UT5 - Conexión a base de datos objeto-relacionales usando Spring Data JPA

Profesor: Luis Javier López López

Índice

UT5 - Conexión a base de datos objeto-relacionales usando Spring Data JPA

Profesor: Luis Javier López López

Introducción general

En las aplicaciones Java modernas que usan **JPA (Java Persistence API)** para el acceso a bases de datos relacionales, una parte fundamental del trabajo consiste en **realizar consultas** sobre las entidades del modelo.
Estas consultas permiten recuperar, filtrar, actualizar y eliminar información de forma flexible, manteniendo la abstracción de la capa de persistencia.

JPA proporciona **tres mecanismos principales** para definir y ejecutar consultas, cada uno con sus propias características, ventajas y usos recomendados:

Tipo de Consulta	Sintaxis / Cómo se define	Ventajas	Desventajas	Uso recomendado
Nativa (SQL puro)	`@Query(value = "SELECT * FROM alumnos WHERE edad > :edad", nativeQuery = true)`	Control total sobre el SQL; permite funciones específicas del motor; compatible con consultas preexistentes	Dependiente del motor de base de datos; menos portable	Consultas complejas, optimizadas o específicas del motor
JPQL (Java Persistence Query Language)	`@Query("SELECT a FROM Alumno a WHERE a.edad > :edad")`	Portabilidad entre motores; trabaja con entidades y atributos; integración total con Hibernate	No todas las funciones SQL están disponibles; menos control sobre optimizaciones	Consultas normales sobre entidades y relaciones
Consultas derivadas de métodos (Spring Data JPA)	`List<Alumno> findAllByApellidoEqualsAndNombreLike(String apellido, String nombre);`	No requiere escribir SQL o JPQL; muy rápido de implementar; integrado con repositorios	Limitado a lo que permite la convención de nombres; consultas complejas pueden ser ilegibles	Consultas simples basadas en atributos de la entidad, filtros combinados y operaciones comunes

1. Nativa (SQL puro)

1.1 Contexto general

En un proyecto que usa **Spring Data JPA**, cada entidad suele tener un repositorio asociado, por ejemplo:

```
public interface AlumnoRepository extends JpaRepository<Alumno, Long> {
}
```

Spring Data genera automáticamente los métodos básicos (***findAll, save, delete***, etc.), pero cuando necesitas consultas personalizadas, puedes usar:

```
@Query("SELECT a FROM Alumno a WHERE a.edad > 18")
```

Eso es **JPQL** (no SQL real).

Pero si necesitas ejecutar **SQL nativo directamente sobre la base de datos**, debes indicar a Spring Data JPA que se trata de una consulta nativa con:

```
@Query(value = "SELECT * FROM alumnos WHERE edad > 18", nativeQuery = true)
```

En ese caso, Hibernate (el proveedor JPA) **ejecutará ese SQL directamente** sobre la conexión actual y mapeará el resultado a la entidad correspondiente o al tipo de retorno especificado.

1.2 Estructura básica de una consulta nativa

Ejemplo simple:

```
public interface AlumnoRepository extends JpaRepository<Alumno, Long> {

@Query(value = "SELECT * FROM alumnos WHERE edad > :edad", nativeQuery = true)
    List<Alumno> findAlumnosMayores(@Param("edad") int edad);
}
```

Explicación:

- **value**: contiene la consulta SQL real (no JPQL).
- **nativeQuery = true:** le indica a Spring Data que la consulta es nativa.
- **@Param("edad"):** vincula el parámetro del método con el del SQL.
- El tipo de retorno (***List<Alumno>***) debe corresponder con la entidad o un tipo compatible.

Hibernate internamente:

- Prepara y ejecuta el SQL directamente sobre la BD.
- Usa su **ResultSet** interno para mapear el resultado a instancias de Alumno.
- Respeta la transaccionalidad JPA de Spring.

1.3 Variantes de consultas nativas

a) Parámetros nombrados

```
@Query(value = "SELECT * FROM alumnos WHERE apellido = :apellido",
nativeQuery = true)
List<Alumno> findByApellido(@Param("apellido") String apellido);
```

b) Parámetros posicionales

```
@Query(value = "SELECT * FROM alumnos WHERE edad > ?1", nativeQuery =
true)
List<Alumno> findMayoresQue(int edad);
```

Hibernate los reemplaza de forma segura (sin riesgo de inyección SQL).

