Manual Avanzado de JPA

Por:
Carmelo Navarro Serna

ÍNDICE

INTRODUCCIÓN 3

BASES DE DATOS 3

CONFIGURACIÓN DE JPA 5

TRANSACCIONES 10

MODIFICAR DATOS 13

CONCLUSIÓN 16

Introducción

El siguiente artículo está dirigido a personas que tienen un buen conocimiento de JPA. No voy a explicar que es JPA, para que sirve, como se configura, cual es la mejor implementación, no voy a explicar Javadoc, ni nada por el estilo. Si quieres algo como lo que enumerado anteriormente mejor mira otro artículo y/o tutorial; en la red hay muchos muy buenos.

Mi objetivo es explicar como utilizar JPA en una aplicación pero no desde el punto de vista solo de la programación sino explicar como utilizar JPA de forma correcta para resolver las situaciones más comunes que se presentan en una aplicación.

Mi intención es profundizar lo suficiente para poder ayudar en la construcción de aplicaciones proporcionando decisiones generales que se deben tomar desde el principio del desarrollo como el uso de las funciones del EntityManager, ciclo de vida de las transacciones, uso de las entidades, etc… Por este motivo voy a ir al grano y daré por entendido que el lector tiene un conocimiento mínimo de JPA.

El artículo es un manual para desarrolladores sobre la especificación de JPA 1.0. Para ilustrar ejemplos he utilizado la implementación OPENJPA 1.2.0.

Bases de datos

El primer problema que se nos plantea es la base de datos. A la hora de planificar el modelo de datos debemos seguir las siguientes pautas:

- La clave primaria mejor que sea un campo numérico por varios motivos:
    - La clave primaria NUNCA puede ser vista por el usuario de la aplicación donde estemos utilizando JPA. Esta prohibido mostrar una clave primaria por pantalla. Esto descarta campos mas pesados que los numéricos como los de texto.
    - El orden es el más obvio de todos los tipos de datos.
    - Permite su uso para operaciones aritméticas de forma fácil.
    - Permite el uso de secuencias.
    - Etc..

- Respetar la lógica de los nombres. Es decir, si la tabla **PERSONA** tiene una clave primaria que es el campo **ID** ó **ID_PERSONA** entonces si la tabla **TRABAJADOR** tiene un campo que hace referencia a la tabla persona se ha de llamar **ID_PERSONA** (no **PERSONA** ó **REF_PERS** ni nada por el estilo). Y esta regla se aplicaría a cualquier tabla que haga referencia a **PERSONA**.
- Si un campo de una tabla es clave ajena entonces DEBEMOS CREAR DICHA CLAVE AJENA. Esto puede parecer una tontería tener que puntualizarlo pero en muchos casos el equipo de desarrollo sabe que un campo es clave ajena pero la clave ajena como tal no esta creada.

- Buena estrategia de índices. No todos los campos pueden ser índices. Merece la pena hacer un esfuerzo por conocer bien el negocio de la aplicación y establecer los campos que van a ser índices en cada tabla. Yo te recomiendo el siguiente proceso:
    - Todos los campos que se utilizan como criterios de búsqueda desde la interfaz son índices. Ejemplo, si tienes una tabla **Persona** que tiene un campo **Id_Provincia** y tienes una pantalla de búsqueda de personas y sale el campo provincia entonces **Id_Provincia** es un índice.
    - Mira TODAS las consultas de la base de datos y si un campo se repite varias veces en consultas que tienen una alta frecuencia de uso entonces ese campo es índice. Ejemplo, si tienes una tabla **Persona** que tiene un campo **Id_Empresa** y en casi todas tus consultas sobre la tabla **Persona** haces JOIN con la tabla **Empresa** a través del campo **Id_Empresa** entonces Id_**Empresa** es un índice.

Todas las operaciones que se hacen de acceso a datos al final son consultas (‘**select**’) y modificaciones (‘**insert**’, ‘**update**’ y ‘**remove**’).

Por rendimiento siempre hay que ver cual es el tamaño de operaciones atómicas (que afectan solo a un registro) de modificación (‘**insert**’, ‘**update**’ y ‘**remove**’) que caben en una transacción. Es decir, en una transacción se pueden incluir cualquier número de operaciones pero dependiendo de muchos factores (tamaño de las tablas, relaciones entre tablas, numero de campos en los registros, etc..) a veces podremos meter 1000 y a veces solo 100. Lo que si que es seguro es que si creas una transacción por cada una de las operaciones estas perdiendo rendimiento.

