JPA (II)

Entity Graph, Herencia y Consultas

Aplicaciones Distribuidas

Curso 2025/2026

Entity Graph

- En las asociaciones entre entidades, la lectura perezosa o inmediata depende del tipo de asociación y/o del uso de las anotaciones `FetchType.LAZY` y `FechaType.EAGER`.
- Este tipo de configuración es estática y no permite el cambio de estrategia en tiempo de ejecución.

```
@OneToMany(mappedBy = "editorial", fetch=FetchType.LAZY)
private ArrayList<Empleado> empleados;

@OneToOne(fetch=FetchType.LAZY)
private Empleado director;
```

- `FetchType.LAZY` no es **obligatorio**. El proveedor de persistencia puede tenerlo en consideración o no. Además, si se usa en atributos que son `EAGER` en **JSE** por defecto requiere el uso de `weaving`, por lo que normalmente se desactivará.

Entity Graph

- El objetivo principal al utilizar un `Entity Graph` es mejorar el rendimiento en tiempo de ejecución al cargar las asociaciones y campos básicos de la entidad.
- El proveedor JPA carga todos los grafos en una consulta SELECT y evita el uso de consultas adicionales para obtener las asociaciones.
- Se pueden definir con anotaciones.

```
@NamedEntityGraph(
  name = "editorial-empleados",
  attributeNodes = {
    @NamedAttributeNode("nombre"),
    @NamedAttributeNode("fechaFundacion"),
    @NamedAttributeNode("empleados"),
    @NamedAttributeNode("director")
  }
)
```

Entity Graph

- Se utilizan subgrafos para definir la carga de las entidades implicadas en la asocación.

```
@NamedEntityGraph(
  name = "editorial-empleados",
  attributeNodes = {
    @NamedAttributeNode("nombre"),
    @NamedAttributeNode("fechaFundacion"),
    @NamedAttributeNode("empleados"),
    @NamedAttributeNode(value="director", subgraph="empleado-director")
  },
  subgraphs = {
    @NamedSubgraph(
      name = "empleado-director",
      attributeNodes = {
        @NamedAttributeNode("plaza-garaje")
        //...
      }
    )
  }
)
```

Entity Graph

- Podemos usarlo en la operación `find`:

```
EntityGraph entityGraph = em.getEntityGraph("editorial-empleados");
Map<String, Object> properties = new HashMap<>();
properties.put("javax.persistence.loadgraph", entityGraph);
Editorial editorial = em.find(Editorial.class, id, properties);
```

- `javax.persistence.loadgraph` carga todos los campos por defecto y los especificados en el grafo.
- `javax.persistence.fetchgraph` carga solo los campos especificados en el grafo (necesita *weaving* habilitado).
- Ambos cargan siempre la clave primaria.

Entity Graph

- Los grafos también se pueden definir con configuración XML o con el API de JPA:

```
EntityGraph<Editorial> entityGraph = em.createEntityGraph(Editorial.class);
entityGraph.addAttributeNodes("nombre");
entityGraph.addAttributeNodes("fechaFundacion");
entityGraph.addAttributeNodes("empleados");
entityGraph.addSubgraph("director")
  .addAttributeNodes("plaza-garaje");
```

Herencia

- JPA permite gestionar como se mapean los objetos cuando interviene el concepto de herencia:
  - Una tabla por familia
  - Estrategia ***Joined*** (una tabla por cada clase)
  - Una tabla por clase concreta.
- Por defecto se usa la estrategia de una tabla por familia

Herencia: Una tabla por familia

- Todas las clases que forman parte de una misma jerarquía son almacenadas en una única tabla
- Aparece una columna adicional donde se almacena el tipo de clase a la que hace referencia cada fila (valor discriminante).

```
@Entity
@Inheritance( strategy = InheritanceType.SINGLE_TABLE )
@DiscriminatorColumn ( name = "tipo", discriminatorType
= DiscriminatorType.CHAR )
@DiscriminatorValue ("C")
public class SuperClase { //...
}
@Entity
@DiscriminatorValue ("S")
public class SubClase extends SuperClase { //...
}
```

