Acceso a datos

ORM – Mapeo objeto-relacional

JPA – Anotaciones JPA para mapeo de relaciones

IES Clara del Rey – Madrid

Relaciones en BD – Uno a uno – 1:1

Cada fila de una tabla se relaciona exactamente con una fila de otra tabla, y viceversa.

Uso típico: Dividir información que podría estar en una sola tabla, por razones de modularidad o privacidad.

Ejemplo:

- Una tabla Person con columnas generales y otra tabla Passport con detalles específicos del pasaporte de la persona.
- Cada persona tiene exactamente un pasaporte y cada pasaporte pertenece a una persona.

Implementación: Se utiliza una clave primaria (que suele coincidir) en ambas tablas y una clave ajena en la tabla "principal" para vincularlas.

Relaciones en BD – Uno a N – 1:N

Una fila de una tabla está relacionada una o más filas de otra tabla, pero cada fila en la segunda tabla está relacionada solo con una en la primera.

Uso típico: modelar jerarquías o dependencias entre elementos.

Ejemplo:

- Una tabla Author y una tabla Book.
- Un autor escribe muchos libros, pero un libro tiene un solo autor.

Implementación: La tabla "muchos" incluye una clave ajena que referencia la clave primaria de la tabla "uno".

Relaciones en BD – N a M – N:M

Muchas filas en una tabla pueden estar relacionadas con muchas filas en otra tabla, y viceversa.

Uso típico: Modelar asociaciones complejas.

Ejemplo:

- Una tabla Student y una tabla Course.
- Un estudiante puede inscribirse en varios cursos, y en un curso puede haber varios estudiantes inscritos.

Implementación: Se crea una tabla intermedia (o de unión) que contiene claves ajenas de ambas tablas principales. Esta tabla forma un "puente" entre las dos tablas, y puede tener atributos (columnas) adicionales.

Relaciones en BD – 0:1 y 0:N

Son casos especiales de las relaciones 1:1 y 1:N en los que la relación es opcional, no es obligatoria. La columna que es la clave ajena puede dejarse con valor "null".

Ejemplos:

- 0:1 – Empleados y coches. No todos los empleados tienen coche asignado (la relación es opcional). En el caso de que lo tengan, sólo tienen asignado 1.
- 0:N – Cliente y pedidos. Un nuevo cliente no tendrá pedidos, pero a medida que pase el tiempo tendrá muchos pedidos.

La diferencia entre una relación 1: y 0: es que la columna que es clave ajena admite valores nulos.

JPA – Anotaciones para relaciones

JPA tiene anotaciones para todas las relaciones expuestas. Establecen relaciones entre dos clases, la relación "directa" y la "inversa", que permiten "navegar" entre objetos relacionados en ambos sentidos.

Relación en BD	Anotación JPA directa	Anotación JPA inversa
0:1 / 1:1	@OneToOne	@OneToOne
0:N / 1:N	@ManyToOne	@OneToMany
N:M	@ManyToMany	@ManyToMany

El lado directo (propietario) de la relación es el que contiene la clave ajena. El lado inverso no contiene la clave ajena, es la tabla "referenciada".

En las relaciones N:M el desarrollador decide el lado directo y el inverso.

JPA – @ManyToOne y @JoinColumn

@ManyToOne se utiliza para establecer una relación de muchos a uno entre dos entidades.

Esta anotación se coloca en el lado propietario de la relación:

- El que contiene la clave ajena (foreign key)
- El que contiene la propiedad de navegación (no colección)

Se usa para mapear esta clave ajena que referencia a otra entidad.

Se puede usar junto a @JoinColumn para personalizar el nombre de la columna de la BD que contiene la clave ajena.

Se puede usar @OneToMany en el lado inverso (la otra entidad) como relación opuesta, de uno a muchos, con una colección.