Configuración de JPA

Vamos a analizar el fichero persistence.xml:

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
   http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
    version="1.0">
      <persistence-unit name="BD">

        <provider>
            org.apache.openjpa.persistence.PersistenceProviderImpl
        </provider>
        <jta-data-source>BDDataSource</jta-data-source>

        <class>es.trafico.sipp.neg.entity.jpa.Anotacion</class>
        <class>es.trafico.sipp.neg.entity.jpa.Comunicacion</class>

        <properties>
            <property name="openjpa.TransactionMode" value="local" />
            <property name="openjpa.ConnectionFactoryMode" value="local" />
            <property name="openjpa.jdbc.DBDictionary" value="oracle" />
            <property name="openjpa.jdbc.UpdateManager" value="constraint"/>
            <property name="openjpa.MetaDataFactory" value="jpa"/>

        </properties>
    </persistence-unit>
</persistence>
```

Opcional.Configuración de dataSource

Identificación de las entidades

Configuración de propiedades de la implementación

Vamos a analizar estas tres partes:

- Configuración del dataSource. Esta parte no tiene demasiado misterio. Si se quiere utilizar definir un dataSource para utilizar JPA (como por ejemplo para un servidor web) se debe definir.

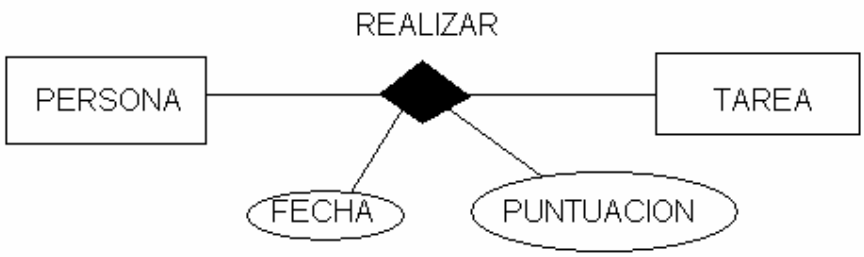
- Identificación de las entidades. Hay que definir las entidades que son las unidades de acceso a las tablas de base de datos. Realmente no hace falta identificar las entidades explícitamente en el persistence.xml pero es mejor ya que así es mas fácil saber cuales son las entidades que están en el contexto de JPA. Puede no parecer importante pero cuando una aplicación empieza a tener un número considerable de entidades tener controlado el conjunto de entidades en el contexto de JPA es una de esas cosas que es mejor que no sean automáticas (No dejar que JPA asuma cuales son las entidades).

  Respecto a identificación de entidades respecto a la base de datos, lo más aconsejable es crear una entidad por cada tabla a excepción de las tablas que sean para expresar la relación muchos a muchos entre otras dos tablas.

Si tenemos una tabla de relación "**REALIZAR**" entre las tablas "**PERSONA**" y "**TAREA**"

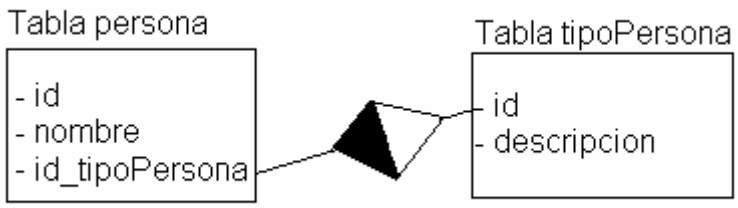
Entonces deberemos crear las entidades "**PERSONA**" y "**TAREA**" y en ellas definir una relación muchos a muchos (**@ManyToMany**) con una relación **@JoinTable** con la tabla "**REALIZAR**".

Pero ojo, si existieran campos de datos en la relación "**REALIZAR**" se tendrían que crear las tres entidades, es decir si se tiene:



A la hora de definir los campos en las entidades, los campos que no sean clave ajena a otra entidad se definen con tipos primitivos pero los campos que si sean clave ajena se pueden definir como campos primitivos del mismo tipo que la clave primaria de la tabla a la que se hace referencia o como del tipo entidad que represente la tabla a la que se hace referencia.

Si tenemos la relación



En esta relación siempre ha de mapearse la entidad "**TipoPersona"** de la siguiente manera:

