JPA

Aplicaciones Distribuidas

Contenido

- Introducción
- Mapeo de Objetos
- Ejemplos
- Identidad
- Lectura Inmediata y Perezosa
- Enumerados
- Transient
- Colecciones
- Tipos insertables
- Tipos de acceso
- Asociaciones
- Herencia
- EntityManager
- Configuración
- Transacciones
- Persistir Entidades
- Leer Entidades
- Modificar Entidades
- EliminarEntidades
- JPQL

JPA

- ***Java Persistence API***
  - Estándar Java para el mapeo objeto relacional
  - Otros mapeadores (Hibernate, Toplink, EclipseLink) orientados al mapeo con JPA → portabilidad
- Arquitectura:
  - Entidades
  - Identificadores
  - Mapeo O/R
  - EntityManager
  - Consultas
  - Persistence.xml

Mapeo de Objetos

- Entidad es la unidad básica en JPA
- Toda entidad debe:
  - Tener un constructor por defecto
  - Ser una clase de primer nivel (no interna)
  - No ser final
  - Implementar la interface java.io.Serializabe si es accedida remotamente

- Metadatos de mapeo:
  - Anotaciones en la propia clase (preferido)
  - Ficheros de mapeo xml externos
    - Elegir depende de preferencias tanto personales como del proyecto.

- Mapeo XML
  - Ventaja:
    - No es necesario recompilar nuestro código para cambiar la configuración de mapeo
  - Inconvenientes:
    - Exige mantener un archivo externo.

- Anotaciones
  - Ventaja:
    - Permite tener en un mismo lugar el código Java y sus instrucciones de comportamiento
  - Inconvenientes:
    - Exige una recompilación cada vez que deseamos cambiar el comportamiento del mapeo
    - Resultar en código menos portable (otra aplicación no-JPA que use nuestras entidades deberá incluir todas las librerías JPA en su classpath para poder compilar correctamente).

- Cuando ambas opciones son utilizadas al mismo tiempo, la configuración XML prevalece sobre las anotaciones

Ejemplo

```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@Entity
public class Pelicula {
    @Id
    @GeneratedValue
    private Long id;
    private String titulo;
    private int duracion;

    // Getters y Setters
}
```

- Ejemplo:

```
@Entity
@Table(name="PET_INFO")
public class Pet {
        @Id
        @Column(name="ID")
        int licenseNumber;
        String name;
        PetType type;
        @ManyToOne
        @JoinColumn(name="OWNER_ID")
        Owner owner;
        ...
}
```

```
@Entity
public class Owner {
        @Id
        int id;
        String name;
        @Column(name="PHONE_NUM")
        String phoneNumber;
        @OneToOne
        Address address;
        @OneToMany(mappedBy="owner")
        List<Pet> pets;
        ...
}
```

Identidad

- Entidades poseen una identidad que las diferencie del resto
  - propiedad marcada con la anotación **@Id**
  - debe admitir valores null, p.e.: Integer en lugar de int
- La identidad de una entidad va a ser gestionada por el proveedor de persistencia
  - propiedad de identidad anotada: **@GeneratedValue**

```
@Entity
public class Pelicula {
    @Id
    @GeneratedValue
    private Long id;
```

- JPA aplica a las entidades que maneja una configuración por defecto, de manera que una entidad es funcional con una mínima cantidad de información: @Id, @Entity

- Se pueden modificar los valores por defecto:

```
@Entity
@Table(name = "TABLA_PELICULAS")
public class Pelicula {
    @Id
    @GeneratedValue
    @Column(name = "ID_PELICULA")
    private Long id;
    ...
}
```

Lectura inmediata y perezosa

- JPA nos permite leer una propiedad desde la base de datos la primera vez que un cliente intenta leer su valor (lectura perezosa), en lugar de leerla cuando la entidad que la contiene es creada (lectura inmediata)
  - Si la propiedad nunca es accedida, nos evitamos el coste de crearla (objetos gran tamaño)

```
@Basic(fetch = FetchType.LAZY)
private Imagen portada;
```

