**M06\_EAC3\_(JPA-PostgreSQL)**

**Cosas a tener en cuenta:**

Anotaciones JPA:

1. En cada clase hay que indicar que se trata de una entidad (@Entity).
2. Hay que indicar el tipo de herencia:
3. **@Inheritance(strategy=estrategia)**:
   1. JOINED: Cada clase es una tabla propia. La clase base utilizará una tabla con un campo para cada uno de sus atributos y otra tabla que indica la subclase del objeto que está representado. Cada subclase utilizará otra con un campo para el identificador y además, un campo adicional para cada uno de los atributos específicos.
   2. SINGLE\_TABLE: Los objetos de todas las clases que intervienen son representados en una única tabla que tiene un campo para cada dato común, otro que indica el tipo de objeto que está representando la fila de la tabla y, además, un campo para cada dato de alguna de las subclases.
   3. TABLE\_PER\_CLASS: Cada clase que interviene tiene asociada una tabla con un campo para cada dato del objeto, tanto para los datos heredados como para los específicos.
4. Si la estrategia es JOINED o SINGLE\_TABLE, podemos indicar más características (opcionales):

**@DiscriminatorColumn(name="nomColumn", discriminatorType=tipo, length=mida)** 🡪 Se pone a la clase base, a continuación de la anotación “@Inheritance (herencia)”. Indica las características del campo que indicará la clase del objeto. Los valores que pueden indicarse son:

* **nomColumn:** es el nombre que se da al campo de la base de datos (opcional).
* **tipo:** es el tipo de campo (opcional). Su valor por defecto es String. Puede tener valor CHAR e INTEGER.
* **mida:** sólo se tiene en cuenta si el tipo es “String” e indica la longitud máxima de las cadenas de caracteres.

**@DiscriminatorValue(valor)** 🡪 Se puede indicar tanto a la clase base como a las subclases. Es opcional. Se pone a continuación de las anotaciones anteriores o, si no hay ninguna, a continuación de la anotación “@Entity”. Indica el valor que tomará el campo de la base de datos que indica la clase de objetos para esta clase en concreto. *En el ejercicio, sólo lo he puesto en las “subclases” y el valor fue de tipo “CHAR”. La clase “ProducteElaborat” tiene el valor “E” y la clase “MateriaPrimera” tiene el valor “M”.*

1. La siguiente anotación sirve para vincular la navegabilidad o las relaciones que hay entre clases. Hay varias opciones:
   1. @OneToOne 🡪 Relación de 1 a 1:
      1. Si en una clase A tenemos una relación 1-1 navegable en un sentido con un atributo de campo “prívate A relación;”, la anotación encima del atributo o del “getter” tendría que ser @OneToOne.
      2. Si tenemos una relación 1-1 navegable en los dos sentidos, entre 2 clases, tendremos un atributo o campo de clase relacionado en cada clase, por ejemplo “private A relación; (en la clase B)” y “private B vinculado; (en la clase A)”, tendremos que poner la anotación JPA en ambos atributos o getters: @OneToOne (clase A) y @OneToOne(mappedBy=”relación”) (clase B).

Resumen:

* Relación 1-1 navegable de B a A:
  + Como **B** puede navegar hacía **A**, la clase **B** tendrá un objeto de la clase **A**:
    - private A relacionado
  1. @ManyToOne 🡪 Relación 1-N:
     1. Navegable N a 1: Si tenemos una clase A y una clase B en la que la clase B (N) es navegable a la clase A, la clase B tendrá un objeto o instancia de la clase A como atributo de clase (private A relación;). La anotación JPA debe ir antes del atributo o del getter de la siguiente manera: @ManyToOne.
     2. Navegable en los dos sentidos: Si tenemos una clase A y una clase B, la clase B (N) tendrá una lista de objetos de la clase A (List<A> relación = new ArrayList();). La clase A sólo tendrá un objeto o instancia de la clase B (private B vinculada;). Por tanto la clase que tiene una lista de objetos, encima del atributo de clase o en su getter tendrá que indicar la anotación JPA @OneToMany(mappedBy=”vinculada”) y en la clase A, en el objeto o su getter tendrá que poner la anotación @ManyToOne.
  2. @ManyToMany 🡪 Relación M-N:
     1. Navegable en un solo sentido: Si tenemos una clase A y una clase B, una de las clases, la que consideremos como clase principal, tendrá una lista de objetos de la otra clase. Consideremos que la clase B es la clase principal y por tanto tendrá dicha lista (List<A> relacionada = new ArrayList();). La anotación en el campo de clase o su getter será @ManyToMany.
     2. Navegable en ambos sentidos: Si tenemos una clase A y una clase B, una de las clases será considerada de nuevo clase padre. Al ser una relación bidireccional, ambas clases tendrán una lista de objetos de la otra clase (Clase A 🡪 List<B> vinculada = new ArrayList(); -- Clase B 🡪 List<A> relacionada = new ArrayList();). Ahora bien, la clase padre mapeará la clase hija, en nuestro caso la clase padre es la B por lo que la anotación en el campo de clase o su getter será @ManyToMany(mappedBy=”vinculada”) y en la clase A, sólo indicaremos en el campo o getter la anotación JPA @ManyToMany.

**NOTA**: a la hora de poner la anotación JPA en las relaciones (@ManyToMany, @ManyToOne, etc…), es importante indicar el segundo argumento “cascade={CascadeType.ALL})”, Ejemplo:

**@ManyToMany(mappedBy="productesElaborats", cascade={CascadeType.ALL})**

1. Importante, todas las clases han de implementar la interfaz “Serializable”:
   1. public class Producte implements **Serializable {**

**Insertar, seleccionar u obtener, eliminar y actualizar datos:**

Insertar:

Para insertar datos los podemos hacer con lenguaje JPL, pero el profesor quiere que lo hagamos con el objeto EntityManager, por lo que si disponemos o tenemos acceso al ID o a un objeto (podremos obtener su ID), utilizaremos EntityManager. Ejemplo:

public void inserir(Producte prod) throws GestorException {

//TODO completar el metode

if(em.find(Producte.class, prod.getIdProd()) != null) {

throw new GestorException("Clave duplicada");

}

em.getTransaction().begin();

em.persist(prod);

em.getTransaction().commit();

}

**NOTA**: em 🡪 Objeto EntityManager

Utilizamos el objeto EntityManager (em) junto con su método “find” para obtener o apuntar a un objeto en concreto. Esto los conseguimos indicando los 2 argumentos que admite “find” (Class.class, id).

Una vez comprobamos que el objeto existe en la BD empezamos la transacción con “getTransaction().begin();”, continuamos haciendo persistente el objeto “persist(prod);” y finalizamos haciendo un commint de la transacción “getTransaction().commit();”.

Actualizar:

Es el mismo caso que para “insertar”, utilizaremos el objeto EntityManager si disponemos de un objeto o el ID del mismo. Ejemplo:

public void modificar(Producte prd) throws GestorException{

//TODO completar el metode

if(em.find(Producte.class, prd.getIdProd()) == null) {

throw new GestorException("No existe el producto");

}

em.getTransaction().begin();

em.merge(prd);

em.getTransaction().commit();

}

El caso es prácticamente idéntico al de insertar. Iniciamos transacción, continuamos (aquí es donde cambia) con la actualización, utilizamos “merge(prd);” y finalizamos con “commit”.

Eliminar:

Es el mismo caso que en los dos casos anteriores. Si disponemos de un ID o un objeto de la clase, iniciamos transacción, continuamos con “em.remove(objeto)” o en el caso de sólo disponer de un ID, “em.remove(em.find(Class.class, id));” y finalizamos con commit.

