JPA

Persistencia en Bases de datos Relacionales

Aplicaciones Distribuidas

Curso 2025/2026

Introducción

- Necesitamos acceder a las bases de datos relacionales desde las aplicaciones que usan lenguaje orientado a objetos.
  - Ejecutar consultas SQL.
  - Convertir filas y columnas en objetos.
- Puede requerir mucho trabajo de programación.

JDBC

- API Java que permite ejecutar consultas SQL en Bases de Datos Relacionales.
- Utiliza un driver para acceder a la base de datos.
- Permite establecer una conexión a la base de datos, ejecutar una consulta y procesar los resultados.

```
//...
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM REVISTA
                                  WHERE ISSN = '123456'");
Revista r = new Revista();
r.setIssn(rs.getString("ISSN"));
r.setTitulo(rs.getString("TITULO"));
r.setFechaPublicacion(rs.getDate("FECHAPUBLICACION"));
r.setNumPaginas(rs.getInt("NUMPAGINAS"));
r.setPrecio(rs.getFloat("PRECIO"));
//...
```

Desventajas con JDBC

- Hay que estar constantemente convirtiendo las filas y columnas del modelo relacional a objetos Java.
- Proceso tedioso y engorroso.
- ¿No se podría usar el modelo de objetos de la aplicación para acceder a la base de datos?

Java Persistence API (JPA)

- Mapeo Objeto-Relacional (ORM):
  - Mapear conceptos de un modelo en el otro.
  - La transformación se hace de forma automática y transparente.
- Eliminar el salto existente entre las bases de datos relacionales (modelo relacional) y la orientación a objetos.
- En JPA los objetos son POJOs.
- El mapeo se define a través de metadatos:
  - Se mapean objetos del modelo orientado a objetos con entidades del modelo relacional.
  - Para ello los objetos se persisten, se les da una entidad, se crean, actualizan y eliminan dentro de transacciones.

Creación de una entidad

- Una clase Java se convierte en una entidad JPA por medio de anotaciones.
- Se anota con `javax.persistence.Entity`.
- Debe tener un constructor sin argumentos.

```java
@Entity
public class Empleado{
    @Id
    private int id;
    private String nombre;
    public Empleado() {}
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }
    public String getNombre() { return nombre; }
    public void setNombre(String name) { this.nombre = nombre; }
}
```

Mapeo por anotaciones

`@Entity` : anota la clase para indicar al motor de persistencia que es una entidad.
`@Id` : anota el campo o propiedad que representa la identidad de la entidad (clave primaria).
`@Table(name="nombreTable")` : especifica el nombre de la tabla en la base de datos que se mapea con la clase anotada.
`@Column` : anota un atributo para indicar propiedades específicas de la columna en la base de datos, por ejemplo `name` .

Mapeo de tipos simples

- La mayoría de tipos se mapean de forma directa:
  - int, char, String, boolean, double, float, long, Integer, Float, Double...

Lectura inmediata y perezosa

- Todos los tipos básicos se recuperan de forma inmediata.
- Podemos controlar esto con la opción `fetch` de la anotación `@Basic`.
- Si sabemos con antelación que ciertas porciones de una entidad van a ser accedidas muy raramente se pueden anotar para que tengan lectura perezosa.

```
@Entity
public class Empleado implements Serializable{
    @Id
    private int id;
    private String nombre;
    @Basic(fetch=FetchType.LAZY)
    private String comentarios
}
```

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}
```

- En este ejemplo se insertaría el valor `2` para `Genero.COMEDIA` (valor ordinal).

Enumerados

- 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.
- Es preferible forzar a la base de datos a utilizar una columna de texto:

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

Objetos grandes

- Las cadenas de carácteres u objetos binarios que pueden tener gran tamaño necesitan una anotación especial para indicar que se acceden de forma especial.
- Se usa la anotación `@Lob` para este tipo de atributos.
- Sin la anotación cualquier String se mapea al tipo `VARCHAR(255)`.