- El comportamiento por defecto de la mayoría de tipos Java es lectura inmediata.
- Uso explícito:

```
@Basic(fetch = FetchType.EAGER)
private Imagen portada;
```

- Solo los objetos de gran tamaño y ciertos tipos de asociación deben ser leidos de forma perezosa
  - Si, por ejemplo, marcamos todas las propiedades de tipo int, String o Date de una entidad con lectura perezosa, entonces se provoca que se efectúen multitud de llamadas a la base de datos, cuando con solo una (al crear la entidad en memoria) podrían haberse leido todas con apenas coste

- **@Basic** solo puede ser aplicado a:
  - tipos primitivos y sus correspondientes clases wrapper (Integer, BigDecimal, …)
  - Date
  - arrays
  - algunos tipos del paquete java.sql
  - enums
  - cualquier tipo que implemente la interface Serializable
- @Basic admite el atributo **optional**
  - Permite valor null

  @Basic(optional=false)

Enumerados

- JPA puede mapear los tipos enumerados (enum) mediante la anotación Enumerated:

```
@Enumerated
private Genero genero;
```

- Mapea cada valor ordinal de un tipo enumerado a una columna de tipo numérico en la base de datos.

```
public enum Genero { TERROR, DRAMA, COMEDIA, ACCION }
```

- Se inserta valor 2 para Genero.COMEDIA (valor ordinal)

- si en el futuro introducimos un nuevo tipo de genero en una posición intermedia, o reordenamos las posiciones de los géneros, nuestra base de datos contendrá valores erróneos
  - forzar a la base de datos a utilizar una columna de texto

```
@Enumerated(EnumType.STRING)
private Genero genero;
```

Transient

- Propiedades de una entidad que pueden no representar su estado

```
@Entity
public class Persona {
    @Id
    @GeneratedValue
    private Long id;
    private String nombre;
    private String apellidos
    private Date fechaNacimiento;
    private int edad;

    …// getters y setters
}
```

```
@Transient
private int edad;
…
public int getEdad() {
    // calcular la edad y devolverla
}
```

Colecciones

- Una entidad puede tener propiedades de tipo java.util.Collection y/o java.util.Map que contengan tipos básicos
- Los elementos de estas colecciones serán almacenados en una tabla diferente a la que contiene la entidad donde están declarados
  - El tipo de colección tiene que ser concreto (como ArrayList o HashMap) y nunca una interface

```
private ArrayList comentarios;
```

- Podemos cambiar el mapeo por defecto:

```
@ElementCollection(fetch = FetchType.LAZY)
@CollectionTable(name = "TABLA_COMENTARIOS")
private ArrayList comentarios;
```

- Si la coleción es de tipo Map se puede añadir la anotación @MapKeyColumn

```
@MapKeyColumn(name = "NOMBRE_COLUMNA”)
```

  - permite definir el nombre de la columna donde se almacenarán las claves del Map

Tipos Insertables

- Los tipos insertables (**@Embeddable**) son objetos que no tienen identidad, por lo que para ser persistidos deben ser primero insertados dentro de una entidad
- Cada propiedad del tipo insertable será mapeada a la tabla de la entidad que lo contenga, como si fuera una propiedad declarada en la propia entidad

```
@Embeddable
public class Direccion {
    private String calle;
    private int codigoPostal;
    ...
}
```

```
@Entity
public class Persona {
    ...
    @Embedded
    private Direccion direccion;
}
```

Tipo de Acceso

- **JPA permite definir dos tipos de acceso:**
  - **Acceso a variable (Field access)**
    - **Si las anotaciones están en los atributos**
  - **Acceso a propiedad (Property access)**
    - **Si las anotaciones de mapeo están en los métodos getter**
- Se pueden producir errores…

```
@Entity
public class Entidad
    @Id
    @GeneratedValue
    private Long id;
    @Embedded
    private Insertable insertable;
}
```

```
@Embeddable
public class Insertable {
    private int variable;

    @Column(name = "VALOR_DOBLE")
    public int getVariable() {
        return variable * 2;
    }

    public void setVariable(int variable) {
        this.variable = variable;
    }
}
```