```
@Entity
public class TipoPersona{
```

```
        @Id
        private String id;

        @Column(name="descripcion")
        private String descripcion;

        @OneToMany(mappedBy="tipoPersona")
        private Set<Persona> movimientos;

    }
```

Pero para la entidad persona pueden haber dos mapeos posibles:

1. Todos los campos como primitivos:

```
    @Entity
    public class Persona{

        @Id
        private String id;

        @Column(name="nombre")
        private String nombre;

        @Column(name="id_tipoPersona")
        private String tipoPersona;

    }
```

2. Indicando la relación entre clases:

```
    @Entity
    public class Persona{

        @Id
        private String id;

        @Column(name="nombre")
        private String nombre;

        @ManyToOne
        @JoinColumn(name="id_tipoPersona")
        private TipoPersona tipoPersona;

    }
```

Los dos mapeos son validos. Solo hay que tener en cuenta si queremos potencia o rendimiento. Me explico, indicando la relación entre la entidad "**Persona**" y la entidad "**TipoPersona**" en la entidad "**Persona**" siempre podemos conocer el tipo de persona al buscar una persona lo cual es mas potente pero si prestamos atención al log de JPA nos aparecerá un mensaje parecido al siguiente mensaje:

```
Creating subclass for "[class TipoPersona]". This means that
your application will be less efficient and will consume more
memory than it would if you ran the OpenJPA enhancer.
Additionally, lazy loading will not be available for one-to-one
and many-to-one persistent attributes in types using field
access; they will be loaded eagerly instead.
```

Por lo que aunque indiquemos en la configuración que queremos modo "**LAZILY**", la entidad "**TipoPersona**" siempre se carga en modo "**EAGERLY**". Es más, dependiendo de la implementación que utilicemos de JPA una sola relación muchos a uno puede hacer que TODAS las entidades a las que se haga referencia desde otra entidad se carguen en modo "**EAGERLY**" tengan las relación que tengan (esto último es, en mi opinión, es un fallo que creo en varias implementaciones cuando el volumen de datos y tablas es grande).

OpenJPA proporciona la herramienta `OpenJPA enhancer` que añade código adicional a las entidades para poder utilizar "**LAZILY**" en todo tipo de relaciones. Se puede aplicar en tiempo de compilación (compilando las clases entidad con la herramienta **enhacer**) o en tiempo de ejecución (añadiendo a la línea de ejecución el parámetro -javaagent:/home/dev/openjpa/lib/org.apache.openjpa.jar=jdoEnhance=true,addDefaultConstructor=false).

Puede que nos interesen los dos mapeos ya que puede pasar que el mapeo con solo campos primitivos nos sirva mejor para una parte de la aplicación y el mapeo indicado las relaciones nos sirva mejor en otra parte de la aplicación. Lo que se puede hacer es crear dos entidades para la misma tabla. La desventaja de esta opción es que metemos otro mapeo de entidad en el contexto de JPA.

- Configuración de propiedades de la implementación: La lista de propiedades depende de la implementación de JPA que estemos utilizando pero hay dos propiedades que tienen, de una manera u otra, todas las implementaciones (y si no la tienen mejor cambia de implementación): Caché y Modo de carga.
    - Caché: JPA dispone de dos tipos de memoria caché llamadas **caché de nivel 1** y **caché de nivel 2**.
      La caché de nivel 1 está siempre disponible y es la zona de memoria que utiliza JPA para guardar los datos que vamos recuperando de base de datos y vamos modificándo. Cuando queremos persistir en base de datos (haciendo un **commit()**) JPA realizará las operaciones oportunas en base de datos para que los datos de este nivel de caché acaben en base de datos. La única forma de vaciar esta caché es con un **clear()** o si se consume el ciclo de vida del contexto de JPA.
      La caché de nivel 2 hay que habilitarla en la configuración y es la memoria que JPA pone a disposición de la aplicación para guardar objetos de uso frecuente en dicha aplicación y conseguir una mejora en el rendimiento. Estos objetos que se pueden cachear son Entidades (Datos) y Querys (Consultas). Cada implementación de JPA tiene una forma de configurar este tipo de caché pero lo que tienen mas o menos en común todas las implementaciones es indicar el tamaño de la caché, el tiempo de vida en caché de las entidades (de los datos almacenados) y si la caché esta en un entorno distribuido o en un sola maquina (que siempre va a ser una sola maquina). Un fallo muy común es

pensar que al activar esta caché es JPA la que memoriza las Entidades y las Querys mas usadas pero NO ES ASÍ, cada implementación de JPA tiene sus propios objetos y mecanismos para que tú (es decir, el código de la aplicación que has programado) guardes en esta caché lo que quieras.

- Modo de carga. El modo de carga de entidades puede ser "**LAZILY**" o "**EAGERLY**". Para explicar la diferencia supongamos que tenemos la siguiente entidad A

