Technologies des services du Web

Mikael Desertot

Université de Valenciennes
Institut des Sciences et Techniques
de Valenciennes

Persistance

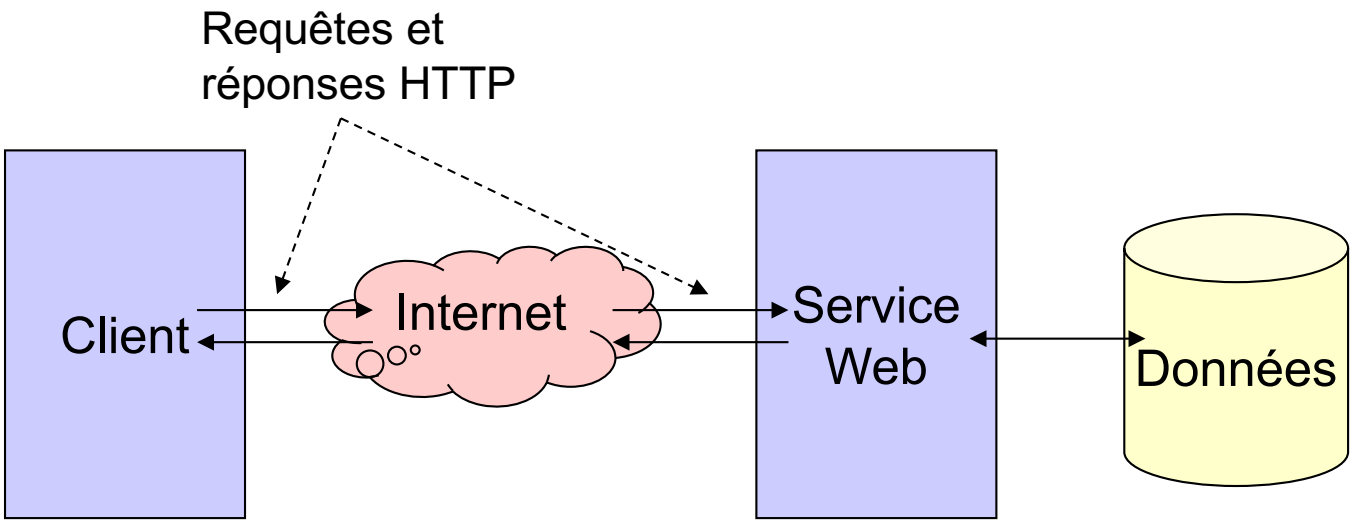
Utilisation des BDDs

- Incontournable
  - Toute application doit disposer d'un moyen d'accès aux données
    - Fiable
    - Persistant
    - Rapide

- Du SI d'entreprise…

…à votre téléphone

Exemple pour un site Web



JPA (1)

- Java Persistence API
- Atouts
  - "Léger"
  - Orienté POJO
- Spécification pour les EJB 3.0
  - 1 document pour l'introduction de nouvelles fonctionnalités dans EJB2.1
  - 1 document pour les beans sessions et message-driven
  - 1 document pour la persistence -> JPA

JPA (2)

- Le modèle se veut
  - Simple
  - Puissant
  - Flexible
- L'utilisation des POJO
  - Un objet applicatif peut être rendu persistant avec +/- 1 ligne de code !
  - Persistance définie uniquement par métadonnées
    - Annotations, XML

JPA (3)

- Persistance non-intrusive
  - L'API est un couche externe aux objets de persistance
  - Appel de l'API dans la logique métier
    - Objets à persister
    - Instructions à réaliser
- Requêtes objets
  - Langage JP-QL
  - Requêtes vers les objets et leurs relations sans forcément utiliser clés externes ou colonnes de la BDD

JPA (4)

- Entités mobiles
  - Architectures clients/serveurs
  - Un objet doit pouvoir "bouger" d'une machine virtuelle à l'autre
- Objets détachés/attachés
- Configuration simple
- Existe avec ou sans serveur d'application

JPA (5)

- Version actuelle
  - JPA 2.1
  - La version JPA 1 est encore employée et reste proche de la dernière version

- JPR
  - Principaux fournisseurs
    - Oracle, EclipseLink, Hibernate, OpenJPA,…

Persitence.xml (v2)

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0"
   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="EmployeeService"
                transaction-type="RESOURCE_LOCAL">
        <class>org.istv.jpa.Employee</class>
        <properties>
                <property name="javax.persistence.jdbc.driver"
                        value="org.hsqldb.jdbcDriver" />
                <property name="javax.persistence.jdbc.url"
                        value="jdbc:hsqldb:hsql://localhost/;create=true" />
                <property name="javax.persistence.jdbc.user" value="SA" />
                <property name="javax.persistence.jdbc.password" value="" />
        </properties>
   </persistence-unit>
</persistence>
```

D'un POJO...

```
public class Employee {

    private int id;
    private String name;
    private long salary;

    public Employee() {}

    public Employee(int id) {this.id = id;}

    public int getId() {return id;}

    public void setId(int id) {this.id = id;}

    public String getName() {return name;}

    public void setName(String name) {this.name = name;}

    public long getSalary() {return salary;}

    public void setSalary(long salary) {this.salary = salary;}
}
```

...vers un Entity

```
@Entity
public class Employee {
    @Id
    private int id;
    private String name;
    private long salary;

    public Employee() {}

    public Employee(int id) {this.id = id;}

    public int getId() {return id;}

    public void setId(int id) {this.id = id;}

    public String getName() {return name;}

    public void setName(String name) {this.name = name;}

    public long getSalary() {return salary;}

    public void setSalary(long salary) {this.salary = salary;}
}
```

Ajout d'un élément persistant