JPA – @ManyToOne y @JoinColumn

```java
@Entity
public class Empleado {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String nombre;

    @ManyToOne
    // Personaliza el nombre de la columna de la clave externa
    @JoinColumn(name = "departamento_id")
    private Departamento departamento;

    // Otros campos, constructores, getters, setters, etc.
}
```

JPA – @OneToMany

Se utiliza para establecer relación de uno a muchos entre entidades.

Puede ser la inversa de un @ManyToOne, y entonces se coloca en el lado "no propietario" de la relación, que:

- No contiene clave ajena, es la tabla referenciada por @ManyToOne
- Contiene una colección de elementos relacionados, y se utiliza para mapear esta colección o lista de entidades.

En este caso se utiliza el atributo / parámetro "mappedBy" para indicar el lado propietario de la relación.

Se puede usar sin @ManyToOne en el lado inverso.

JPA – @OneToMany

```java
@Entity
public class Departamento {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String nombre;
    // "departamento" es el nombre del atributo en la clase Empleado
    @OneToMany(mappedBy = "departamento")
    private List<Empleado> empleados;

    // Otros campos, constructores, getters, setters, etc.
}
```

JPA – @ManyToMany y @JoinTable

@ManyToMany se utiliza para establecer una relación de muchos a muchos entre dos entidades. En la base de datos habrá una tabla de unión, relacionando las filas de las dos tablas con relación N:M.

Esta anotación se coloca en ambas entidades que participan en la relación y se utiliza para mapear las asociaciones entre ellas.

Es muy habitual usarla junto a la anotación @JoinTable para especificar la tabla de unión. Esto:

- En un enfoque code-first aporta control más preciso sobre el esquema de la base de datos generado por JPA para la relación.
- En un enfoque database-first permite establecer la relación, aunque las columnas y tablas no sigan ciertas convenciones por defecto.

JPA – @ManyToMany y @JoinTable

```java
@Entity
public class Estudiante {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String nombre;

    @ManyToMany
    @JoinTable(
      name = "estudiante_curso",
      joinColumns = @JoinColumn(name = "estudiante_id"),
      inverseJoinColumns = @JoinColumn(name = "curso_id")
    )
    private Set<Curso> cursos;

    // Otros campos, constructores, getters, setters, etc.
}
```

JPA – @ManyToMany y @JoinTable

```java
@Entity
public class Curso {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String nombre;

    @ManyToMany(mappedBy = "cursos")
    private Set<Estudiante> estudiantes;

    // Otros campos, constructores, getters, setters, etc.
}
```

JPA – @OneToOne

Se utiliza para establecer una relación de uno a uno entre dos entidades.

Esta relación significa que cada entidad en un lado de la relación está relacionada con exactamente una entidad en el otro lado.

En estas relaciones, una de las entidades es la propietaria de la relación, la que tiene la clave ajena en la base de datos.

La entidad "no propietaria" es la que tiene el atributo "mappedBy".

Como en otras relaciones, se puede usar @JoinColumn para especificar el nombre de la columna que contiene las claves ajenas.

Hay otras anotaciones como @MapsId o @PrimaryKeyJoinColumn que pueden necesitarse, pero son uso más avanzado y esporádico.

JPA – @OneToOne

```java
@Entity
public class Persona {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String nombre;

    @OneToOne
    @JoinColumn(name = "pasaporte_id")
    // Clave ajena en la tabla Persona
    private Pasaporte pasaporte;

    // Getters y setters...
}
```

JPA – @OneToOne

```java
@Entity
public class Pasaporte {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String numero;

    @OneToOne(mappedBy = "pasaporte")
    // Mapeo inverso hacia Persona
    private Persona persona;

    // Getters y setters...
}
```

JPA – Resumen de anotaciones

En relaciones @OneToOne:

	Propietario (directo)	Secundario (inverso)
Anotación	@OneToOne + @JoinColumn	@OneToOne(mappedBy="atributo")
Clave ajena	Sí, clave ajena en la tabla	No, sólo refleja la relación

En relaciones @ManyToOne / @OneToMany:

	Propietario (directo)	Secundario (inverso)
Anotación	@ManyToOne + @JoinColumn	@OneToMany(mappedBy="atributo")
Clave ajena	Sí, clave ajena en la tabla	No, sólo refleja la relación

En relaciones @ManyToMany / @ManyToMany no se puede establecer un lado "directo" e "inverso", las relaciones son "igualitarias", pero sí un propietario. El propietario será el que no tenga "mappedBy".