Seleccionar u Obtener:

Este caso es un poco particular, ya que depende de si queremos obtener un único objeto (debemos disponer al menos de su ID) o de un conjunto de objetos. Vemos ambos casos:

1. Obtener un objeto: en este caso es casi idéntico a los anteriores, es decir, utilizamos el objeto EntityManager. Ejemplo:
   1. Producte producte = em.find(Producte.class, id);
2. Obtener conjunto de objetos: este es el primer caso donde deberemos utilizar el lenguaje JPL (es muy similar a MySQL). Si queremos obtener una lista de todos los Productos de la base de datos debemos realizar las siguientes sentencias:

List<Producte> productes = null;

Query query = em.createQuery(“SELECT p FROM Producte p”);

Productes = query.getResultList();

Return productes;

Lo primero que hacemos es crear una lista de objetos de tipo “Producto”, después creamos una consulta (**Query query = em.createQuery(“SELECT p FROM Producte p”);**). El resultado de esa consulta o Query la asignamos a la lista que hemos creado anteriormente “Productes = query.getResultList();”.

1. Hay otra opción y es obtener una lista de objetos con unos determinados parámetros. Este parámetro no es un ID, por lo que debemos utilizar el objeto “Query”. Ejemplo:

public List<ProducteElaborat> obtenirProducteElaboratPerPreu(float preu) {

//TODO completar el metode

List<ProducteElaborat> productes = null;

Query q = em.createQuery("Select pe From ProducteElaborat pe Where pe.preu>=:preu");

q.setParameter("preu", preu);

productes = q.getResultList();

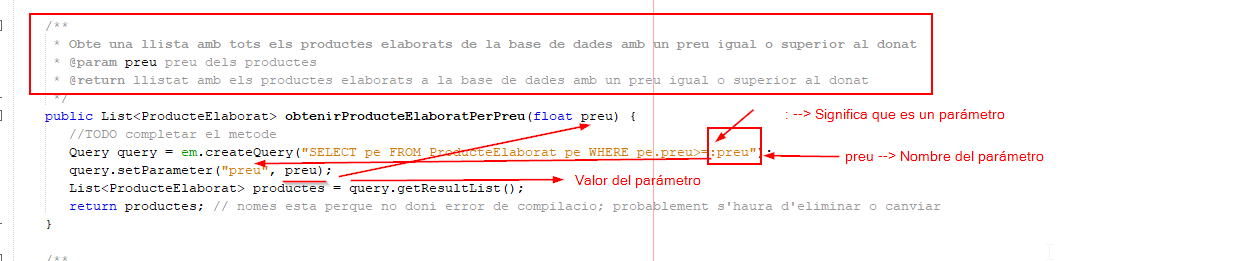
return productes;

}

Este método devuelve una lista de “Productos Elaborados” a partir de un precio que tiene que ser “mayor o igual (>=) al precio pasado por parámetro. Hay varias cosas importantes a tener en cuenta en este proceso:

* + - 1. Para indicar que al valor a evaluar, es decir el parámetro a evaluar es un valor externo (parámetro del método 🡪 preu) hay que hacerlo poniendo “:” después de la expresión “>=”, es decir: Where pe.preu**>=:**preu
      2. Si indicamos que le hemos pasado un parámetro, tenemos que decirle con el método “setParameter(“preu”, preu);” los valores del mismo. El primer argumento va entre comillas dobles y hace referencia al último valor de la sentencia JPL (preu). El segundo argumento el el valor de ese parámetro, en nuestro caso es el parámetro del método.

Después, asignamos el valor de la query a la lista de productes.



**NOTA**: todas las sentencias de acción (insert, update, delete) han de iniciar (getTransaction().begin()), continuar (.remove(), .persist()., merge(), executeUpdate) y finalizamos (getTransaction().commit()). Las transacciones se realizan con el objeto “EntityManager” y la inserción, update, delete, dependiendo de si utilizamos el objeto “Query” o el propio “EntityManager” se hará de una manera diferente:

* Con objeto “Query”:
  + Query.executeUpdate()
* Con el objeto “EntityManager”:
  + em.persist(), em.remove(), em.merge()