```
em.getTransaction().begin();
Employee emp = new Employee(id);
emp.setName(name);
emp.setSalary(salary);
em.persist(emp);
em.getTransaction().commit();
```

→Penser au contexte transactionnel

Recherche d'un élément

```
em.find(Employee.class, id);
```

```
Query query =
em.createQuery("SELECT e FROM Employee e");
Collection<Employee> ce =
      (Collection<Employee>) query.getResultList();
```

Suppression

```
Employee emp =
        em.find(Employee.class, id);

if (emp != null) {
   em.remove(emp);
}
```

Accès aux données

```
Employee emp = em.find(Employee.class, id);


if (emp != null) {
      emp.setSalary(emp.getSalary() + raise);
}
```

Tester la persistance

```
public class Test {

public static void main(String[] args) {
   EntityManagerFactory emf =
       Persistence
           .createEntityManagerFactory("EmployeeBDD");
       EntityManager em = emf.createEntityManager();

   em.close();
   emf.close();
```

Mapping Objet/Relationnel

Lien vers une table

- Annotation de la classe
- Attribut @Table

```
@Entity
@Table(name="EMP")
public class Employee { ... }
```

```
@Entity
@Table(name="EMP", schema="HR")
public class Employee { ... }
```

```
@Entity
@Table(name="EMP", catalog="HR")
public class Employee { ... }
```

Lien vers une colonne

- Annotation de l'attribut
- Attribut @Column

```
@Entity
public class Employee {
    @Id
    @Column(name="EMP_ID")
    private int id;
    private String name;
    @Column(name="SAL")
    private long salary;
    @Column(name="COMM")
    private String comments;
    // ...
}
```

Clé primaire

- Génération automatique
- Automatique

```
@Entity
public class Employee {
    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    private int id;
    // ...
}
```

- Par une table

```
@Id GeneratedValue(strategy=GenerationType.TABLE)
private int id;
```

Relations

- Exemple Many to one

- Exemple Many to Many

Mappings possibles

- Many to One
- One to Many
- One to One
- Many to Many

One to Many unidirectionelle

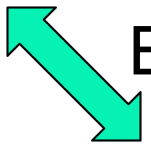
- Relation inverse de Many to One

```
@Entity
public class Department {
    @Id private int id;
    private String name;
    @OneToMany
    private Collection<Employee> employees;
    // ...
}
```

One to Many bidirectionelle

- Relation inverse de Many to One

```
@Entity
public class Department {
    @Id private int id;
    private String name;
    @OneToMany(mappedBy="department")
    private Collection<Employee> employees;
    // ...
}
```



Equivalent

```
@Entity
public class Department {
    @Id private int id;
    private String name;
    @OneToMany(targetEntity=Employee.class, mappedBy="department")
    private Collection employees;
    // ...
}
```

One To One (1)

```
@Entity
public class Employee {
    @Id private int id;
    private String name;
    @OneToOne
    @JoinColumn(name="PSPACE_ID")
    private ParkingSpace parkingSpace;
    // ...
}
```

One To One (2)

- Bi-directionnel

```
@Entity
public class ParkingSpace {
    @Id private int id;
    private int lot;
    private String location;
    @OneToOne(mappedBy="parkingSpace")
    private Employee employee;
    // ...
}
```

Many to Many

```
@Entity
public class Employee {
    @Id private int id;
    private String name;
    @ManyToMany
    private Collection<Project> projects;
    // ...
}

@Entity
public class Project {
    @Id private int id;
    private String name;
    @ManyToMany(mappedBy="projects")
    private Collection<Employee> employees;
    // ...
}
```

Requêtes JP-QL

Sélection simple

- Ca ressemble à du SQL
  - Tant mieux !

- L'idée est de parler objet

```
SELECT e
FROM Employee e
```

Sélection d'attributs

- Liste de noms d'employés

```
SELECT e.name
FROM Employee e
```

- Liste d'attributs à travers une relation

```
SELECT e.department
FROM Employee e
```

Requêtes paramétrables

- 2 types de déclarations supportées

```
SELECT e
FROM Employee e
WHERE e.department = ?1 AND
      e.salary > ?2
```

```
SELECT e
FROM Employee e
WHERE e.department = :dept AND
      e.salary > :base
```

Utilisation de requêtes statiques paramétrables

```
private static final String QUERY =
    "SELECT e.salary " +
    "FROM Employee e " +
    "WHERE e.department.name = :deptName AND " +
    "      e.name = :empName ";
```

```
public long queryEmpSalary(String deptName, String empName) {
    return (Long) em.createQuery(QUERY)
                    .setParameter("deptName", deptName)
                    .setParameter("empName", empName)
                    .getSingleResult();
}
```

Requêtes nommées (1)

- Généralement définie dans le composant Entity le plus proche de la requête

```
@NamedQuery(name="findSalaryForNameAndDepartment",
            query="SELECT e.salary " +
                  "FROM Employee e " +
                  "WHERE e.department.name = :deptName AND " +
                  "      e.name = :empName")
```

Requêtes nommées (2)

- Définition de plusieurs requêtes

```
@NamedQueries({
    @NamedQuery(name="Employee.findAll",
                query="SELECT e FROM Employee e"),
    @NamedQuery(name="Employee.findByPrimaryKey",
                query="SELECT e FROM Employee e WHERE e.id = :id"),
    @NamedQuery(name="Employee.findByName",
                query="SELECT e FROM Employee e WHERE e.name = :name")
})
```

Requêtes nommées (3)

- Exécution

```
public Employee findEmployeeByName(String name) {
     return (Employee) em.createNamedQuery("Employee.findByName")
                         .setParameter("name", name)
                         .getSingleResult();
}
```