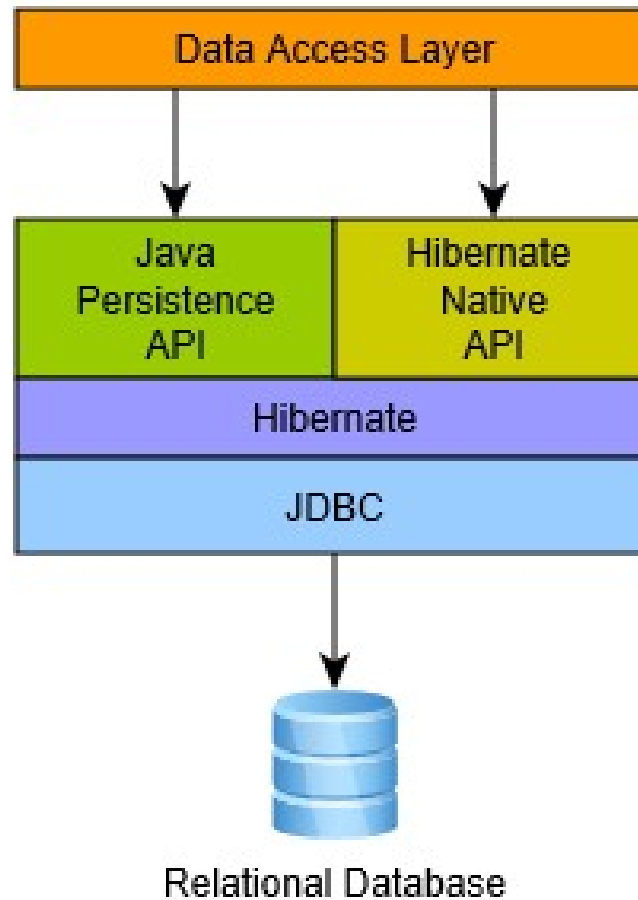
Hibernate

- **Framework ORM** (Object Relationnal Mapping)
 - persistance des objets en base de données relationnelle (mapping objet-relationnel)
 - logiciel libre
- Disponible sur **plusieurs architectures**
 - client lourd
 - web léger
 - Java EE
- Une solution relativement facile à mettre en œuvre
 - notamment grâce aux **annotations**

Architecture Hibernate

Source image :

[http://docs.jboss.org/hibernate/orm/5.4/userguide/html_single/Hibernate User Guide.html#architecture](http://docs.jboss.org/hibernate/orm/5.4/userguide/html_single/Hibernate%20User%20Guide.html#architecture)



Hibernate implémente les spécifications JPA (Java Persistence API)

Mapping avec les annotations

Exemple : La classe Participant

```
@Entity
@Table(name = "participant")
public class Participant {


    @Id
    @GeneratedValue(generator="increment")
    @GenericGenerator(name="increment", strategy="increment")
    @Column(name = "num_pers")
    private int numpers;

    @Column(name="nom", nullable = false)
    private String nom;

    @Column(name="prenom", nullable = false)
    private String prenom;

    @Column(name="email", nullable = false)
    private String email;

    @Column(name="date_naiss", nullable = false)
    private LocalDate datenaiss;
```



Utilisation des annotations Hibernate

Mapping avec les annotations

Remarques

- La norme JPA impose **un constructeur sans paramètre**

```
public Participant() {}
```

- Sans annotation `@Table` ou `@Column`, Hibernate utilise par défaut le nom de la classe ou du champs.
- Dans ce cours, nous utiliserons les annotations mais on peut aussi décrire les mappings avec des fichiers XML (comme avec Spring)

La clé primaire

- La clé primaire n'a pas de sens en programmation objet, elle est fixée par la base de données
 - Le constructeur sans paramètre est utilisé pas Hibernate pour fixer la clé primaire
 - L'autre constructeur, et les setter permettront à votre code d'interagir avec cet objet avec sa clé par défaut

```
public Participant() {} //Pour Hibernate
```

```
public Participant(String last_name, String first_name, String email, //Pour Java  
                  String birth_date, String organisation, String observations) {  
    this.last_name = last_name;  
    this.first_name = first_name;  
    this.email = email;  
    this.birth_date = birth_date;  
    this.organisation = organisation;  
    this.observations = observations;  
}
```

sessionFactory

- Un **cache** temporaire
 - maintient **pour certaines lignes** de la base de données **un unique objet Java** correspondant.
- contient une **connexion JDBC**
 - ne vit donc pas très longtemps
- L'utilisation de la classe *EntityManager* (JPA) est parfois préférée pour maintenir la connexion.

