JPA : Java Persistent API

Farid AIT KARRA

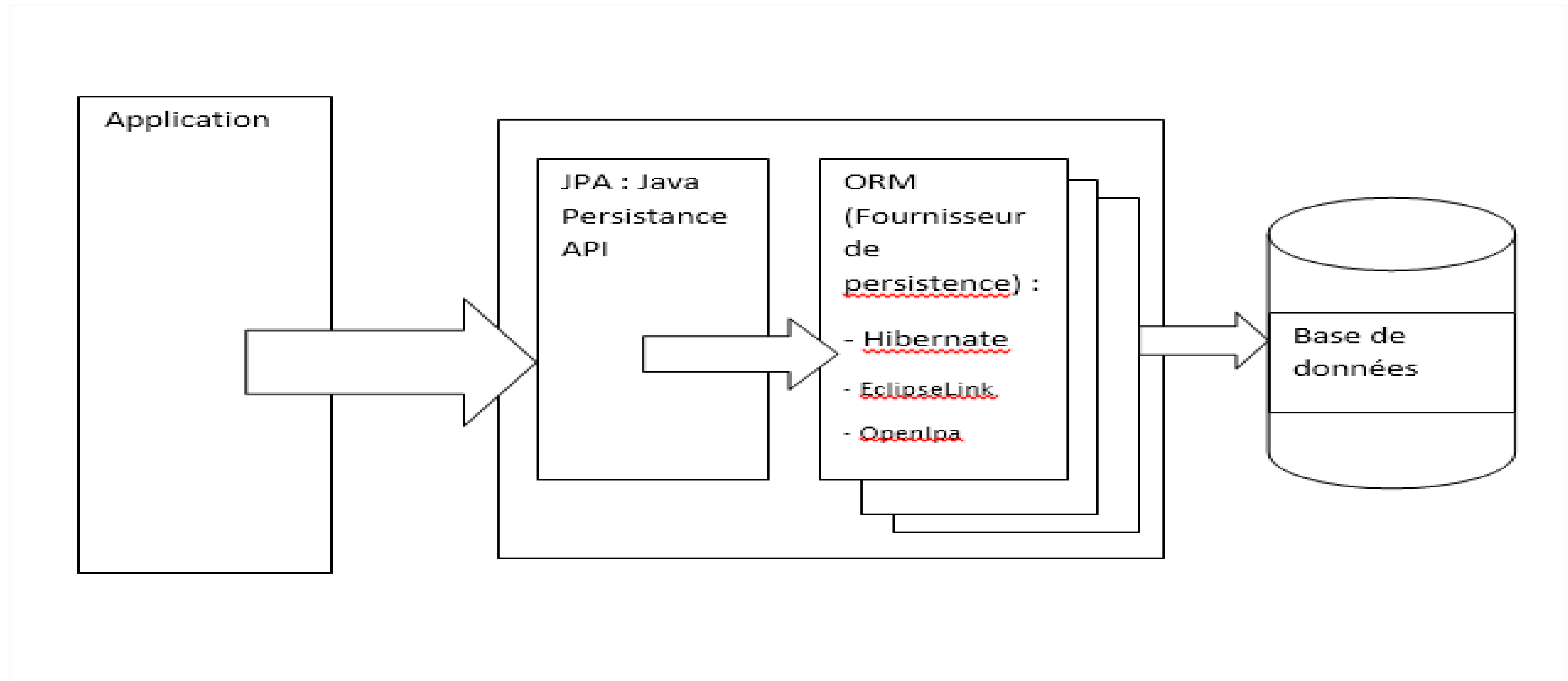
Copyright 2011 © ESUP Portail

Généralités

- **ORM**
  - Object-Relational Mapping = Mapping Objet-Relationnel
- **Les normes**
  - EJB 1 et 2
    - EJB Session et entité
    - Hibernate a une approche différente
  - EJB 3
    - Les EJB entité disparaissent au profit de JPA
    - JPA est très proche de Hibernate
    - Jboss fait partie du groupe de spécification de JPA
    - Hibernate est une des implémentations de JPA
      - C'est le choix de ESUP-Commons V2

