Material descargable

JPA: Java Persistence Query Language

egg

Java Persistence Query Language

JPQL (Java Persistence Query Language) es un lenguaje de consulta orientado a objetos, independiente de la plataforma, que forma parte integral de la especificación **Java Persistence API (JPA)**. Su principal propósito es permitir la realización de consultas sobre las entidades gestionadas por JPA, las cuales están mapeadas a una base de datos relacional.

A diferencia de SQL, que trabaja directamente con las tablas y registros de la base de datos, **JPQL** opera a un nivel más abstracto, permitiéndote trabajar con las clases y objetos que representan esas tablas en el contexto de una aplicación Java. Esto significa que, en lugar de utilizar nombres de tablas y columnas, trabajarás con los nombres de las clases y atributos definidos en tu código Java. De esta manera, JPQL te permite interactuar de manera directa y natural con la información nativa de los objetos, aprovechando la orientación a objetos del lenguaje Java.

Aunque está fuertemente inspirado en SQL y comparte gran parte de su sintaxis, **JPQL** se enfoca en las entidades de JPA, brindando una forma más intuitiva y alineada con la programación orientada a objetos para gestionar y consultar datos persistentes.

Cláusulas SELECT – FROM en JPQL

La cláusula **FROM** define de qué entidades se seleccionarán los datos. En JPA, las entidades están mapeadas a tablas en la base de datos, lo que significa que, en lugar de utilizar los nombres de las tablas, trabajarás con los nombres de las entidades en tu código Java. De igual manera, los atributos de las entidades sustituyen a las columnas de las tablas.

La sintaxis de una cláusula **FROM** en JPQL es similar a la de SQL, pero en vez de manejar nombres de tablas y columnas, trabaja con el modelo de entidad. Un ejemplo sencillo de una consulta JPQL que selecciona todas las entidades Autor es el siguiente:

```
SELECT a FROM Autor a;
```

En este caso, se hace referencia a la entidad **Autor** en lugar de la tabla correspondiente, y se le asigna el alias a. Este alias actúa como una variable identificadora que puede usarse en el resto de la consulta para hacer referencia a la entidad. El alias es similar a una variable en tu código Java y es esencial para simplificar el manejo de las entidades en las consultas.

Por ejemplo, si deseas seleccionar atributos específicos de la entidad Autor en lugar de todos los campos, puedes utilizar el alias para hacer referencia a esos atributos:

```
SELECT a.nombre, a.apellido FROM Autor a;.
```

En este caso, la consulta selecciona únicamente los atributos nombre y apellido de la entidad Autor, demostrando la capacidad de JPQL para trabajar directamente con los atributos de los objetos, sin necesidad de preocuparse por los nombres de las columnas subyacentes en la base de datos.

Cláusula `WHERE` en JPQL

La cláusula **`WHERE`** en JPQL se utiliza para filtrar los resultados de una consulta, similar a cómo se hace en SQL. Aunque la sintaxis es casi idéntica, **JPQL** admite solo un subconjunto de las características de SQL, ya que está diseñado para trabajar con entidades y no directamente con tablas.

En JPQL, puedes utilizar un conjunto de operadores básicos para definir expresiones de comparación. La mayoría de estos operadores son idénticos a los de SQL, y puedes combinarlos con los operadores lógicos **`AND`**, **`OR`**, y **`NOT`** para crear expresiones más complejas.

Ejemplos de operadores comunes en JPQL:

- **Igualdad:** `SELECT a FROM Autor a WHERE a.id = 10;`
- **Desigualdad:** `SELECT a FROM Autor a WHERE a.id <> 10;`
- **Mayor que:** `SELECT a FROM Autor a WHERE a.id > 10;`
- **Mayor o igual que:** `SELECT a FROM Autor a WHERE a.id >= 10;`
- **Menor que:** `SELECT a FROM Autor a WHERE a.id < 10;`
- **Menor o igual que:** `SELECT a FROM Autor a WHERE a.id <= 10;`
- **Rango (`BETWEEN`):** `SELECT a FROM Autor a WHERE a.id BETWEEN 5 AND 10;`
- **Patrones (`LIKE`):** `SELECT a FROM Autor a WHERE a.firstName LIKE '%and%';`
- **Valor nulo (`IS NULL`):** `SELECT a FROM Autor a WHERE a.firstName IS NULL;`

- Puedes negar esta condición utilizando **`NOT`**: `SELECT a FROM Autor a WHERE a.firstName IS NOT NULL;`
- **Conjunto (`IN`):** `SELECT a FROM Autor a WHERE a.firstName IN ('John', 'Jane');` (Esta consulta seleccionará todos los autores cuyo nombre sea "John" o "Jane").