1.4 Consultas nativas que devuelven entidades

Cuando el **SELECT** devuelve todas las columnas que forman la entidad, Spring Data JPA puede mapear automáticamente el resultado.

```
@Query(value = "SELECT * FROM alumnos WHERE edad >= :edad", nativeQuery
= true)
List<Alumno> buscarMayoresDe(@Param("edad") int edad);
```

Importante:
*El nombre de las columnas devueltas por el SQL debe coincidir con los nombres de columna mapeados en la entidad (o con los **@Column(name = "columna")**).*

1.5 Consultas nativas que devuelven campos específicos o DTOs

A veces no quieres devolver la entidad completa, sino solo ciertos campos.
Para eso tienes **tres opciones**.

Opción 1 — Devolver *List<Object[]>*

```
@Query(value = "SELECT nombre, edad FROM alumnos WHERE edad > :edad",
nativeQuery = true)
List<Object[]> findNombresYEdades(@Param("edad") int edad);
```

Uso:

```
for (Object[] fila : repo.findNombresYEdades(18)) {
    String nombre = (String) fila[0];
    int edad = ((Number) fila[1]).intValue();
}
```

Opción 2 — Devolver una proyección basada en interfaz

```
public interface AlumnoResumen {
    String getNombre();
    int getEdad();
}
```

Y el repositorio:

```
@Query(value = "SELECT nombre, edad FROM alumnos WHERE edad > :edad",
nativeQuery = true)
List<AlumnoResumen> findResumen(@Param("edad") int edad);
```

Hibernate mapeará automáticamente cada columna al método con el mismo nombre (por convención). Muy útil para reportes o consultas ligeras.

Opción 3 — Devolver un DTO explícito (Java record o clase)
Spring Data JPA no construye DTOs automáticamente en consultas nativas,
pero puedes usar ***@SqlResultSetMapping*** si necesitas hacerlo.

Por ejemplo:

```
@Entity
@SqlResultSetMapping(
    name = "AlumnoDTOMap",
    classes = @ConstructorResult(
        targetClass = AlumnoDTO.class,
        columns = {
            @ColumnResult(name = "nombre", type = String.class),
            @ColumnResult(name = "edad", type = Integer.class)
        }
    )
)
@NamedNativeQuery(
    name = "Alumno.findResumen",
    query = "SELECT nombre, edad FROM alumnos WHERE edad > :edad",
    resultSetMapping = "AlumnoDTOMap"
)
public class Alumno {
    @Id
    private Long id;
}
```

Y en el repositorio:

```
@Query(name = "Alumno.findResumen", nativeQuery = true)
List<AlumnoDTO> findResumen(@Param("edad") int edad);
```

1.6 Consultas de modificación nativas (UPDATE, DELETE, INSERT)

Spring Data JPA también permite ejecutar sentencias nativas que modifican datos, pero debes marcar el método con @Modifying y, si es necesario, **@Transactional**.

```
@Transactional
@Modifying
@Query(value = "UPDATE alumnos SET edad = edad + 1 WHERE edad <
:limite", nativeQuery = true)
int incrementarEdad(@Param("limite") int limite);
```

- **@Transactional**: asegura que la operación esté dentro de una transacción.
- **@Modifying**: le indica a Spring que no es un SELECT, sino una operación de escritura.
- El método devuelve el número de filas afectadas.

1.7 Paginación y ordenación con consultas nativas

Spring Data permite combinar consultas nativas con **Pageable** y **Sort**, siempre que se proporcione un countQuery para calcular el total.

```
@Query(
  value = "SELECT * FROM alumnos WHERE edad >= :edad",
  countQuery = "SELECT count(*) FROM alumnos WHERE edad >= :edad",
  nativeQuery = true
)
Page<Alumno> findMayoresDe(@Param("edad") int edad, Pageable pageable);
```

Uso:

```
Pageable pageable = PageRequest.of(0, 10, Sort.by("edad").descending());
Page<Alumno> pagina = repo.findMayoresDe(18, pageable);
```

Hibernate se encarga de ejecutar ambas consultas (**SELECT** y **COUNT)** y mapear los resultados.