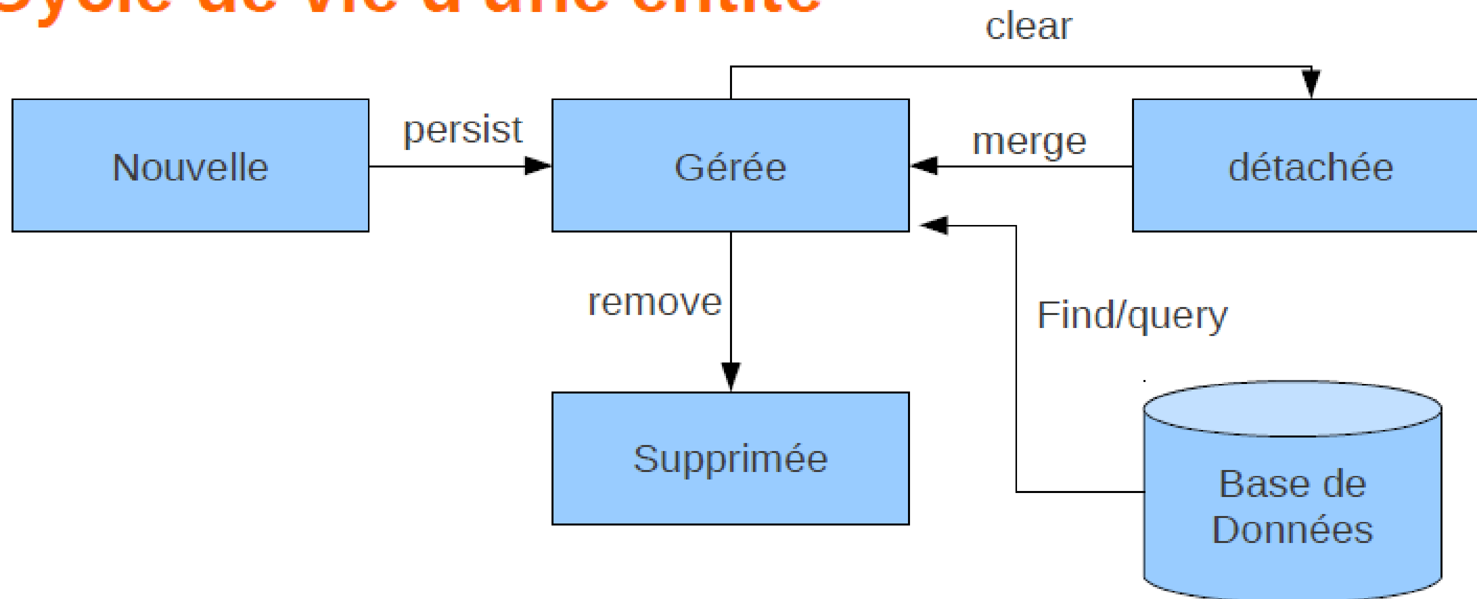
Généralités 2

Bien qu'étant à l'origine du standard JPA, Hibernate n'en est pas l'implémentation de référence. Hibernate propose le support de JPA en v1 JSR 220 du Java Community Process et v2 était le travail du Groupe d'experts JSR 317, et reste massivement utilisé aujourd'hui.



Cycle de vie

- **Notion de session du contexte de persistance**
  - En général maintenu le temps de la requête
    - A ne pas confondre avec la session applicative
  - En fin de session les instances gérées sont enregistrées en base
- **Cycle de vie d'une entité**



Mapping

- **Par annotation**
  - Pratique
    - http://www.objectdb.com/api/java/jpa/annotations
  - Sur les classes
    - @Entity
      - Précise que la classe est une entité à persister
  - Sur les champs
    - @Id
      - Clé primaire
    - @GeneratedValue
      - Valeur générée automatiquement
    - @Column
      - Permet de donner les caractéristiques de colonne qui stockera le champ (Ex : nom, null, taille, etc.)
    - @Transient
      - Le champ n'est pas persisté