Unir Entidades (`JOIN`) en JPQL

Si necesitas seleccionar datos de más de una entidad, por ejemplo, todos los libros escritos por un autor, debes unir las entidades en la cláusula **`FROM`**. La forma más sencilla de hacerlo es utilizando las asociaciones definidas en las entidades. A continuación, se muestra un ejemplo de cómo se realiza un `JOIN` en JPQL:

```
SELECT l FROM Libro l JOIN l.autor a;
```

En este caso, la entidad **`Libro`** está unida a la entidad **`Autor`** a través de la relación definida en su mapeo. No es necesario especificar una declaración **`ON`**, ya que las asociaciones entre entidades en JPA proporcionan toda la información requerida para hacer la unión.

También puedes utilizar el operador **`.`** para navegar a través de los atributos de la entidad relacionada y filtrar los resultados. Por ejemplo, si quieres seleccionar los libros escritos por un autor con un nombre específico, puedes hacerlo de la siguiente manera, sin necesidad de un `JOIN` explícito:

```
SELECT l FROM Libro l WHERE l.autor.nombre LIKE 'Homero';
```

Esto crea una relación implícita entre las dos entidades, utilizando la asociación definida entre ellas.

Crear Consultas con EntityManager

El **`EntityManager`** permite crear consultas dinámicas con JPQL mediante el método **`createQuery(query)`**. Este método recibe una consulta en JPQL, la envía a la base de datos vinculada con la Unidad de Persistencia, y devuelve el resultado de la consulta.

Para obtener los resultados de la consulta, puedes usar dos métodos principales de la clase `Query`:

- **`getResultList()`**: Captura todos los resultados y los almacena en una lista.
- **`getSingleResult()`**: Se usa cuando esperas un único resultado de la consulta.

Aunque **`getSingleResult()`** puede ser útil, se recomienda usar **`getResultList()`**, ya que si la consulta devuelve más de un resultado, **`getSingleResult()`** lanzará una excepción. Para evitar posibles errores, es mejor usar **`getResultList()`** en la mayoría de los casos.

Ejemplo de uso `createQuery`:

```
List<Autor> autores = em.createQuery("SELECT a FROM Autor a", Autor.class)

                    .getResultList();
```

En este ejemplo, se obtiene una lista de todos los autores almacenados en la base de datos. La consulta **`SELECT a FROM Autor a`** devuelve todas las instancias de la entidad `Autor`, que luego son capturadas en una lista mediante el método **`getResultList()`**.

Consideraciones sobre el uso de consultas sin parámetros

Es posible construir consultas sin usar parámetros, insertando valores literales directamente en la consulta JPQL. Sin embargo, esta práctica puede ser insegura, ya que te expones a vulnerabilidades como la inyección de SQL si los valores provienen de fuentes externas sin validación.

Por ejemplo, podrías tener una consulta como esta, en la que se inserta directamente el valor de **`nombreABuscar`**:

```
List<Cliente> clientes = em.createQuery("SELECT c FROM Cliente c WHERE
c.nombreContacto LIKE '%" + nombreABuscar + "%'", Cliente.class)

                        .getResultList();
```

Si bien esto funciona, **no es recomendable** en aplicaciones que interactúan con datos externos, ya que puede comprometer la seguridad. Por esta razón, es preferible usar **parámetros** en las consultas, lo cual se verá más adelante.