Exemple Hibernate

Mise en place d'une fabrique de sessions

- Fabrique des sessions Hibernate
 - une seule par application

Configuration d'après le fichier hibernate.cfg.xml

```
private static SessionFactory createSessionFactory() {  
    final StandardServiceRegistry registry =  
        new StandardServiceRegistryBuilder()  
            .configure()  
            .build();  
  
    try {  
        return new MetadataSources(registry)  
            .buildMetadata()  
            .buildSessionFactory();  
    } catch (Exception e) {  
        e.printStackTrace();  
        StandardServiceRegistryBuilder  
            .destroy(registry);  
    }  
    return null;  
}
```

Utilisation de la SessionFactory

```
public class ParticipantService {  
  
    private final SessionFactory sessionFactory;  
  
    public ParticipantService(SessionFactory sessionFactory) {  
        this.sessionFactory = sessionFactory;  
    }  
  
    public void insert(Participant participant) {  
        Session session = this.sessionFactory.openSession();  
        session.beginTransaction();  
        session.persist(participant);  
        session.getTransaction().commit();  
        session.close();  
    }  
  
    public List getAllParticipant(){  
        Session session = this.sessionFactory.openSession();  
        List result = session.createQuery("from Participant").list();  
        session.close();  
        return result;  
    }  
}
```


Création de la sessionFactory

(dans la classe principale en général)

```
SessionFactory sessionFactory =  
    createSessionFactory();
```

```
ParticipantService participantService =  
new ParticipantService(sessionFactory);
```

```
participantService.insertParticipant(new  
    Participant("toto", "tutu",  
        "toto@gmail.com", "2000-12-12"));
```

```
closeSessionFactory(sessionFactory);
```

Réutilisation d'une session + Gestion des exceptions Hibernate

```
Session session=sessionFactory.getCurrentSession();  
Transaction tx = null;
```

```
try {  
    tx= session.beginTransaction();  
  
    ...  
  
    tx.commit();  
} catch (HibernateException e) {  
    if (tx != null) tx.rollback();  
    throw e;  
}
```

- **getCurrentSession** essaie de réutiliser la même session
 - *Attention* : il ne faut pas qu'elle soit fermée

Le fichier de configuration

hibernate.cfg.xml

```
<hibernate-configuration>
  <session-factory>
    <!-- Database connection settings -->
    <property name="connection.driver_class">org.postgresql.Driver</property>
    <property name="connection.url">      jdbc:postgresql://localhost:5432/test </property>
    <property name="connection.username">test</property>
    <property name="connection.password">test</property>

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

    <!-- Lors de l'utilisation des annotations @ManyToOne et OneToMany, l'ordre a son importance -->
    <mapping class="fr.descartes.cvven.Participant" />
  </session-factory>
</hibernate-configuration>
```

Etat des objets pour la session

- **Transient:** Objet qui n'a jamais été dans une session et dont le champ @Id est à sa valeur par défaut ou qui a été supprimé de la session.
- **Persistent:** Objet qui est dans la session. Les modifications sur cet objet seront répercutées dans la BD (cf. slides suivants).
- **Detached:** Les objets qui ne sont plus dans l'EntityManager mais qui y ont été (hors suppression). En particulier, ces objets ont un champ @Id qui n'a plus sa valeur par défaut.

Actions possibles pendant une session

- La classe Session propose 4 méthodes principales :
 - **persist(Participant participant)**
 - ▢ crée une entrée dans la table avec les données de l'objet participant
 - ▢ participant doit être un POJO (jamais associé à une session, on dit **transient**)
 - ▢ le champ correspondant à la clé primaire est fixé à la valeur correspondant à la valeur de la ligne correspondante dans la BD
 - ▢ **Attention**, la norme garantie que le champ id du POJO est fixé après le commit
 - **get(Participant.class, id)**
 - ▢ renvoie l'objet Participant correspondant à la ligne de clé primaire id dans la table correspondant à la classe Participant.
 - **merge(Participant participant)**
 - ▢ prend un POJO dont l'id est fixé (typiquement un POJO renvoyé par get) et met la ligne correspondant de la table à jour avec les données de participant
 - **delete(Participant participant)**
 - ▢ prend un POJO dont l'id est fixé (typiquement un POJO renvoyé par get) et supprime la ligne correspondante

Requêtes

- en HQL
 - Proche du langage SQL
 - On travaille avec les classes mais pas de tables
 - Préserve les informations de typage
 - Nommé JPQL avec JPA

```
List<Participant> result =  
    session.createQuery("from Participant", Participant.class)  
        .getResultList();
```

- en SQL
 - on peut aussi faire du SQL plus classique
 - `createNativeQuery`

Requêtes préparées

- Permettent de limiter notamment les attaques par injection de SQL
- Simplifient l'écritures de certaines requêtes

Exemple :

```
String nom = "toto";  
Query<Participant> req=  
    session.createQuery("select * from Participant p  
                        where p.nom = :name",  
                        Participant.class);  
query.setParameter("name", nom);  
List<Participant> result = query.getResultList();
```