1.8 Named Native Queries en combinación con Repositorios

Si defines consultas nativas con **@NamedNativeQuery** dentro de la entidad, puedes invocarlas desde el repositorio sin escribir la consulta en la interfaz.

```java
@NamedNativeQuery(
    name = "Alumno.buscarPorApellido",
    query = "SELECT * FROM alumnos WHERE apellido = :apellido",
    resultClass = Alumno.class
)
@Entity
public class Alumno {
    @Id
    private Long id;
}
```

Repositorio:

```java
@Query(name = "Alumno.buscarPorApellido", nativeQuery = true)
List<Alumno> findByApellido(@Param("apellido") String apellido);
```

1.8 Resumiendo

Concepto	Descripción	Ejemplo
nativeQuery = true	Indica que la consulta usa SQL puro.	@Query(value = "SELECT * FROM alumnos", nativeQuery = true)
Parámetros	Se vinculan con ?1, ?2 o :nombre.	WHERE edad > :edad
Mapeo a entidad	Si el SELECT devuelve todas las columnas, se puede mapear directamente.	List<Alumno>
Proyección	Puedes devolver Object[] o una interfaz de proyección.	List<AlumnoResumen>
Actualización	Usar @Modifying y @Transactional.	UPDATE alumnos SET ...
Paginación	Combinar con Pageable y countQuery.	Page<Alumno>

2. Consultas JPQL en Spring Data JPA

2.1 Introducción a JPQL

JPQL (Java Persistence Query Language) es un lenguaje de consultas **orientado a objetos**, que funciona sobre **entidades y atributos de Java**, en lugar de sobre tablas y columnas de la base de datos.
Su sintaxis es similar a SQL, pero con diferencias importantes:

- Se usan **nombres de clases y atributos**, no nombres de tablas o columnas.
- Devuelve **objetos gestionados por JPA**, no registros crudos.
- Hibernate traduce JPQL a SQL nativo según el motor de base de datos usado.

Ejemplo básico:

```
@Query("SELECT a FROM Alumno a WHERE a.edad > :edad")
List<Alumno> findAlumnosMayores(@Param("edad") int edad);
```

2.2. Sintaxis básica de JPQL

- **Selección de entidades completas**

```
SELECT a FROM Alumno a
```

- **Filtros con `WHERE`**

```
SELECT a FROM Alumno a
```

- **Ordenación**

```
SELECT a FROM Alumno a ORDER BY a.nombre ASC
```

- **Funciones agregadas**

```
SELECT COUNT(a) FROM Alumno a WHERE a.edad > :edad
```

2.3 Declaración en repositorios Spring Data JPA

En Spring Data JPA, las consultas JPQL se definen con **@Query,** sin nativeQuery = true:

```
public interface AlumnoRepository extends JpaRepository<Alumno, Long> {

    @Query("SELECT a FROM Alumno a WHERE a.apellido = :apellido")
    List<Alumno> findByApellido(@Param("apellido") String apellido);
}
```

- JPQL trabaja con **entidades y sus atributos**, no columnas.
- Hibernate genera automáticamente el SQL equivalente al motor de base de datos configurado.

2.4 Uso de parámetros

JPQL permite **parámetros nombrados** y **posicionales**:

a) Parámetros nombrados

```
@Query("SELECT a FROM Alumno a WHERE a.nombre LIKE :nombre")
List<Alumno> findByNombreLike(@Param("nombre") String nombre);
```

- Más legible y flexible que los posicionales.

b) Parámetros posicionales

```
@Query("SELECT a FROM Alumno a WHERE a.edad > ?1")
List<Alumno> findMayoresQue(int edad);
```

- Menos usado que los parámetros nombrados en Spring Data JPA.

2.5 Proyecciones y selección parcial

Si solo necesitamos ciertos atributos, JPQL permite crear **proyecciones**:

```
@Query("SELECT a.nombre, a.edad FROM Alumno a WHERE a.edad > :edad")
List<Object[]> findNombresYEdades(@Param("edad") int edad);
```

- Cada elemento de la lista será un **Object[]** con las columnas seleccionadas.

Usando DTOs con constructor

```
public class AlumnoDTO {
    private String nombre;
    private int edad;

    public AlumnoDTO(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }
}

@Query("SELECT new com.ejemplo.dto.AlumnoDTO(a.nombre, a.edad) FROM
Alumno a WHERE a.edad > :edad")
List<AlumnoDTO> findDTOByEdad(@Param("edad") int edad);
```

Ventaja: recibes objetos tipados directamente, evitando **Object[]**.