Herencia: Estrategia *Joined*

- Cada clase y subclase será almacenada en su propia tabla
- La tabla raíz tiene las columnas comunes y cada subclase almacena sus atributos propios en cada subtabla.
- La tabla raíz también tiene una columna con valor discriminante.

```
@Entity
@Inheritance ( strategy = InheritanceType.JOINED )
public class SuperClase { //...
}
```

Herencia: Una tabla por clase concreta

- Cada clase que no sea abstracta será almacenada en su propia tabla (incluyendo atributos propios y heredados).
- Ya no hay tablas compartidas, columnas compartidas o columna con valor discriminante.

```
@Entity
@Inheritance ( strategy = InheritanceType.TABLE_PER_CLASS )
public class SuperClase { //...
}
```

Herencia: MappedSuperclass

- Podemos tener una clase padre para agrupar atributos comunes de las sublcases, pero no querer que esta se mapee en JPA usando la anotación `@MappedSuperclass`.
- Solo las subclases se mapean con sus propios atributos y los heredados.
- Simplifica el diseño si no necesitas trabajar directamente con la clase padre.
- No permite realizar consultas JPQL sobre la clase padre directamente.

```
@MappedSuperclass
public class SuperClase { //...
    @id
    String id;
    //..
}
@Entity
public class SubClase extends SuperClase{

}
```

Java Persistence Query Language (JPQL)

- El lenguaje de consulta principal en JPA es JPQL. Permite hacer consultas sobre el modelo lógico de entidades en lugar de sobre el modelo físico. Sintaxis similar a SQL.
  - Resultados de único y múltiple valor.
  - Funciones de agregación, orden y agrupamiento.
  - Sintaxis de join más natural con el modelo lógico.
- Tipos de consultas
  - Consultas dinámicas
  - Consultas nombradas: se crean con `em.createNamedQuery`.
- Tambien consultas nativas en SQL.

Uso de parámetros en JPQL

- A la hora de construir consultas en JPQL, lo adecuado es definir parámetros en lugar de concatenar cada texto con las entradas de usuario.

```
//sin parámetros
public List<Empleado> getEmpleados(String dpto){
    String queryString = "SELECT e " +
                         "FROM Empleado e "+
                         "WHERE e.departamento = "+dpto;
    Query query = em.createQuery(queryString);
    query.getResultList();
}
//con parámetros
public List<Empleado> getEmpleados(String dpto){
    String queryString = "SELECT e " +
                         "FROM Empleado e "+
                         "WHERE e.departamento = :departamento";
    Query query = em.createQuery(queryString);
    query.setParameter("departamento", dpto);
    query.getResultList();
}
```

Definición de JOINS

- En JPQL podemos definir ***JOIN*** entre entidades de una manera similar a SQL o aprovechando la definición lógica de las entidades:

```
SELECT dir.nombre
FROM Editorial ed, Empleado dir
WHERE dir = e.director AND ed.id = :id;

SELECT dir.nombre
FROM Editorial ed
INNER JOIN ed.director dir
WHERE ed.id = :id;
```

- Existe la opción de definirlas nativas (los parámetros se definen con números):

```
SELECT dir.nombre
FROM editorial ed, empleado dir
WHERE dir.id = e.director AND ed.id = ?1;
```

Consultas nombradas

- La anotación `@NamedQuery` permite definir consultas para luego ejecutarlas por su nombre.

```
@Entity
@NamedQuery ( name =" Empleado.getEmpleados ", query =" SELECT e FROM
Empleado e")
public class Empleado { //...
}
//...
Query query = em.createNamedQuery("Empleado.getEmpleados");
```

- Al igual que las consultas dinámicas, las consultas nombradas permiten la definición de parámetros que se pueden instanciar en el momento de la ejecución.
- Las consultas nombradas también pueden ser nativas `@NamedNativeQuery`.

