Persistencia de objetos con JPA

Adolfo Sanz De Diego

Junio de 2011

# Persistencia de objetos

Acción de preservar la información de un objeto de forma permanente (**guardar**), para poder recuperar la información del mismo (**leer**) en un futuro.

En el caso de objetos la información que persiste son los valores que contienen los **atributos** en ese momento, no la funcionalidad de sus métodos.

# Técnicas de persistencia

**Serialización**:

* Objeto -> bytes -> (fichero, red, campo tabla bbdd, etc.) -> bytes -> Objeto
* Proceso manual.

**Motores de persistencia**:

* Proceso semi-autotico.

**Bases de datos orientadas a objetos**:

* Objeto -> Objeto bbdd -> Objeto
* Proceso automático, pero bbdddoo no han terminado de despegar.

# ORM

**O**bject-**R**elational **M**apping o **mapeo objeto/relacional**

Técnica para **persistir objetos en bases de datos relacionales**.

Persigue no perder las ventajas de la orientación a objetos al interactuar con una base de datos relacional.

# JPA

**J**ava **P**ersistence **A**PI.

API de persistencia desarrollada para la plataforma Java EE.

Se puede usar fuera de un contenedor Java EE.

Cubre tres áreas:

* La API en sí misma, definida en **javax.persistence**
* La **J**ava **P**ersistence **Q**uery **L**anguage (**JPQL**)
* Los **metadatos** para el mapeo objeto/relacional (**ORM**)

# Implementaciones JPA

JPA es una **especificación** (una serie de Interfaces).

Para utilizarla necesitas una **implementación** y los **drivers** de la bbdd.

Las implementaciones más populares son:

* **Hibernate** (el más conocido/usado y además LGPL)
* **TopLink**
* **EclipseLink**
* **OpenJPA**
* **Kodo**
* **DataNucleus**

# Entidades

Una entidad es un clase persistente.

Una entidad representa una **tabla** en la base de datos.

Cada instancia (objeto) de la entidad representa un **registro** en la base de datos.

# Requerimientos para las Entidades

La clase debe ser anotada con javax.persistence.**Entity**.

La clase debe ser pública o protegida, y en ningún caso final.

La clase debe tener al menos un constructor público o protegido sin argumentos.

La clase puede implementar la interface Serializable.

Una clase abstracta puede ser declarada como entidad.

Todas la entidades deben tener una anotación de PK (Primary Key)

Los atributos se deben declarar como privados.

Los atributos no se deben declarar como final.

Los atributos deben tener **getters/setters** públicos o protegidos.

Los atributos no persistentes deden de ser anotados con javax.persistence.**Transient**.

# Atributos persistentes

Tipos primitivos y sus Wrappers.

Matrices de tipos primitivos.

java.lang.String

java.math.BigInteger y java.math.BigDecimal

java.util.Date y java.util.Calendar

java.sql.Date, java.sql.Time y java.sql.TimeStamp

Clases serializables

Clases tipo Enum

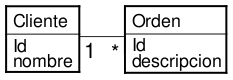
Otras entidades

Collecciones de todo lo anterior.

# Ejemplo: Entidad

import javax.persistence.\*;  
@Entity // obligatorio  
@Table(name = "PERSONAS")  
public class Persona {  
  
 @Id // obligatorio  
 @Column(name="ID\_PERSONA")  
 private int id;  
  
 @Column(name="NOMBRE") // opcional  
 private String nombre;  
  
 public void setID();  
 public String getID();  
  
 public void setNombre();  
 public String getNombre();  
}

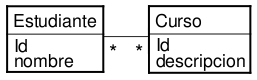
# Ejemplo: Uno a Muchos



@Entity public class Cliente {  
 ...  
 @OneToMany(cascade=CascadeType.ALL, mappedBy="cliente")  
 private Set<Orden> ordenes;  
 ...  
}

@Entity public class Orden {  
 ...  
 @ManyToOne  
 @JoinColumn (name="ID\_CLIENTE")  
 private Cliente cliente;  
 ...  
}

# Ejemplo: Muchos a Muchos



@Entity public class Estudiante {  
 @ManyToMany(cascade=CascadeType.ALL)  
 @JoinTable(name="CURSO\_ESTUDIANTE",  
 joinColumns={@JoinColumn(name="ID\_ESTUDIANTE",referencedColumnName="ID\_ESTUDIANTE")},  
 inverseJoinColumns={@JoinColumn(name="ID\_CURSO",referencedColumnName="ID\_CURSO")})  
 private List<Course> cursos;  
}

@Entity public class Curso {  
 @ManyToMany(cascade=CascadeType.ALL)  
 @JoinTable(name="CURSO\_ESTUDIANTE",  
 joinColumns=@JoinColumn(name="ID\_CURSO",referencedColumnName="ID\_CURSO"))  
 inverseJoinColumns=@JoinColumn(name="ID\_ESTUDIANTE",referencedColumnName="ID\_ESTUDIANTE"),  
 private List<Student> estudiantes;  
}

# Herencia

Se hace de 3 maneras distintas: 1. Una tabla por familia (comportamiento por defecto) 2. Un join de tablas 3. Una tabla por clase concreta

@Entity public class SuperClase {  
 @Id private Long id;  
 private int propiedadUno;  
 private String propiedadDos;  
 // Getters y Setters  
}

@Entity public class SubClase extends SuperClase {  
 @Id private Long id;  
 private float propiedadTres;  
 private float propiedadCuatro;  
 // Getters y setters  
}

# Herencia: una tabla

Comportamiento **por defecto**.

Las instancias de SuperClase y de SubClase son almancenadas en una **única tabla**.

