



Frameworks de persistencia - JPA

Sesión 3 - Mapeado de tablas, Bean Validation



Índice

- Introducción
- Mapeado de tipos
- Objetos embebidos
- Clave primaria compuesta
- Herencia
- *Bean Validation*



Mapeo de entidades

- Por defecto una entidad se mapea con una tabla en la base de datos con el mismo nombre
- El esquema de la base de datos se define en la unidad de persistencia
- Es posible modificar estos valores por defecto con la anotación `@Table` y los elementos `name` y `schema`

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



Mapeo de columnas

- Por defecto un atributo de una entidad se mapea en una columna con el mismo nombre
- Es posible modificar esto con la anotación `@Column` y el elemento `name`
- Otros elementos: `nullable`, `length`

```
@Entity
public class Empleado {
    @Id
    @Column(name="EMP_ID")
    private int id;
    private String nombre;
    @Column(name=SAL, nullable="false")
    private Long sueldo;
    @Column(name=COM, length=40)
    private String comentario;
    // ...
}
```



Mapeo de tipos

- JPA realiza un mapeo automático de los tipos Java de las entidades a tipos SQL
- El tipo SQL asociado con el tipo Java viene definido por el proveedor de persistencia, pudiendo variar de un gestor de base de datos a otro
- La especificación JPA define una lista de tipos que podemos usar en los atributos de las entidades



Posibles tipos Java

- JPA realiza un mapeo automático de los tipos Java de las entidades a tipos SQL
- El tipo SQL asociado con el tipo Java viene definido por el proveedor de persistencia, pudiendo variar de un gestor de base de datos a otro
- La especificación JPA define una lista de tipos que podemos usar en los atributos de las entidades
- **Tipos primitivos Java:** byte, int, short, long, boolean, char, float, double
- **Clases *wrapper* de los tipos primitivos:** Byte, Integer, Short, Long, Boolean, Character, Float, Double
- **Arrays de bytes y char:** byte[], Byte[], char[], Character[]
- **Tipos numéricos largos:** java.math.BigInteger, java.math.BigDecimal
- **Strings:** java.lang.String
- **Tipos temporales de Java:** java.util.Date, java.util.Calendar
- **Tipos temporales de JDBC:** java.sql.Date, java.sql.Time, java.sql.Timestamp
- **Tipos enumerados:** cualquier tipo enumerado del sistema o definido por el usuario
- **Objetos serializables:** cualquier tipo serializable del sistema o definido por el usuario



LOBs

- LOBs (Large Objects): tipos especiales en las BD que permiten almacenar objetos de gran tamaño
- Dos tipos
 - CLOBs: caracteres (texto, por ejemplo el contenido de una página HTML)
 - BLOBs: binarios (por ejemplo, una imagen)
- Anotación `@Lob` para indicar este tipo de dato:
 - `String`, `char[]` y `Character[]`: se mapea con un CLOB
 - `byte[]`, `byte[]` y `Serializable`: se mapea con un BLOB

```
@Entity
public class Empleado {
    @Id private int id;
    @Basic(fetch=FetchType.LAZY)
    @Lob @Column(name="PIC")
    private byte[] foto;
    // ...
}
```



Tipos enumerados

- Es posible usar tipos enumerados y definir el modo de mapearlos en la BD
- Un ejemplo de tipo enumerado y su uso en una entidad
- Por defecto se mapea en una columna de enteros y se asocia cada valor de la enumeración a un entero
- Es posible hacer que en el mapeo se conviertan los valores en Strings, utilizando la anotación `@Enumerated(EnumType.STRING)` junto al tipo enumerado



Ejemplo de tipo enumerado

```
public enum TipoEmpleado {  
    EMPLEADO_TIEMPO_PARCIAL,  
    EMPLEADO_TIEMPO_COMPLETO,  
    EMPLEADO_EXTERNO  
}
```

```
@Entity  
public class Empleado {  
    @Id private int id;  
    private TipoEmpleado tipo;  
    // ...  
}
```

```
@Entity  
public class Empleado {  
    @Id private int id;  
    @Enumerated(EnumType.STRING)  
    private TipoEmpleado tipo;  
    // ...  
}
```