```
@Entity
public class A

	@Id
	private String id;

	@Column(name="B)
	private B b; // B es una entidad

	@Column(name="C)
	private C c; // C es una entidad

}
```

La diferencia entre los dos modos de carga consiste en que al traer una entidad "**A**" al contexto JPA (al hacer una consulta) el modo "**LAZILY**" solo carga los campos de esa entidad "**A**" y no los campos de las entidades "**B**" y "**C**" a los que se hace referencia desde "**A**". Mientras que el modo "**EAGERLY**" trae al contexto de JPA todo. Por lo general, parece mejor el modo "**LAZILY**" pero el modo "**EAGERLY**" es ideal cuando la aplicación que utiliza JPA es una aplicación WEB (siempre y cuando el tamaño de las tablas no sea muy grande con respecto a la memoria y velocidad de acceso del servidor donde este desplegada la aplicación WEB).

La configuración indica la forma en que se va rellenando el contexto de JPA. Sea cual sea la configuración que utilicemos, hay que tener claro que el contexto de JPA no hace mas que crecer y ocupar mas memoria por lo que a la hora de diseñar una aplicación tenemos que establecer de forma correcta los puntos en los que vaciar el contexto de JPA (operación "**clear()**" del "**EntityManager**"). Si no tienes en cuenta ésto desde el principio del diseño de la aplicación entonces lo que pasará es que acabarás haciendo un "**clear()**" en cualquier punto y te afectará al rendimiento.

Transacciones

La transacción es la unidad de trabajo de JPA. Cuando se cierra una transacción, JPA ve cual es el estado de las entidades y realiza las operaciones necesarias sobre base de datos para mantener la coherencia entre las entidades y la base de datos. Hay que tener claro que esto pasa si o si (a menos de nos preocupemos expresamente de que no pase como marcando la entidad como "**readOnly**" o haciendo un "**clear()**" en el **EntityManager**, pero esto no es el caso que nos ocupa ya que en la mayoría de los casos no será nuestra intención). Muchas veces que monitorizamos las operaciones que realiza JPA sobre base de datos al finalizar la transacción vemos operaciones ("**insert**", "**update**", "**detele**") que se ejecutan y no sabemos el porqué. El motivo es que hemos modificado una entidad y no nos hemos dado cuenta. Las entidades solo se utilizan para obtener datos y modificarlos en base de datos, nunca se pasan a un framework, a presentación ni nada por el estilo. Si modificamos una propiedad es a sabiendas de que se va a modificar.

Las dos formas (habrán más pero yo solo utilizaría una de estas dos) que se utilizan para abrir y cerrar una transacción son:

- Directamente tirando del "**EntityManager**". Debes utilizar el método "**begin()**" para abrir una transacción y "**commit()**" para cerrarla.

```
getEntityManager().getTransaction().begin();

Persona persona = new Persona();
persona.setNombre("Carlos");

getEntityManager().persist(persona);

getEntityManager().getTransaction().commit();
```

- Con anotaciones. Debes colocar la anotación "**@Transactional**" al principio de la función donde quieras que se abra y cierre la transacción.