Otros Ejemplos de Consultas JPQL con `EntityManager`:

- **Consulta con cláusula `WHERE`:** En este caso, se obtienen todos los libros cuyo autor tiene el nombre "Homero".

```
List<Libro> libros = em.createQuery("SELECT l FROM Libro l WHERE l.autor.nombre = 'Homero'", Libro.class)
                       .getResultList();
```

- **Consulta con ordenamiento (`ORDER BY`):** Aquí, la consulta devuelve una lista de libros ordenados alfabéticamente por su título.

```
List<Libro> librosOrdenados = em.createQuery("SELECT l FROM Libro l ORDER BY l.titulo ASC", Libro.class)
                                .getResultList();
```

Navegación vs. JOIN en JPQL

Cuando trabajas con entidades que están relacionadas entre sí mediante asociaciones en JPA, puedes navegar directamente por esas relaciones sin la necesidad de hacer un **`JOIN`** explícito. Esto simplifica el código y mejora la legibilidad de la consulta.

Por ejemplo, si tienes una entidad **`Libro`** que tiene una relación con **`Autor`**, puedes acceder a los atributos de la entidad **`Autor`** a través del objeto **`Libro`** sin necesidad de hacer un **`JOIN`**.

Ejemplo de consulta sin `JOIN` explícito:

```
List<Libro> librosConAutor = em.createQuery("SELECT l FROM Libro l WHERE l.autor.nombre = 'Gabriel García Márquez'", Libro.class)
```

```
                                .getResultList();
```

En este ejemplo, no se utiliza un **`JOIN`** porque la relación entre **`Libro`** y **`Autor`** ya está mapeada. Navegamos directamente desde **`Libro`** hacia **`Autor`** usando **`l.autor.nombre`**, lo cual es más limpio y eficiente.

¿Cuándo deberías usar un `JOIN` explícito?

Aunque puedes navegar por relaciones de entidades en muchas consultas, en algunos casos es necesario o más claro usar un **`JOIN`** explícito:

- **Consultas más complejas**: Cuando necesitas combinar varias entidades que no están directamente relacionadas o cuando trabajas con relaciones más complejas.
- **Uniones externas (`LEFT JOIN`)**: Si quieres traer resultados donde una entidad no tiene una relación, pero aún deseas incluirla en los resultados, necesitas un **`LEFT JOIN`**.

Ejemplo con `LEFT JOIN` explícito: En este caso, la consulta devolvería libros que no tienen un autor asociado, utilizando un **`LEFT JOIN`**

```
List<Libro> librosConAutoresOpcionales = em.createQuery("SELECT l FROM Libro l
LEFT JOIN l.autor a WHERE a.nombre IS NULL", Libro.class)
                                        .getResultList();
```

Agregar Parámetros a Consultas JPQL

Para hacer que las consultas JPQL sean dinámicas, es necesario usar parámetros en lugar de valores literales. Esto permite que la consulta se adapte a diferentes valores sin necesidad de reescribir el código de la consulta cada vez.

Uso de Parámetros en JPQL

En una consulta JPQL, puedes definir parámetros que se reemplazan por valores en tiempo de ejecución. Esto se realiza utilizando el método **`setParameter()`** de la clase **`Query`**.

Supongamos que tienes la siguiente consulta:

```
String jpql = "SELECT p FROM Persona p WHERE p.edad = :edad";
```

Aquí, **`:edad`** es un parámetro de la consulta que debe ser reemplazado por un valor real. Para hacer esto, se utiliza **`setParameter()`**:

```
int edadBuscada = 20;

List<Persona> personas = em.createQuery(jpql, Persona.class)

                          .setParameter("edad", edadBuscada)

                          .getResultList();
```

En este ejemplo:

- **`createQuery(jpql, Persona.class)`**: Crea la consulta JPQL.
- **`setParameter("edad", edadBuscada)`**: Asigna el valor de la variable **`edadBuscada`** al parámetro **`:edad`** en la consulta.

Este enfoque permite que la consulta sea flexible y reutilizable con diferentes valores para el parámetro **`:edad`**. En este ejemplo, la consulta busca todas las personas con la edad de 20 pero puedes cambiar el valor de **`edadBuscada`** para buscar personas de diferentes edades, sin modificar la consulta JPQL.

Beneficios de Usar Parámetros

1. **Seguridad**: Usar parámetros ayuda a prevenir ataques de inyección de SQL, ya que el valor del parámetro es tratado de forma segura.
2. **Flexibilidad**: Puedes cambiar el valor del parámetro sin modificar la consulta misma.
3. **Reusabilidad**: Las consultas parametrizadas son más fáciles de mantener y reutilizar en diferentes contextos.