```
@Entity
@Table(name="editorial")
public class Editorial implements Serializable {
    @Lob
    private String descripcion;
    //...
}
```

Tipos temporales

- JPA soporta los tipos temporales:
  - `java.sql.(Date, Time y Timestamp)`
  - `java.time.(LocalDate,LocalTime,LocalDateTime)`
  - `java.util.(Date,Calendar)`
- Los de tipo `java.sql` y `java.time` no necesitan anotación, se mapean de forma directa.
- Los de tipo `java.util` requieren la anotación `@Temporal` y el tipo `TemporalType (DATE, TIME, TIMESTAMP)`.

Estado Transient

- Las entidades en Java pueden tener ciertos atributos que no queramos que se persistan. Esto se consigue con la anotación `@Transient`.

```java
@Entity
@Table(name="editorial")
public class Editorial implements Serializable {
    @Column(name ="fecha_fundacion", columnDefinition = "DATE")
    private LocalDate fechaFundacion;
    @Transient
    private long anyos;
    //...
}
```

- En el ejemplo, el atributo `anyos` es calculado y depende de `fechaFundacion`, por lo que no es necesario persistirlo.

Mapeo de Clave Primaria

- Además de la notación `@Id`, en ocasiones podemos necesitar que los valores de identificación se generen automáticamente y no tener que preocuparnos por la unicidad de los mismos. Para estos casos, JPA ofrece la anotación `GeneratedValue`.
- Existen distintas estrategias de generación:
  - AUTO: el proveedor de persistencia elige la estrategia. En *MySQL* es `TABLE`.
  - TABLE: utiliza una tabla para almacenar el último id que se generó.
  - SEQUENCE: en bases de datos que soportan la generación de secuencias, JPA puede usar una para la generacíon y asignación de identificadores. En MySQL genera una columna autonumerada. El tipo del atributo anotado debe ser numérico.
  - IDENTITY: utiliza una columna autonumerada si la base de datos lo permite. El tipo del atributo anotado debe ser numérico.

Método de Clave Primaria

```
@Entity
@Table(name="editorial")
public class Editorial implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.TABLE)
    private String id;

    //...
}
```

Asociaciones

- La mayoría de las entidades se asocian con otras entidades dentro del modelo del dominio.
- Cada entidad juega un rol dentro de una asociación, teniendo distintos roles dependiendo de la asociación.
- Las asociaciones pueden ser unidireccionales o bidireccionales.
- Cada entidad tiene una participación y una cardinalidad en la asociación.
- Debemos ser capaces de mapear estas asociaciones.

Asociaciones Uno a Uno

- Si uno de los empleados de una editorial tiene el rol de director de la misma, nos encontramos con una asociación **uno a uno** entre las entidades `Editorial` y `Empleado`. Lo normal es acceder al director a través de la editorial, por lo que se define como asociación unidireccional con la anotación `@OneToOne`.

```
@Entity
@Table(name="editorial")
public class Editorial implements Serializable {
    @Id
    private Integer id;
    private String nombre;

    @OneToOne
    private Empleado director;
    //...
}
```

Asociaciones Uno a Uno

- Una asociación Uno a Uno puede ser bidireccional si ambas entidades tienen un atributo que apunta a la otra. Por ejemplo, si un empleado tiene una plaza de parking:

```
@Entity
public class PlazaParking implements Serializable{
  @Id
  private Long id;
  private Integer numero;
  private String ubicacion;
  @OneToOne(mappedBy="plazaParking")
  private Empleado empleado;
}
@Entity
public class Empleado {
  //...
  @OneToOne
  private PlazaParking plazaParking;
```

Asociaciones Uno a Uno

- En los ejempos anteriors `Editorial` es la dueña de la **asociación con Director** de manera implícita, pues es unidireccional, y por tanto, la tabla correspondiente tendrá una columna con una clave ajena.
- Dicha columna se puede personalizar:

```
@OneToOne
@JoinColumn(name="director_fk")
private Empleado director;
```

- En el caso de la relación bidireccional, cualquiera de las entidades puede ser propietaria de la asociación. Debemos indicar `@mappedBy` en uno de los atributos para indicar que ambos atributos pertencen a la misma asociación. La entidad que **no tiene `@mappedBy` es la propietaria de la relación** y puede definir `@JoinColumn`.

Asociaciones Muchos a Uno

- Una editorial tiene empleados en plantilla y cada empleado trabaja en una única editorial. Esta asociación, vista desde el punto de vista de la entidad `Empleado` es Muchos a Uno, pues muchos empleados trabajarán en la misma editorial. Si solo definimos el atributo en el lado de `Empleado`, la asociación es unidireccional y se usa la anotación `@ManyToOne`:

```
@Entity
public class Empleado {
    //...
    @ManyToOne
    @JoinColumn(name="editorial_fk")
    private Editorial editorial;
```

- El mapeo `@ManyToOne` siempre es el lado propietario de la asociación.

Asociaciones Muchos a Uno y Uno a Muchos bidireccionales

- Si hacemos bidireccional la asociación anterior, hacemos uso de la anotación `@OneToMany` en la otra entidad e indicamos `mappedBy` para establecer que ambos atributos pertencen a la misma asociación.

```
@Entity
public class Editorial {
    //...
    @OneToMany(mappedBy = "editorial")
    private List<Empleado> empleados;
```

- Esta versión del mapeo bidireccional, al igual que la versión anterior creará una columna en `Empleado` que será una clave ajena a la tabla `Editorial`.

Asociaciones Uno a Muchos unidireccionales

- Si definimos una asociación uno a mucho como unidireccional, nos estamos centrando en el lado de la asociación que tiene una colección de entidades. Al dejar el mapeo como unidireccional, por defecto JPA creará una tabla intermedia en la que se almacenará una clave ajena por cada entidad implicada. Con la anotación `@JoinTable` se puede personalizar esta tabla unión:

```
@Entity
public class Editorial implements Serializable{
    //...
    @OneToMany
    @JoinTable(name="editorial_empleado",
        joinColumns=@JoinColumn(name="editorial_fk"),
        inverseJoinColumns=@JoinColumn(name="empleado_fk"))
        private List<Empleado> empleados;
```

- **Es menos frecuente**.

Asociaciones Uno a Muchos unidireccionales

- Si queremos forzar que no haya tabla intermedia en este caso y JPA solamente cree una columna con clave ajena en la tabla que representa las entidades de la colección, debemos indicar nombre de la columna que hará de clave ajena con `@JoinColumn` :

```
@Entity
public class Editorial implements Serializable{
    //...
    @OneToMany
    @JoinColumn(name = "editorial_fk")
    private List<Empleado> empleados;
```

Algunos proveedores de JPA pueden no tolearlo bien, en esos casos puede ser mejor opción hacer la relación bidireccional o quedarnos con la `JoinTable` . Por ejemplo, puede afecta al funcionamiento de las consultas JPQL.

Asociaciones Muchos a Muchos

- Un libro puede tener varios autores y a su vez un autor puede haber escrito varios libros, por lo tanto estamos ante una asociación muchos a muchos.

```
public class Autor implements Serializable {
    //...
    @ManyToMany
    @JoinTable(name = "autor_libro", joinColumns = {
            @JoinColumn(name = "autor_fk") },
            inverseJoinColumns = { @JoinColumn(name = "libro_fk") })
    private List<Libro> libros;
}
public class Libro implements Serializable {
    //...
    @ManyToMany(mappedBy="libros")
    private List<Autor> autores;
}
```

Lectura perezosa

- A nivel de asociaciones la lectura perezosa mejora la eficiencia.
  - Reduce la cantidad de SQL que se ejecuta.
  - Acelera las consultas y la carga de objetos.
- En asociaciones `@OneToOne` y `@ManyToOne` la lectura es inmediata por defecto.
- En asociaciones con colecciones, `@OneToMany` y `@ManyToMany` la lectura es perezosa por defecto.
- Esto se puede modificar con el elemento `fetch` en cada una de las anotaciones para asociaciones.

Objetos embebidos

- Un objeto embebible puede ser aquel que no tiene identidad por sí mismo, sino que depende de otra entidad para su identidad. Aparece como embebido en la entidad de la que depende.
- En Java tendremos dos clases separadas, en la base de datos el objeto embebido aparecerá como columnas en la tabla de la entidad de la que depende.

```
@Embeddable
public class Direccion implements Serializable{
    private String calle;
    private Integer numero;
    private String ciudad;
    //...
@Entity
public class Editorial implements Serializable{
    //...
    @Embedded
    private Direccion direccion;
}
```

Colecciones

- Una entidad puede tener propiedades de tipo `java.util.Collection` y/o `java.util.Map` que contengan tipos básicos u objetos embebibles.
- Los elementos de estas colecciones serán almacenados en una tabla diferente a la de la entidad que tiene la colección.