Tipos temporales

- Es posible mapear los siguientes tipos temporales Java del paquete `java.sql`:
`java.sql.Date`, `java.sql.Time` y `java.sql.Timestamp`
- Se mapean en los correspondientes tipos SQL
- También se pueden mapear los tipos: `java.util.Date` y `java.util.Calendar`
- Hay que indicar en qué tipo SQL se mapean con la anotación `@Temporal` y la especificación del tipo: `TemporalType.DATE`, `TemporalType.TIME` o `TemporalType.TIMESTAMP`



Ejemplo de tipos temporales

```
@Entity
public class Empleado {
    @Id private int id;
    @Temporal(TemporalType.DATE)
    private java.util.Date fechaNacimiento;
    @Temporal(TemporalType.TIMESTAMP)
    private java.util.Date horaSalida;
    // ...
}
```



Mapeo explícito del tipo

- Es posible definir explícitamente el tipo de la base de datos al que hacer la conversión utilizando el atributo `@columnDefinition` de la anotación `@Column`:

```
@Column(name="duracion", columnDefinition="SMALLINT")  
int duracion;
```



Estado transitorio

- Una atributo se marca con el modificador transient cuando no se quiere mapear en la BD

```
@Entity
public class Empleado {
    @Id private int id;
    private String nombre;
    private Long sueldo;
    transient private String traduccion;
    // ...

    public String toString() {
        if (traduccion == null) {
            traduccion =
                ResourceBundle.getBundle("EmpResources").getString("Empleado");
        }
        return traduccion + ": " + id + " " + nombre;
    }
}
```



Mapeo del identificador

- El identificador de la entidad se mapea con la clave primaria de la tabla
- El atributo debe ser de uno de los siguientes tipos:
 - **Tipos Java primitivos:** byte, int, short, long, char
 - **Clases wrapper de tipos primitivos:** Byte, Integer, Short, Long, Character
 - **Arrays de tipos primitivos o de clases wrappers**
 - **Cadenas:** java.lang.String
 - **Tipos numéricos grandes:** java.math.BigInteger
 - **Tipos temporales:** java.util.Date, java.sql.Date



Generación automática del identificador

- Es posible descargar la responsabilidad de generar el identificador único de una entidad en JPA
- Se pueden definir cuatro estrategias: AUTO, TABLE, SEQUENCE o IDENTITY en la que se utilizan valores enumerados del tipo GenerationType
- La más sencilla es AUTO, que se basa en el sistema de generación de claves primarias de la BD

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



Objetos embebidos

- Es posible definir atributos que se van mapear en más de una columna, utilizando objetos embebidos

```
@Embeddable
public class Direccion {
    private String calle;
    private String ciudad;
    private String provincia;
    @Column(name="COD_POSTAL")
    private String codigoPostal;
    // ...
}

@Entity
public class Empleado {
    @Id private int id;
    private String nombre;
    private Long sueldo;
    @Embedded private Direccion direccion;
    // ...
}
```



Características de los objetos embebidos

- Los objetos embebidos no son entidades
- Es una forma muy cómoda de implementar con JPA una especie de tipos de datos definido por el usuario en SQL
- Las instancias de los objetos embebidos no pueden compartirse por más de una entidad (no son entidades)



Recuperación perezosa

- El comportamiento por defecto de JPA cuando carga un objeto de la BD es cargar todos sus atributos (*eager fetching*, en inglés)
- Es posible marcar ciertos atributos que tienen gran tamaño con la característica de carga perezosa (*lazy fetching*) utilizando el elemento `fetch=FetchType.LAZY` de la anotación `@Basic`
- JPA sólo recupera el atributo de la BD cuando se accede a él

```
@Entity
public class Empleado {
    // ...
    @Basic(fetch=FetchType.LAZY)
    @Column(name=COM)
    private String comentario;
    // ...
}
```



Clave primaria compuesta

- Es posible mapear tablas con clave primaria compuesta
- Varias estrategias, todas ellas pasan por crear una clase auxiliar con los atributos de la clave primaria compuesta

<<Table>> USUARIOS
VARCHAR USUARIO <<PK>> VARCHAR DEPTO <<PK>> ...