Mapping 2

- **Les relations**
  - Annotations
    - @ManyToMany, @ManyToOne, @OneToMany, @OneToOne
  - Navigabilité
    - Une relation peut être unidirectionnelle ou bidirectionnelle
  - Lazy loading
    - Par défaut les enfants ne sont chargés que tardivement
      - Attention au cas où la session a pris fin. On peut avoir besoin de rattacher une entité à la nouvelle session (merge).
  - Cascade
    - En général on n'enregistre pas explicitement une entité (fait automatiquement en fin de session). De plus ses enfants (ajoutés, supprimés, modifiés) sont aussi enregistrés automatiquement.
      - cascade=ALL <--> cascade={PERSIST, MERGE, REMOVE, REFRESH, DETACH}

Mapping 3

- Les requêtes
  - Déclarées par annotation
    - ```
      @NamedQueries({
        @NamedQuery(
          name="tasksForUser",
          query="SELECT t FROM Task t WHERE t.owner.login = :userLogin"
        )
      })
      ```
  - Possibilité de passer des paramètres (:userLogin)
  - Possibilité de naviguer dans les objets (t.owner.login)
      ```

Persistence.xml

```
<persistence  version="2.0"
          xmlns="http://java.sun.com/xml/ns/persistence"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
                    http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
  <persistence-unit  name="jpa-exemple"  transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>org.univ.artois.bean.Personne</class>
    <properties>
      <property  name="javax.persistence.jdbc.url"  value="jdbc:derby://localhost:1527/test"/>
      <property  name="javax.persistence.jdbc.password"  value="jpa_pwd"/>
      <property  name="javax.persistence.jdbc.driver"  value="com.mysql.jdbc.Driver"/>
      <property  name="javax.persistence.jdbc.user"  value="jpa_user"/>
      <property  name="eclipselink.ddl-generation"  value="drop-and-create-tables"/>
    </properties>
  </persistence-unit>
</persistence>
```

Du simple POJO au BEAN

```
@Entity
public  class Personne {

@Id
@GeneratedValue(strategy = GenerationType.AUTO)
private Long id;

public Personne(){}


public Long getId() {
  return id;
}

public  void setId(Long id) {
  this.id = id;
}
// suivent les méthode toString(), equals() et hashCode()
}
```

Exemple d'utilisation

```
public  class TestJPA {
  public  static  void main(String... args) {
    EntityManagerFactory emf = Persistence.createEntityManagerFactory("jpa-exemple")
;
    EntityManager em = emf.createEntityManager() ;
    Personne personne =  new Personne();
    em.getTransaction().begin() ;
    em.persist(personne) ;
    em.getTransaction().commit() ;
    System.out.println("Id = " + personne.getId()) ;
  }
}
```

CRUD

- CRUD
  - ```
    public void addTask(Task task) {
      entityManager.persist(task);}
    ```
  - ```
    public Task getTask(long id) {
      Task task = entityManager.find(Task.class, id);
      return task;}
    ```
  - Rien à faire en update sauf si l'instance est détachée
  - ```
    public void deleteTask(Task task) {
      Task t = entityManager.find(Task.class, task.getId());
      entityManager.remove(t);}
    ```
    ```

CRUD 2

- Rattacher une instance
  - ```
    public Task updateTask(Task task) {
      return entityManager.merge(task);}
    ```
    - L'objet détaché est passé en paramètre
    - La fonction renvoie l'objet rattaché
- Rechercher des objets
  - ```
    public List<Task> getTasksForUser(User u) {
      Query q = entityManager.createNamedQuery("tasksForUser");
      q.setParameter("userLogin", u.getLogin());
      return (List<Task>)q.getResultList();
    }
    ```

Quelques bonnes pratiques

- **Clé métier != clé primaire**
  - Pour permettre une évolution simple du MPD
    - Définir une clé primaire auto-générée
    - Assurer la cohérence des données par une contrainte d'unicité sur la clé métier
- **hashCode() et equals()**
  - Si on stocke des objets en tant que List, Map ou Set alors il est requis d'implémenter hashCode() et equals() sur ces objets
  - hashCode() et equals() utilisent les éléments de la clé métier

??? Questions ???