```
@Entity
@Table(name="editorial")
public class Editorial implements Serializable {
    //...
    @ElementCollection
    @CollectionTable ( name="genero")
    private List<String> generos;
    //...
}
```

Entity Manager

- Las entidades no se persistence al crearlas ni se borran cuando el recolector de basura libera su memoria.
- JPA proporciona la interfaz `EntityManager` para gestionar y buscar entidades en una base de datos relacional.
- En JPA, el contexto de persistencia es un conjunto de instancias de entidades gestionadas por JPA y que se configura a través de una unidad de persistencia.
- El objeto `EntityManager` gestiona el contexto de persistencia.
- Todos los objetos `EntityManager` se obtienen a través de la factoria `EntityManagerFactory`.

```
EntityManagerFactory emf =
Persistence.createEntityManagerFactory("nombreUnidadPersistencia");
EntityManager em = emf.createEntityManager();
```

Operaciones con `EntityManager`

Persistir una entidad

- Guardar en la base de datos una instancia que solo tiene representación en memoria.

```
Editorial e = new Editorial(1,'alba editorial', '1234566', ...);
em.persist(e);
```

Buscar una entidad

- Recuperar una instancia de una entidad de la base de datos

```
Editorial e = em.find(Editorial.class, 1);
```

- Recibe la clase de la entidad que se busca y la clave primaria de la instancia.

Operaciones con `EntityManager`

Eliminar una entidad

- Para borrarla debe estar en el contexto de persistencia.

```
Editorial e = em.find(Editorial.class, 1);
em.remove(e);
```

Actualizar una entidad

- Si la entidad está en el contexto de persistencia, no es necesario utilizar `EntityManager`.

```
Editorial e = em.find(Editorial.class, 1);
e.setNombre('alba ediciones');
```

Transacciones

- Excepto el método `find` que no modifica la base de datos, el resto deben ejecutarse dentro de una transacción.
- Contexto de ejecución dentro del cual podemos tener varias operaciones como si fueran una sola.