Clase auxiliar UsuarioId

- Clase auxiliar UsuarioId

```
@Embeddable
public class UsuarioId implements Serializable {
    private String username;
    private String departamento;

    ...

    public boolean equals(Object o) {
        if (o != null && o instanceof UsuarioId) {
            Id that = (Id) o;
            return this.username.equals(that.username) &&
                this.departamento.equals(that.departamento);
        } else {
            return false;
        }
    }

    public int hashCode() {
        return username.hashCode() + departamento.hashCode();
    }
}
```



Entidad

<<Table>> USUARIOS
VARCHAR USUARIO <<PK>>
VARCHAR DEPTO <<PK>>
...

```
@Entity
@Table(name = "USUARIOS")
public class Usuario {
    @Id
    @AttributeOverrides({
        @AttributeOverride(name="username", column=@Column(name="USUARIO")),
        @AttributeOverride(name="depto", column=@Column(name="DEPTO"))
    })
    private UsuarioId usuarioId;
    ...
}
```

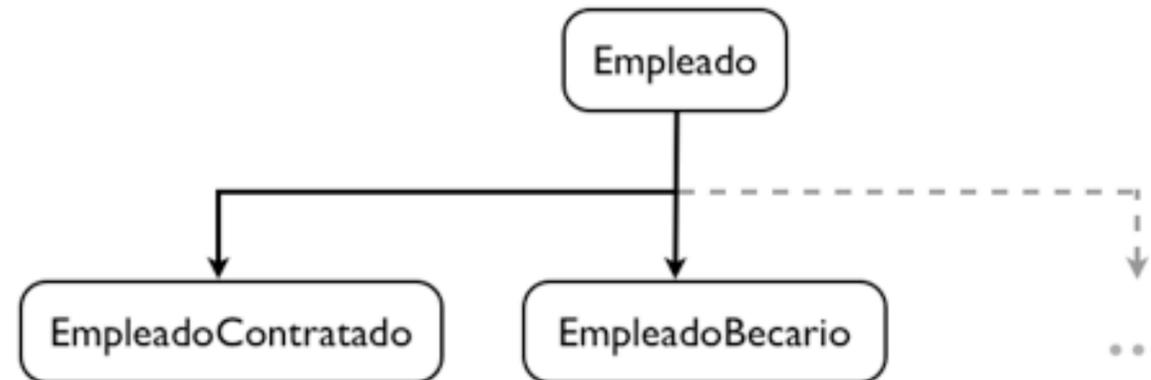


Herencia

- La herencia es una característica fundamental del modelo OO que no existe en el modelo relacional
- Formas de mapeo:
 - Tabla única
 - Tablas join
 - Una tabla por clase
- El mapeo más sencillo es el de tabla única: todos los registros de la jerarquía de clases van a una misma tabla. El tipo de objeto se indica con una columna discriminante.



Ejemplo



ID	Nombre	Salario	Tipo	PlanPensiones	SeguroMedico
1	Antonio	2.300	Contrato	230	NULL
2	Juan	1.200	Beca	NULL	150
3	María	2.400	Contrato	240	NULL



Código (1)

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="Tipo", discriminatorType=DiscriminatorType.STRING)
public abstract class Empleado {
    ...
}
```

```
@Entity
@DiscriminatorValue(value="Contrato")
public class EmpleadoContratado extends Empleado {
    private Long planPensiones;

    public Long getPlanPensiones() {
        return planPensiones;
    }

    public void setPlanPensiones(Long planPensiones) {
        this.planPensiones = planPensiones;
    }
}
```



Código (2)

```
@Entity
@DiscriminatorValue(value="Beca")
public class EmpleadoBecario extends Empleado {
    private Long seguroMedico;

    public Long getSeguroMedico() {
        return seguroMedico;
    }

    public void setSeguroMedico(Long seguroMedico) {
        this.seguroMedico = seguroMedico;
    }
}
```