2.6 Consultas con joins y relaciones

Si **Alumno** tiene una relación con **Curso**:

```
@Entity
public class Alumno {
    @Id
    private Long id;

    @ManyToOne
    private Curso curso;
}
```

JPQL permite hacer joins:

```
@Query("SELECT a FROM Alumno a JOIN a.curso c WHERE c.nombre = :curso")
List<Alumno> findByCurso(@Param("curso") String nombreCurso);
```

- Hibernate traducirá el **JOIN** a SQL con la clave foránea correspondiente.
- Se puede usar **JOIN FETCH** para evitar LazyInitializationException al cargar colecciones.

2.7. Funciones y agregaciones

- **COUNT, SUM, AVG, MAX, MIN**

```
@Query("SELECT COUNT(a) FROM Alumno a WHERE a.edad > :edad")
long contarMayores(@Param("edad") int edad);
```

UPPER, LOWER, CONCAT, LENGTH

```
@Query("SELECT a FROM Alumno a WHERE UPPER(a.nombre) LIKE
:nombre")
List<Alumno> findByNombreIgnoreCase(@Param("nombre") String
nombre);
```

2.8. Ordenación y paginación

Spring Data JPA permite combinar JPQL con **Sort** y **Pageable**:

```
@Query("SELECT a FROM Alumno a WHERE a.edad > :edad")
Page<Alumno> findMayoresDe(@Param("edad") int edad, Pageable pageable);
```

Uso:

```
Pageable pageable = PageRequest.of(0, 10,
Sort.by("nombre").ascending());
Page<Alumno> pagina = repo.findMayoresDe(18, pageable);
```

- Hibernate genera el **LIMIT/OFFSET** automáticamente según el motor.
- El resultado es un **Page<Alumno>** que incluye total de elementos y total de páginas.

2.9 Buenas prácticas

1. Usar **parámetros nombrados** siempre que sea posible.

2. Usar DTOs con constructor para evitar retornar listas de **Object[]**.

3. Evitar consultas JPQL demasiado complejas; para eso podrían usar nativas.

4. Recordar que JPQL trabaja con entidades gestionadas, por lo que modificaciones no necesitan **@Modifying**.

5. Para joins, usar **JOIN FETCH** si se quiere inicializar colecciones relacionadas en la misma consulta.

3. Consultas derivadas de métodos (Spring Data JPA / Hibernate)

3.1 Introducción

Spring Data JPA permite crear consultas automáticas simplemente definiendo el nombre del método en la interfaz del repositorio.

No hace falta escribir SQL ni usar **@Query**:
Spring analiza el nombre del método y genera la consulta JPQL correspondiente, que Hibernate luego traduce a SQL nativo según el motor de base de datos.

Esto se conoce como **Consultas derivadas de método** (Derived Query Methods).

Ejemplo básico

```
public interface AlumnoRepository extends JpaRepository<Alumno, Long> {

   List<Alumno> findAllByApellidoEqualsAndNombreLike(String apellido, String nombre);
}
```

Spring Data JPA:

1. Analiza el nombre del método (**findAllByApellidoEqualsAndNombreLike**).

Traduce automáticamente a JPQL:

```
SELECT a FROM Alumno a WHERE a.apellido = :apellido AND a.nombre LIKE :nombre
```

2. Hibernate ejecuta el SQL equivalente sobre la base de datos.

Resultado: sin escribir ni una línea de SQL o JPQL.

3.2 Estructura general del nombre de método

El formato básico es:

```
<acción>By<Propiedad1><Operador1>[And|Or]<Propiedad2><Operador2>...
```

Donde:

Parte	Ejemplo	Descripción
Acción	***find, read, get, count, exists, delete***	Indica qué hace la consulta
Propiedad	***Nombre, Edad, Apellido***	Corresponde al atributo de la entidad
Operador	***Equals, Like, Between, GreaterThan, In, IsNull…***	Define el tipo de comparación
Conector	***And, Or***	Permite combinar condiciones