JPA – Importancia del propietario

En JPA es importante saber quién es el propietario de una relación.

El propietario de la relación es el que la controla, y eso implica que sólo el lado propietario puede hacer cambios en la relación. Esto es:

- Para eliminar la relación entre dos objetos, se hace en el propietario. Si se elimina en el "secundario", no se reflejarán los cambios.
- Del mismo modo, para crear una relación entre objetos, también se hace en el propietario. Si no, no se crea la relación.
- JPA, cuando genera SQL para consultas, utiliza el lado propietario para generar las consultas insertar, actualizar y eliminar relaciones.

JPA – Propietario – Ejemplo

Clase "Author", relación con "Book". Un libro es de un solo autor. Un autor tiene muchos libros publicados. "Book" es el lado propietario.

```java
@Entity
public class Book {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String title;

    @ManyToOne
    @JoinColumn(name = "author_id")
    private Author author;

    // Getters y setters...
}
```

```java
@Entity
public class Author {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    @OneToMany(mappedBy = "author")
    private List<Book> books = new ArrayList<>();

    // Getters y setters...
}
```

JPA – Propietario – Ejemplo

Para añadir un libro a un autor, esto no funcionaría:

```
Author author = buscarAutor( ... );
Book newBook = crearBook( ... );
author.books.add(newBook);
```

Sin embargo, esto sí funcionaría:

```
Author author = buscarAutor( ... );
Book newBook = crearBook( ... );
book.setAuthor(author);
```

El "extemo" modificado tiene que ser el propietario de la relación.

JPA – Anotación @Enumerated

Se utiliza para almacenar valores de enumeración (tipo enum) en una columna de base de datos.

Puede tomar dos valores:

- EnumType.STRING: Guarda en la base de datos el nombre de la constante de enumeración.
- EnumType.ORDINAL: Guarda en la base de datos el ordinal de la constante de enumeración. Es la opción por defecto.

STRING suele ser más legible en la BD, mientras que ORDINAL es más eficiente en términos de espacio si la cantidad de columnas y registros con elementos enum es grande.

JPA – Anotación @Enumerated

```java
public enum TipoCombustible {
    GASOLINA, DIESEL, ELECTRICO
}

@Entity
public class Coche {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String modelo;

    @Enumerated
    private TipoCombustible tipoCombustible;

    // Otros campos, constructores, getters, setters, etc.
}
```

```java
@Enumerated(EnumType.STRING)
private TipoCombustible tipoCombustible;
```

```java
@Enumerated(EnumType.ORDINAL)
private TipoCombustible tipoCombustible;
```

JPA – Claves compuestas con @Embeddable y @EmbeddedId

Para definir claves primarias compuestas, también conocidas como claves primarias con multicolumna, usando @Embeddable y @EmbeddedId:

- Se crea una clase que representa la clave primaria, con los atributos necesarios, y se anota con @Embeddable. Esta clase tiene que ser pública, ser serializable, tener constructor por defecto (sin parámetros) y definir equals y hashCode, para poder comparar claves.
- Se usa un objeto de esta clase como clave primaria de la entidad, de forma que la clave primaria ya no es un tipo primitivo, sino un objeto con varios atributos.

Ejemplo: clase "LineaPedido" con una clave compuesta por el id del pedido y el id del producto. En la tabla de líneas de pedido cada registro se identifica por los dos campos (id del pedido e id del producto).