```
Empleado emp = new EmpleadoContratado();
emp.setId(id);
emp.setNombre(nombre);
emp.setSueldo(sueldo);
emp.setPlanPensiones(sueldo/10);
```



Bean Validation

- Incluido en Java EE a partir de 5.0 en la especificación [JSR 349](#). El API se encuentra en el paquete `javax.validation.constraints`.
- Se usa para definir restricciones que deben cumplir los valores de atributos en memoria
- El API define anotaciones de validación predefinidas y métodos que realizan la validación y devuelven una colección de atributos que no cumplen la restricción
- Es posible definir nuevas validaciones
- En la integración con JPA no hace falta que llamemos nosotros a los métodos de validación, los llama el proveedor de persistencia (Hibernate en nuestro caso) en el momento de creación, modificación y borrado y lanza una excepción si no se cumple alguna restricción



Ejemplos de validaciones

Anotación	Ejemplo
@DecimalMax	@DecimalMax("30.00") BigDecimal descuento;
@Digits	@Digits(integer=6, fraction=2) BigDecimal precio;
@Future	@Future Date eventoFuturo;
@NotNull	@NotNull String username;
@Pattern	@Pattern (regexp="\\(\\d{3}\\)\\d{3}-\\d{4}") String numeroTelefono;
@Size	@Size(min=2, max=240) String mensajeCorto;



Ejemplo

```
@Entity
public class Empleado {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private int idEmpleado;

    @NotNull
    @Size(max = 25)
    @Column(nullable = false)
    private String nombre;

    @NotNull
    @Size(max = 25)
    @Column(nullable = false)
    private String apellidos;

    @NotNull
    @DecimalMax(value = "50000.00")
    @Column(nullable = false)
    private Double sueldo;
}
```



Validación con JPA

- En el momento de creación, modificación o borrado de la entidad se lanza una excepción unchecked de tipo `javax.validation.ConstraintViolationException`

```
public class PeliculaServicio {  
  
    public void actualizaRecaudacionPelicula(  
        Long idPelicula,  
        Double recaudacion) {  
  
        EntityManager em = EmfSingleton.getInstance().createEntityManager();  
        EntityTransaction tx = em.getTransaction();  
        try {  
            tx.begin();  
            PeliculaDao daoPelicula = new PeliculaDao(em);  
            Pelicula pelicula = daoPelicula.find(idPelicula);  
            pelicula.setRecaudacion(recaudacion);  
            daoPelicula.update(pelicula);  
            tx.commit();  
        } catch (Exception ex) {  
            tx.rollback();  
            String mensaje = "Error al actualizar la pelicula "  
                + id + " con la recaudación " + recaudacion;  
            logger.error(mensaje, ex);  
            throw new FilmotecaException(mensaje, ex)  
        } finally {  
            em.close();  
        }  
    }  
}
```



Configuración Maven

- Librerías necesarias para JPA en aplicaciones Java SE, en aplicaciones Java EE las provee el propio servidor de aplicaciones

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-validator</artifactId>
  <version>5.1.3.Final</version>
</dependency>
<dependency>
  <groupId>javax.el</groupId>
  <artifactId>javax.el-api</artifactId>
  <version>2.2.4</version>
</dependency>
<dependency>
  <groupId>org.glassfish.web</groupId>
  <artifactId>el-impl</artifactId>
  <version>2.2</version>
</dependency>
```



Configuración JPA - persistence.xml

- Basta con añadir una línea en el persistence.xml
- Modos:
 - AUTO: Valida si el JAR de validación se encuentra en el classpath
 - CALLBACK: Valida siempre, error si el JAR de validación no se encuentra
 - NONE: No se utiliza Bean Validation

```
<persistence version="2.1"
  xmlns="http://xmlns.jcp.org/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence">

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

    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
    <class>org.expertojava.jpa.mensajes.modelo.Autor</class>
    <class>org.expertojava.jpa.mensajes.modelo.Mensaje</class>

    <validation-mode>CALLBACK</validation-mode>

    <properties>

      <!-- JPA properties -->

      <!-- Hibernate properties -->

    </properties>
  </persistence-unit>
</persistence>
```



¿Preguntas?