Ejecución de consultas

- Métodos del objeto `EntityManager`:
  - `createQuery`: para crear consultas dinámicas.
  - `createNamedQuery`: para crear consultas nombradas.
  - `createNativeQuery`: para crear consultas nativas.
- Para obtener los resultados de la ejecución de las consultas, a partir del objeto `query`:
  - `getResultList()`: para colecciones de elementos.
  - `getSingleResult()`: para consultas que devuelven un único elemento.

Devolución de resultados

El tipo devuelto por una consulta JPQL dependerá de la consulta.

- Devuelve un colección de objetos `Editorial`:

```
SELECT e FROM Editorial e;
```

- Devuelve una colección de `String`:

```
SELECT e.nombre FROM Editorial e;
```

- Devuelve una colección de `Object[]`:

```
SELECT e.nombre, e.fechaFundacion FROM Editorial e;
```

Ejecución de consultas

Si sabemos de antemano el tipo que va a devolver la consulta, se puede crear una `TypedQuery`:

```
String queryString = "SELECT e FROM Editorial e where e.id = :id";
TypedQuery<Editorial> typedQuery = em.createQuery(queryString, Editorial.class);
typedQuery.setParameter("id", id);
return typedQuery.getSingleResult();
```

- Las consultas se pueden paginar:
  - `setFirstResult()`: especifica el primer resultado que debería recuperarse.
  - `setMaxResults()`: especifica el máximo número de resultados que deberían recuperarse.

Ejecución de consultas - EntityGraph

- Podemos ejecutar una consulta haciendo uso de un EntityGraph:

```
EntityGraph entityGraph = em.getEntityGraph("editorial-empleados");
String queryString = "select e from Editorial e where e.id = :id";
Query query = em.createQuery(queryString);
query.setParameter("id", id);

query.setHint("javax.persistence.loadgraph", entityGraph);

return query.getSingleResult();
```

- Si no usamos un `EntityGraph` se puede forzar de forma estática una lectura **eager** en una consulta JPQL:

```
SELECT e FROM Editorial
INNER JOIN FETCH e.empleados;
```

Criteria API

- JPA también ofrece una API Java para la construcción de consultas de forma más programática.

```
SELECT e
FROM Empleado e
WHERE e.nombre = 'Juan Juan'
```

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Empleado> c = cb.createQuery(Empleado.class);
Root<Empleado> emp = c.from(Empleado.class);
c.select(emp)
.where(cb.equal(emp.get("nombre"), "Juan Juan"));
```

Otros aspectos de JPA

- `orphanRemoval` : para eliminar entidades que ya no están ligadas a su padre:

```
@Entity
public class Editorial {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private String id;


    @OneToMany(orphanRemoval = true, cascade = CascadeType.REMOVE)
    private List<Empleado> empleados;
```

- Si borramos una editorial se borrarán todos los empleados debido a `CascadeType.REMOVE`, pero si lo que hacemos es eliminar un empleado de la lista `empleados` de la editorial, la editorial seguirá existiendo, pero el empleado se eliminará debido a `orphanRemoval = true`.

Otros aspectos de JPA

- `@PrePersist`, `@PostPersist`, `@PreUpdate`, `@PostUpdate`, `@PreRemove`, `@PostRemove`:
  - Se utilitan para anotar métodos de una entidad de manera que se ejecuten cuando surgen dichos eventos durante el ciclo de vida de la entidad.
- `@OrderBy`: Ejemplo `@OrderBy("nombre DESC")`: añadida a atributos de tipo colección con anotaciones de asociación, permite recuperar dichas colecciones en un orden específico.
- `@MapKey`: Para asociaciones con colección de tipo mapa (Map<Key, Value>). Define qué campo o propiedad de la entidad relacionada se usará como clave en el mapa.

```
@OneToMany
@MapKey(name = "nss")  // Usa nss del empleado como clave del mapa
private Map<String, Empleado> empleados;
```