JPA – Claves compuestas con @Embeddable y @EmbeddedId

Clase para definir la clave primaria:

```java
@Embeddable
public class LineaPedidoId implements Serializable {
    private Long idProducto;
    private Long idPedido;

    // Constructores, equals, hashCode, etc.
}
```

Clase LineaPedido, que usa la clase anterior como clave primaria:

```java
@Entity
public class LineaPedido {
    @EmbeddedId
    private LineaPedidoId id;

    private int cantidad;

    // Otros campos, constructores, getters, setters, etc.
}
```

JPA – Claves compuestas con @Embeddable y @EmbeddedId

Creación de una línea de pedido con este mecanismo:

```
// Creación de un objeto LineaPedido con la versión @Embeddable
// y @EmbeddedId
LineaPedidoId id = new LineaPedidoId();
id.setIdProducto(1L);
id.setIdPedido(101L);

LineaPedido lineaPedido = new LineaPedido();
lineaPedido.setId(id);
lineaPedido.setCantidad(5);
```

JPA – Claves compuestas con @Id e @IdClass

Alternativa a @Embeddable y @EmbeddedId. Permite "desestructurar" el objeto para la clave primaria, y no tener un objeto como clave, sino los atributos por separado.

En este caso:

- También se crea una clase que representa la clave primaria, con los atributos necesarios, pero no es necesario anotarla con @Embeddable. El resto de los requisitos se mantienen.
- Se usa el atributo @IdClass para indicar la clase que representa la clave primaria en el objeto, pero no hay que usar el objeto como clave primaria. Se usan atributos independientes anotados con @Id.

JPA – Claves compuestas con @Id e @IdClass

Clase para definir la clave primaria:

```java
public class LineaPedidoId implements Serializable {
    private Long idProducto;
    private Long idPedido;
    // Constructores, equals, hashCode, etc.
}
```

Clase LineaPedido, que identifica la clase anterior como clave primaria:

```java
@Entity
@IdClass(LineaPedidoId.class)
public class LineaPedido {
    @Id
    private Long idProducto;
    @Id
    private Long idPedido;
    private int cantidad;
    // Otros campos, constructores, getters, setters, etc.
}
```

JPA – Claves compuestas con @Id e @IdClass

Creación de una línea de pedido con este mecanismo:

```
// Creación de un objeto LineaPedido con la versión @IdClass
LineaPedido lineaPedido = new LineaPedido();
lineaPedido.setIdProducto(1L);
lineaPedido.setIdPedido(101L);
lineaPedido.setCantidad(5);
```

JPA + Hibernate + MySQL – Dependencias

Vamos a acceder a MySQL utilizando Hibernate como proveedor JPA, pero sin utilizar Spring o Spring data. En este caso necesitaremos una serie de dependencias y ficheros de configuración.

Para utilizar Hibernate necesitamos la siguiente dependencia:

- Group ID: org.hibernate
- Artifact ID: hibernate-core

Para acceder a la base de datos necesitaremos el driver JDBC adecuado. En el caso de MySQL podría ser:

- Group ID: mysql
- Artifact ID: mysql-connector-java

JPA + Hibernate – Fichero persistence.xml

El fichero persistence.xml es un fichero de configuración de JPA que define configuraciones específicas de persistencia.

Normalmente se ubica en el directorio META-INF dentro del classpath de la aplicación.

En Maven se ubica en la carpeta src/main/resources/META-INF.

Contiene información sobre la unidad de persistencia, proveedores de persistencia, clases de entidad, o configuraciones específicas del proveedor de persistencia.

JPA + Hibernate – Fichero persistence.xml

El fichero contiene la definición de la unidad de persistencia.

- Se define con el elemento <persistence-unit>. El atributo "name" asigna un nombre para poder referenciarla desde el código.
- La unidad de persistencia incluye y gestiona las entidades definidas con @Entity, que representan los objetos de la base de datos.

La definición de la unidad de persistencia incluye propiedades de persistencia, como, por ejemplo:

- Driver que se usará para conectar a la BD
- Cadena de conexión (normalmente JDBC) para conectar a la BD
- Usuario y contraseña para conectar a la BD.