- Solución:

```
@Embeddable
@Access(AccessType.PROPERTY)
public class Insertable { ... }
```

- También se puede declarar:

```
@Access(AccessType.FIELD)
```

Asociaciones

- Para mapear colecciones de entidades, debemos usar asociaciones: @OneToOne, @OneToMany, @ManyToOne, @ManyToMany
  - Asociaciones unidireccionales
  - Asociaciones bidireccionales
- **@OneToOne**:
  - Ejemplo unidireccional

```
@Entity
public class Cliente {
    @Id
    @GeneratedValue
    private Long id;
    @OneToOne
    private Direccion direccion;

    // Getters y setters
}
```

```
@Entity
public class Direccion {
    @Id
    GeneratedValue
    private Long id;
    private String calle;
    private String ciudad;
    private String pais;
    private Integer codigoPostal;

    // Getters y setters
}
```

- Cliente contiene *una* referencia a una entidad Direccion (contiene la anotación @OneToOne)
  - Cliente es el *dueño* de la relación de manera implícita, y por tanto cada entidad de este tipo contendrá por defecto una columna adicional en su tabla correspondiente de la base de datos
  - Cada entidad será almacenada en su propia tabla, añadiendo a la tabla donde se almacenán los clientes (la dueña de la relación) una columna con las claves foraneas necesarias.
  - Personalizar columna:

```
@OneToOne
@JoinColumn(name = "DIRECCION_FK")
private Direccion direccion;
```

- **@OneToMany**

```
@Entity
public class Cliente {
    @Id
    @GeneratedValue
    private Long id;
    @OneToMany
    private List direcciones;

    // Getters y setters
}
```

- JPA utilizará por defecto una *tabla de unión* (join table)
  - las tablas donde se almacenan ambas entidades contienen una clave foranea a una tercera tabla con dos columnas
  - Personalizar tabla de unión:

```
@OneToMany
@JoinTable( name = …,
            joinColumn = @JoinColumn(name = …),
            inverseJoinColumn = @JoinColumn(name = …))
private List direcciones;
```

- **@OneToOne**
  - Ejemplo bidireccional
    - ambos lados de la relación deben estar anotados con **@OneToOne**, pero ahora uno de ellos debe indicar de manera explícita que la parte contraria es dueña de la relación:
      - **mappedBy** en la anotación de asociación de la parte no-dueña
- **@MantToMany**: igual que anteriores…

```
@Entity
public class Marido {
    @Id
    @GeneratedValue
    private Long id;
    @OneToOne(mappedBy = "marido")
    private Mujer mujer;
}
```

```
@Entity
public class Mujer {
    @Id
    @GeneratedValue
    private Long id;
    @OneToOne
    private Marido marido;

    // Getters y setters
}
```

Tipo de lectura en asociaciones

- El tipo de lectura por defecto para las relaciones uno-a-uno y muchos-a-uno es inmediata (eager).
- El tipo de lectura para los dos tipos de relaciones restantes (uno-a-muchos y muchos-a-muchos), es perezosa (lazy)
- Modificar:

```
@OneToMany(fetch = FetchType.EAGER)
private List pedidos;
```

Ordenación en asociaciones

- Ordenar los resultados devueltos por una asociacion mediante la anotación @OrderBy:

```
@OneToMany
@OrderBy("nombrePropiedad asc")
private List pedidos;
```

Herencia

- JPA permite gestionar la forma en que los objetos son mapeados cuando en ellos interviene el concepto de herencia
  - Una tabla por familia (comportamiento por defecto)
  - Unión de subclases
  - Una tabla por clase concreta