```
em.getTransaction().begin();
Editorial e = new Editorial(1,'alba editorial', '1234566', ...);
em.persist(editorial);
em.getTransaction().commit();
```

También `em.getTransaction().rollback()` para deshacer una transacción que no ha finalizado correctamente con el `commit`.

Gestión por contenedor

- En los ejemplos vistos, el objeto `EntityManager` está siendo gestionado por la aplicación `Application-Managed`.
- En aplicaciones JEE, el `EntityManager` puede ser gestionado por el contenedor `Container-Managed`.
- Obtenemos el objeto `EntityManager` por inyección de dependencia.

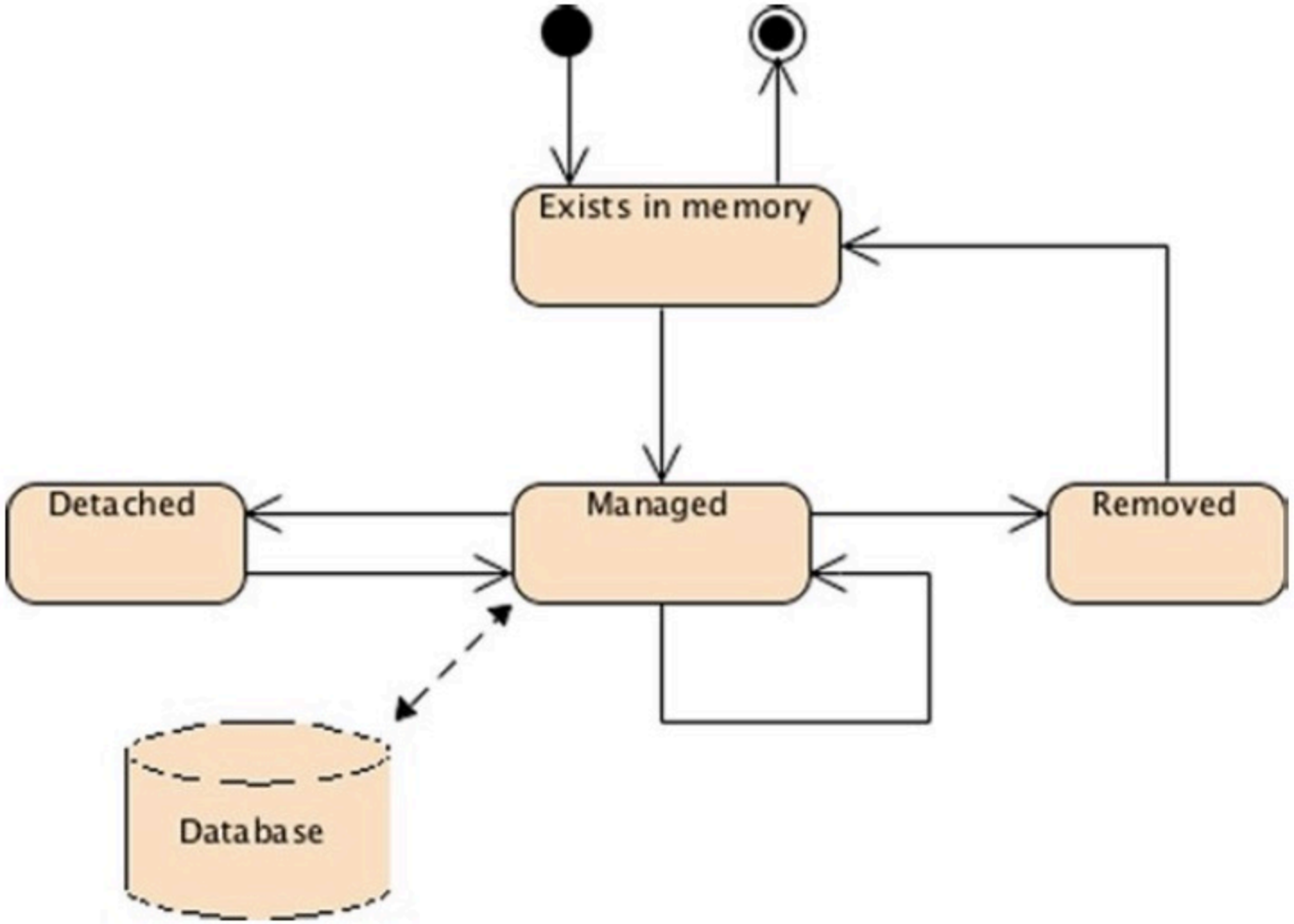
```
@Stateless
public class Servicio {
    @PersistenceContext(unitName="nombreUnidadPersistencia")
    EntityManager em;
```

- En aplicaciones JEE también podemos tener transaccionalidad JTA, donde las transacciones las gestiona el contenedor.
- Necesitamos utilizar un contenedor JEE completo (Ej. *payara*, *wildfly*).

Estados de una entidad JPA

- Una entidad JPA 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 o no gestionada): Los cambios realizados en la entidad no están sincronizados con la base de datos.

Ciclo de vida de una entidad



Otras operaciones con `EntityManager`

Obtener una referencia a una entidad

```
Editorial e = em.getReference(Editorial.class, 1);
```

Forzar los cambios en la base de datos

```
em.flush();
```

Actualizar una entidad con los datos que hay en la base de datos

```
em.refresh(editorial);
```

Pasar una entidad de estado *Detached* a estado *Managed*

```
Editorial editorial = em.merge(editorial);
```

Operaciones en cascada

- Por defecto, cualquier operación del objeto `EntityManager` se aplica solamente a la entidad que se pasa por parámetro.
- Podemos configurar que las operaciones se propaguen en cascada a otras entidades que tienen asociaciones con la entidad que se pasa por parámetro. Atributo `cascade` de todas las anotaciones de asociación.

```
@Entity
@Table(name="editorial")
public class Editorial implements Serializable {
    @OneToMany(mappedBy = "editorial",
    fetch = FetchType.EAGER, cascade = CascadeType.ALL)
    private List<Empleado> empleados;
    //...
```

- Se puede definir para las operaciones `PERSIST`, `REFRESH`, `REMOVE`, `MERGE` y `DETACH` o usar `ALL` para indicarlas todas.