3.3 Operadores más comunes

Operador	Descripción	Ejemplo	Equivalente SQL
Equals (implícito)	Igualdad exacta	*findByNombre(String n)*	*WHERE nombre = ?*
Like	Coincidencia parcial	*findByNombreLike(String n)*	*WHERE nombre LIKE ?*
StartingWith	Empieza con	*findByNombreStartingWith("A")*	*WHERE nombre LIKE 'A%'*
EndingWith	Termina con	*findByNombreEndingWith("z")*	*WHERE nombre LIKE '%z'*
Containing	Contiene	*findByNombreContaining("ar")*	*WHERE nombre LIKE '%ar%'*
GreaterThan, LessThan, Between	Comparaciones numéricas o de fechas	*findByEdadBetween(18,25)*	*WHERE edad BETWEEN 18 AND 25*
IsNull, IsNotNull	Nulos	*findByApellidoIsNull()*	*WHERE apellido IS NULL*
In, NotIn	Listas de valores	*findByNombreIn(List<String> nombres)*	*WHERE nombre IN (...)*
OrderBy	Ordenación	*findByEdadGreaterThanOrderByNombreAsc(int e)*	*ORDER BY nombre ASC*

3.4 Parámetros y coincidencia

Los parámetros del método se asignan automáticamente a los nombres y posiciones según el orden en el que aparecen.

Ejemplo:

```
List<Alumno> findAllByEdadGreaterThanAndApellidoEquals(int edad, String apellido);
```

JPQL generado internamente:

```
SELECT a FROM Alumno a WHERE a.edad > ?1 AND a.apellido = ?2
```

3.5 Consultas con paginación y ordenación

Puedes combinar estos métodos con los tipos Pageable y **Sort** de Spring Data.

Paginación y ordenación:

```
Page<Alumno> findAllByEdadGreaterThan(int edad, Pageable pageable);
```

Uso:

```
Pageable pageable = PageRequest.of(0, 5, Sort.by("nombre").ascending());
Page<Alumno> pagina = repo.findAllByEdadGreaterThan(18, pageable);
```

Hibernate genera automáticamente la cláusula **LIMIT** o **FETCH FIRST** según el motor.

3.6 Contadores y verificadores

Spring Data JPA también genera consultas para contar o verificar existencia:

```
long countByEdadGreaterThan(int edad);
boolean existsByNombreAndApellido(String nombre, String apellido);
```

Esto se traduce a:

```
SELECT COUNT(*) FROM alumnos WHERE edad > ?;
SELECT CASE WHEN COUNT(*) > 0 THEN true ELSE false END FROM alumnos
WHERE ...
```

3.7 Eliminaciones y actualizaciones automáticas

Para eliminar registros:

```
@Modifying
@Transactional
void deleteByEdadLessThan(int edad);
```

Importante:

- Las consultas **DELETE** y **UPDATE** deben ir con **@Modifying** y **@Transactional**.
- Hibernate las ejecuta directamente en la base de datos.

3.8 Navegación por relaciones (joins implícitos)

Spring Data JPA interpreta las relaciones de entidad para consultas anidadas.

Ejemplo:

```
List<Alumno> findByCursoNombre(String nombreCurso);
```

JPQL generado:

```
SELECT a FROM Alumno a JOIN a.curso c WHERE c.nombre = :nombreCurso
```

Sin escribir manualmente el **JOIN**.

3.9 Ejemplos prácticos combinando operadores

```
// 1 Todos los alumnos de apellido "Pérez" y nombre que contiene "an"
List<Alumno> findAllByApellidoEqualsAndNombreContaining(String apellido, String nombre);
```

```java
// 2 Alumnos entre 18 y 25 años, ordenados por edad descendente
List<Alumno> findAllByEdadBetweenOrderByEdadDesc(int edadMin, int edadMax);

// 3 Verificar si existe algún alumno con cierto apellido
boolean existsByApellido(String apellido);

// 4 Eliminar los menores de edad
@Modifying
@Transactional
void deleteByEdadLessThan(int edad);
```

3.10 Ventajas y desventajas

Ventajas	Desventajas
No requiere escribir SQL ni JPQL	Limitado a consultas relativamente simples
Totalmente tipado y validado en compilación	Los nombres largos pueden ser difíciles de leer
Muy rápido de implementar	Menos control sobre el SQL generado
Compatible con paginación, ordenación y transacciones	No ideal para consultas complejas o con subconsultas