- Por defecto es *una tabla por familia* (familia son las subclases que están relacionadas por herencia con una clase padre)

- **Una tabla por familia:**
- Todas las clases que forman parte de una misma familia son almacenadas en una única tabla (nombre de la clase padre).
  - En esta tabla existe una columna por cada atributo de cada clase y subclase de la familia
  - Además de una columna adicional donde se almacena el tipo de clase al que hace referencia cada fila

- Se puede hacer explícito:

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
public class SuperClass { ... }
```

…y personalizar:

```
@Entity
@Inheritance
@DiscriminatorColumn(name = "...", discriminatorType = CHAR)
@DiscriminatorValue("C")
public class SuperClase { ... }

@Entity
@DiscriminatorValue("S")
public class SubClase extends SuperClase { ... }
```

- **Unión de subclases**
  - Cada clase y subclase (sea abstracta o concreta) será almacenada en su propia tabla

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

  - La tabla raíz contiene una columna con una clave primaria usada por todas las tablas, así como la columna discriminatoria
  - Cada subclase almacenará en su propia tabla únicamente sus atributos propios (nunca los heredados), así como una clave foránea que hace referencia a la clave primaria de la tabla raíz

- **Una tabla por clase concreta**
  - Cada entidad será mapeada a su propia tabla (incluyendo todos los atributos propios y heredados)
  - No hay tablas compartidas, columnas compartidas, ni columna discriminatoria.

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

EntityManager

- EntityManager es la clase principal del API
  - Crear entidades
  - Crear consultas que devuelven entidades
  - Actualizar y borrar entidades
  - etc…
- Dos tipos de EntityManger
  - Container-Managed
    - Proporcionado por Service Provider Interface de JPA (mediante inyección de dependencia)
  - Non-Managed
    - Desde cualquier implementación JPA, usando factorías

- Ejemplo:
  - Non-Managed

```
import javax.persistence.*;
...
EntityManagerFactory emf = Persistence.createEntityManagerFactory("PetShop");
EntityManager em = emf.createEntityManager();
...
em.close();
```

  - Container-Managed

```
@Stateless
public class MyBean implements MyInterface {
    @PersistenceContext(unitName="PetShop")
    EntityManager em;
    ... }
```

Configuración

- META-INF/persistence.xml
- Unidad de persistencia
  - Configuración de acceso a un origen de datos con JPA
  - Requiere nombre para ser referenciado desde programación
  - Tipos
    - Managed Container (referenciado mediante JNDI)

```
<persistence-unit name="PetShop">
    <jta-data-source>jdbc/PetShopDB</jta-data-source>
</persistence-unit>
```

    - Non-Managed (continua…)

```
<persistence-unit name="PetShop">
    <class>com.acme.petshop.Pet</class>
    <class>com.acme.petshop.Owner</class>
    <properties>
        <property name="javax.persistence.jdbc.driver"
                  value="oracle.jdbc.OracleDriver"/>
        <property name="javax.persistence.jdbc.url"
                  value="jdbc:oracle:thin:@localhost:1521:XE"/>
        <property name="javax.persistence.jdbc.user" value="scott"/>
        <property name="javax.persistence.jdbc.password" value="tiger"/>
    </properties>
</persistence-unit>
```

Transacciones

- El concepto de transacción representa un contexto de ejecución dentro del cual podemos realizar varias operaciones como si fuera una sola:
  - o todas ellas son realizadas satisfactoriamente o el proceso se aborta en su totalidad

- Dos tipos de transaccionalidad
  - **JTA container**
    - Usa técnicas de transaccionalidad del contenedor (p.e. transaccionalidad del contenedor EJB 'required')

    <persistence-unit name="AADDPersistencia" transaction-type="JTA">
  - **Resource Local**
    - Indicar explícitamente la transaccionalidad mediante programación.
    - EntityTransaction → begin, commit, rollback

    <persistence-unit name="AADDPersistencia" transaction-type="RESOURCE_LOCAL">

Estados entidades

- Una entidad puede estar en uno de los dos estados siguientes:
  - **Managed** (gestionada)
    - Todos los cambios que efectuemos sobre ella dentro del contexto de una transacción se verán reflejados también en la base de datos, de forma transparente para la aplicación.
  - **Detached** (separada)
    - Los cambios realizados en la entidad no están sincronizados con la base de datos.
    - Una entidad se encuentra en estado separado antes de ser persistida por primera vez, y cuando tras haber estado gestionada es separada de su contexto de persistencia

Persistir entidades