```
@Transactional(propagation = Propagation.REQUIRED,
rollbackFor = Throwable.class)
public final void crearPersona(String nombrePersona){

      Persona persona = new Persona();
      persona.setNombre(nombrePersona);

      getEntityManager().persist(persona);


}
```

Las propagaciones más comunes son:

- **REQUIRE_NEW**. Crea la transacción y si ya existe la cierra y la abre de Nuevo.
- **REQUIRED**. Crea la transacción y si ya existe sigue con la misma.
- **NOT_SUPORTTED.** Sin transacción.

La anotación "**@Transactional**" no es nativa de JAVA (no esta en JPA y no está en la implementación de JPA que estés utilizando) si no que es de SPRING que será el encargado de gestionar el ciclo de vida de la transacción. Que esto no te preocupe porque es muy intuitivo pero obviamente tendrás que cargar JPA desde la configuración de SPRING.
La idea es muy simple **"SPRING se hace cargo de todo"** solo tienes que crear los BEANS que tiene que utilizar que son los siguientes:

- Crea un dataSource con **org.springframework.jdbc.datasource.DriverManagerDataSource**.

- Instancia el bean de gestion de entityManager **org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean** al que tienes que indicarle el dataSource y el persistence.xml

- Instancia los gestores de anotaciones **org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor** y **org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor**

- Crea un gestor de transacciones **org.springframework.orm.jpa.JpaTransactionManager** al que le pasas la instancia de gestión de entityManager.

- Instancia el gestor de transacciones con <tx:annotation-driven/>.

  Con ésto ya tienes configurado JPA por SPRING, ahora en cualquier clase de la aplicación que puedes utilizar la anotación "**@PersistenceUnit**" para inicializar un **entityManagerFactory**.

  OJO, el tipo de gestion de transacciones (se configura en el persistence.xml) tiene que ser JTA.

Es más cómodo utilizar anotaciones para abrir y cerrar la transacción actual y configurarla pero hay que tener SIEMPRE presente que es muy fácil equivocarse y cerrar la transacción antes de lo deseado sin darnos cuenta. Un ejemplo muy común de cerrar la transacción si darnos cuenta:

```
@Transactional(propagation = Propagation.REQUIRED, rollbackFor =
Throwable.class)
public final void crearPersona(String nombrePersona){

            Persona persona = new Persona();


      persona.setNombre(buscarNombreCompleto(nombrePersona));

            // ahora aquí la transacción esta cerrada
            getEntityManager().persist(persona);

}

@Transactional(propagation = Propagation.NOT_SUPORTED,
rollbackFor = Throwable.class)
public final String  buscarNombreCompleto (String
nombrePersona){

      ……
```

Al llamar a "**buscarNombreCompleto**" se ha cerrado la transacción por lo que el "**persist()**" dará un error de que se necesita una transacción abierta para realizar esta operación.

Una gran equivocación que he visto cuando se esta programando con JPA es pensar que se pueden crear transacciones anidadas.
El fuljo del programa es lineal. Cuando abres una transacción y si abres otra transacción antes de cerrar la anterior crees que tienes dos transacciones, pero no es así. Si abres una transacción antes de cerrar EXPLICITAMENTE la anterior, será el "**EntityManager**" el que tome la decisión (dependiendo de lo que hayas echo anteriormente) de si hace "**commit()**" y abre otra transacción, de si hace "**rollback()**" y abre otra transacción o de si te devuelve la misma transacción que estabas utilizando.
Esta explicación es un poco ambigua, veámoslo mejor con un ejemplo. Si tienes el código:

```
EntityTransaction t1 =
getEntityManager().getTransaction().begin();

Persona persona1 = new Persona();
persona.setNombre("Carlos");


 EntityTransaction t2 =
 getEntityManager().getTransaction().begin();

 Persona persona2 = new Persona();
 persona.setNombre("Maria");

 getEntityManager().persist(persona2);

 t2. commit();

 getEntityManager().persist(persona1);

 t1.commit();
```

Puedes creer que has creado una transacción dentro de otra, pero el objeto t1 y t2 es el mismo objeto por lo que te sobra el objeto t2 y habrías hecho dos veces “**commit()**” sin necesidad.

Modificar datos

Para modificar la información de base de datos hay que abrir una transacción, realizar las operaciones pertinentes para modificar la información y cerrar la transacción.

Solo se debe utilizar el “**persist()**” del “**EntityManager**” para guardar un registro por primera vez (para hacer un “**insert**”) y el “**remove()**” del “**EntityManager**” para borrar un registro (para hacer un “**delete**”). Para modificar un dato lo que se debe hacer es buscar los registros que queremos modificar, se modifica el objeto (entidad) donde se busco el dato y esperarse a que se cierre la transacción.

Existe un error muy común que devuelve en el log un error muy parecido al siguiente:

```
Encountered new object
"TipoPersona@1db72h1" in persistent field "Persona.tipoPersona" of
managed object " Persona@1236h63" during attach.  However, this field
does not allow cascade attach.  You cannot attach a reference to a new
object without cascading.
```

Esto pasa cuando intentas guardar una entidad en la que uno de los campos es otra entidad y el contexto de JPA no tiene los datos de esta última entidad que quieres guardar, veamos un ejemplo:

```
@Entity
public class Persona {

	@Id
	@GeneratedValue
	private int id;

	private String nombre;

	private Calle calle;

	public void setId(int id) {
		this.id = id;
	}

	public int getId() {
		return id;
	}
```

```
	public void setNombre(String nombre) {
		this.nombre = nombre;
	}

	public String getNombre() {
		return nombre;
	}

	public void setCalle(Calle calle) {
		this.calle = calle;
	}

	public Calle getCalle() {
		return calle;
	}
}

@Entity
public class Calle {

	@Id
	@GeneratedValue
	private int id;

	private String nombre;

	public void setId(int id) {
		this.id = id;
	}

	public int getId() {
		return id;
	}

	public void setNombre(String nombre) {
		this.nombre = nombre;
	}

	public String getNombre() {
		return nombre;
	}
}
```

Si queremos crear la persona:

```
....
	entityManager.begin();

	Persona persona = new Persona();
	persona.setNombre("Carlos");

	Calle calle = new Calle();
	calle.setId(1);
	persona.setCalle(calle);

	entityManager.persist(persona);
	entityManager.commit();


....
```

El problema lo tienes porque la entidad "**Calle**" (id=1) no lo tienes en el contexto de JPA y JPA no ha ido a buscarlo. Si esto te sucede tienes dos opciones:

- Hacer una búsqueda de la entidad "**Calle**" (id=1).

- Crear una referencia desde el "**EntityManager**" de la siguiente manera:

```
entityManager.begin();

Persona persona = new Persona();
persona.setNombre("Carlos");

Calle calle =(Calle)entityManager.getReference(Calle.class , 1);
persona.setCalle(calle);

entityManager.persist(persona);
entityManager.commit();
```

Si tienes muy claro que la entidad "**Calle**" (id=1) la vas a necesitar es mejor buscarla (Primera opción) pero si no esta claro que se vaya a necesitar es mejor crear una referencia (Te ahorras una "**select**").

El operación "**getReference()**" devuelve la referencia valida a una entidad. Si la entidad esta en el contexto de JPA (ya la has utilizado antes) la operación "**getReference()**" te devolverá la entidad y si no esta en el contexto de JPA entonces te devolverá una referencia al objeto. Ten cuidado porque si devuelve una referencia y modificas algún campo de la referencia este campo no se guarda en base de datos pero si devuelve una entidad y la modificas entonces SI se guardará en base de datos (siempre y cuando hayas abierto una transacción).

En mi opinión NUNCA se debe utilizar la propagación en cascada por dos aspectos:

- Por base de datos: Nunca hay que dejar que las relaciones entre tablas sean las que decidan si se borra un registro. Si quieres borrar un registro tiene que ser con una sentencia (con un "**delete**").
- Por JPA: Si tu aplicación tiene cierto volumen, seguramente se realizaran comandos "**update**" contra la base de datos que no hagan falta. Esto es porque en algún momento JPA cree que has modificado una entidad antes de cerrar una transacción. Esto ya es bastante perdida de rendimiento y de control para que encima JPA realice comando "**delete**" y "**update**" sin tu control por definir la propagación en cascada y se realicen mas operaciones innecesarias.

Conclusión

A la hora de utilizar JPA (o cualquier otro ORM) la tendencia natural que tenemos todos los programadores es querer utilizarlo de cualquier manera o como si fuera un DAO. Esto es el gran error que creo que tenemos que cambiar por lo que espero que los conceptos que he tratado en este articulo sirvan para que antes de diseñar las clases de tu aplicación, antes de que establezcas capas en tu aplicación, antes de ponerte a “picar código”, etc… primero tengas en cuenta todo lo que te he contado y que te leas la documentación de la implementación de JPA que vayas a utilizar QUE ES GRATIS.