Nombre por defecto de la tabla el de la clase raíz (SuperClase).

Dentro de esta tabla habrá:

* Una columna para el ID (válido para todas las entidades)
* Una columna para cada propiedad.
* Una columna discriminatoria que suele contener el nombre de la clase.

Las propiedades de las subclases no deben ser configuradas como not null pues da error.

# Herencia: un join de tablas

Cada clase y subclase será almacenada en su propia tabla.

La tabla raíz contiene una columna con un **ID compartido** y usado por todas las tablas, y una columna discriminatoria.

Cada subclase almacenará en su propia tabla únicamente sus atributos propios.

Las tablas de las subclase tendrán una **PK a la tabla raíz**.

Es un sistema intuitivo, pero si hay varios niveles de herencia se necesitarán varios JOIN, lo cual puede producir un impacto en el rendimiento.

@Entity  
@Inheritance(strategy = InheritanceType.JOINED)  
public class SuperClase { ... }

# Herencia: una tabla por clase

Cada entidad será mapeada a su propia tabla.

Con este sistema no hay tablas compartidas, columnas compartidas, ni columna discriminatoria.

Todas las tablas deberán **compartir el ID**.

Puede provocar problemas de rendimiento, ya que ante determinadas solicitudes, la base de datos tendrá que realizar múltiples JOIN.

@Entity  
@Inheritance(strategy = InheritanceType.TABLE\_PER\_CLASS)  
public class SuperClase { ... }

# El EntityManager

Administra la conexión con la base de datos.

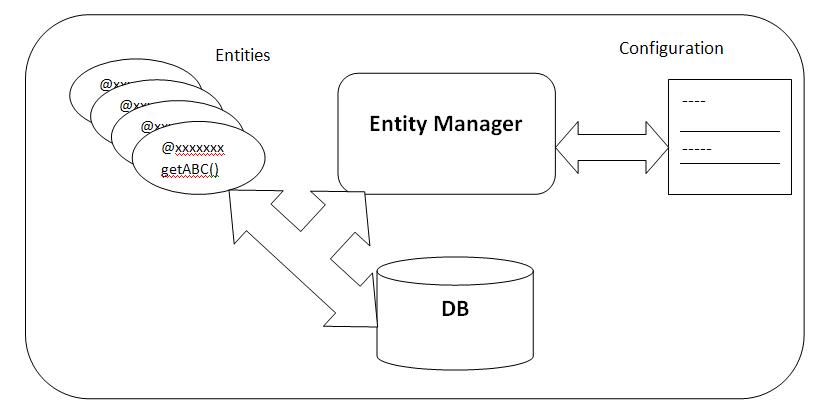
Administra la persistencia y la recuperación de entidades.

Puede ser administrada por un contenedor JEE o por la aplicación.

Puede administrar las transacciones o delegar en el contenedor JEE.

Soporta la ejecución de consultas (queries)

# Arquitectura JPA



# persistence.xml

<persistence  
 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"  
 version="2.0">  
 <persistence-unit name="ejemplo-unidad-persistencia" transaction-type="RESOURCE\_LOCAL">  
 <class>ejemplo.Cliente</class>  
 <class>ejemplo.Orden</class>  
 <class>ejemplo.Estudiante</class>  
 <class>ejemplo.Curso</class>  
 <properties>  
 <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>  
 <property name="javax.persistence.jdbc.user" value="usuario"/>  
 <property name="javax.persistence.jdbc.password" value="Pa$.w0rd"/>  
 <property name="javax.persistence.jdbc.url" value="jdbc:mysql://192.168.1.1/ejemplo"/>  
 <property name="hibernate.dialect" value="org.hibernate.dialect.MySQLDialect"/>  
 <property name="hibernate.show\_sql" value="true"/>  
 </properties>  
 </persistence-unit>  
</persistence>

# Ejemplo: persistir

EntityManagerFactory emf;  
EntityManager em;  
try {  
 emf = Persistence.createEntityManagerFactory("ejemplo-unidad-persistencia");  
 em = emf.createEntityManager();  
 Cliente cliente = new Cliente();  
 cliente.setNombre("Adolfo");  
 em.getTransaction().begin()  
 em.persist(cliente);  
 em.getTransaction().commit();  
} catch(Exception e) {  
 em.getTransaction().rollback()  
} finally {  
 if (em != null) em.close();  
 if (emf != null) emf.close();  
}

# Otros ejemplos

em.flush(); // fuerza a que los cambios pendientes se guarden en la base de datos

Cliente c = em.find(Cliente.class, id); // recupera cliente de la base de datos por ID

em.refresh(cliente); // recarga de la base de datos y sobreescribe en memoria

em.remove(cliente); // borra de la base de datos

# JPQL: una query

String jpql = "SELECT c FROM Cliente c";  
Query query = em.createQuery(jpql);  
List<Cliente> clientes = query.getResultList();  
for(Cliente c : clientes) {  
 // ...  
}

# JPQL: query con parámetros

String jpql = "SELECT c FROM Cliente c WHERE c.nombre = ?1 AND c.id > ?2";  
Query query = em.createQuery(jpql);  
query.setParameter(1, "Pepito");  
query.setParameter(2, 468);  
List<Cliente> clientes = query.getResultList();  
for(Cliente c : clientes) {  
 // ...  
}

String jpql = "SELECT c FROM Cliente c WHERE c.nombre = :nombre AND c.id > :id";  
Query query = em.createQuery(jpql);  
query.setParameter("nombre", "Pepito");  
query.setParameter("id", 468);  
List<Cliente> clientes = query.getResultList();  
for(Cliente c : clientes) {  
 // ...  
}