```
EntityManagerFactory emf =
        Persistence.createEntityManagerFactory(”AADDPersistencia");
EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();

Pelicula pelicula = new Pelicula();
pelicula.setTitulo("Pelicula uno");
pelicula.setDuracion(142);

tx.begin();
try {
    em.persist(pelicula);
    tx.commit();
} catch(Exception e) {
    tx.rollback()
}

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

- La sincronización puede no ocurrir hasta que instamos a la transacción para completarse (tx.commit())
  - Depende la implementación concreta del proveedor de persistencia que usemos
- Forzar a que cualquier cambio pendiente
  - **flush**() de EntityManager
- Sincronización en sentido inverso, actualizando una entidad con los datos que actualmente se encuentran en la base de datos
  - **refresh()** de EntityManager

Leer entidades

- JPA permite leer una entidad previamente persistida en la base de datos para construir un objeto Java:
  - **Obteniendo un objeto real**
    - Pelicula p = **em.find**(Pelicula.class, id);
    - Lectura inmediata de la instancia
  - **Obteniendo una referencia** a los datos persistidos
    - Pelicula p = **em.getReference**(Pelicula.class, id);
    - Lectura perezosa de la instancia

Actualizar entidades

- Una instancia managed puede ser modificada y su sincronización es automática
- Se puede separar una o todas las entidades gestionadas actualmente por el contexto de persistencia mediante los métodos **detach**() y **clear**()
  - los cambios en su estado dejan de ser sincronizados con la base de datos
  - **em.persist**() lanzará una excepción
  - **em.merge**() pasa de nuevo a managed

Eliminar entidades

- La entidad es eliminada de la base de datos y separada del contexto de persistencia.
  - Sin embargo, la entidad seguirá existiendo como objeto Java

```
em.remove(pelicula);
pelicula.setTitulo("ya no soy una entidad, solo un objeto Java normal");
```

- Cuando existe una asociación uno-a-uno y uno-a-muchos entre dos entidades no eliminan en cascada, por defecto

- Indicar:

```
@OneToOne(orphanRemoval = true)
private Descuento descuento;
```

Operaciones en cascada

- Uso del atributo cascade:

```
@OneToOne(cascade = CascadeType.REMOVE)
private Descuento descuento;
```

```
CascadeType.PERSIST
        .REMOVE
        .MERGE
        .REFRESH
        .DETACH
        .ALL
```

- Configurar varias operaciones en cascada de la lista superior usando un **array** de constantes **CascadeType**

```
@OneToOne(cascade = {
        CascadeType.MERGE,
        CascadeType.REMOVE,
        })
private Descuento descuento;
```

JPQL

- Lenguaje similar a SQL orientado a objetos, trabaja con entidades JPA
- Pequeñas diferencias con HQL (p.e.: requiere SELECT en la proyección, UPDATE y DELETE, …)
- Tres tipos de sentencias:

  1. select_clause from_clause [where_clause] [groupby_clause] [having_clause] [orderby_clause]
  2. update_clause [where_clause]
  3. delete_clause [where_clause]

- Soporta las funciones:
  - Agregados: AVG, MAX, MIN, SUM,COUNT
  - Cadena: CONCAT, SUBSTRING, TRIM, LOWER, UPPER
  - Numericas: LENGTH, ABS, SQRT, MOD, SIZE
  - Fecha: CURENT_DATE, CURRENT_TIME