JPA

Comienzo de la práctica	25-10-2016
Entrega de la práctica	15-11-2016

1 Introducción a JPA

Java Persistence API (JPA) es una especificación de interfaz de programación que hace posible la gestión de datos relacionales dentro de las aplicaciones para Java, que utilizan las plataformas de *Standard Edition* y *Enterprise Edition*.

JPA permite una correspondencia entre objetos y bases de datos relacionales que hace más fácil convertir los objetos de la lógica de negocio de una aplicación a una base de datos relacional. A esto también se le conoce como el "O/R mapping" ("object / relational mapping") u ORM en la literatura.

El API de Persistencia de Java se originó como parte del trabajo de grupo experto JSR 220 del denominado *Java Community Process*. JPA 2.0 ha sido el resultado del trabajo del grupo experto JSR 317. La persistencia que nos proporciona JPA cubre tres importantes ámbitos:

- el propio API definido en el paquete `javax.persistence` del lenguaje Java
- el lenguaje de consulta *Java Persistence Query Language* (JPQL)
- los metadatos de objetos y relacionales

Lo que la hace útil para nosotros es que resulta más fácil programar con JPA que con JDBC debido a una serie de características propias que lo hacen diferente de un sistema de gestión de bases de datos al uso:

- JPA puede crear automáticamente tablas de una base de datos, directamente a partir de las relaciones entre los *objetos de negocio*.
- JPA automáticamente hacer la correspondencia entre los objetos del diseño de la aplicación y las filas de una tabla de una base de datos relacional.
- JPA puede realizar automáticamente uniones ("joins") para satisfacer las relaciones entre los objetos anteriormente mencionados.
- JPA se ejecuta encima de JDBC y, por tanto, es compatible con cualquier base de datos que posea un driver JDBC.
- El uso de JPA evita al programador de aplicaciones Java tener que escribir código JDBC y SQL.

1.1 Implementaciones actuales de JPA

Existen varias implementaciones de este estándar y todas ellas siguen la especificación de JPA:

- EclipseLink
- Hibernate
- TopLink.

La implementación que se considera actualmente como de referencia para JPA es *EclipseLink*

Un servidor completo de Java EE normalmente establecerá una implementación de JPA como propia; por ejemplo: *Glassfish* utiliza *TopLink* y *WildFly* utiliza *Hibernate*.

Cuando se utiliza Tomcat, podemos elegir la implementación de JPA que más nos acomode. En esta práctica vamos a utilizar exclusivamente la implementación EclipseLink.

1.2 Entidades y sus gestores

Cuando trabajamos con JPA los objetos de negocio se les da el nombre de *entidades* y son gestionadas por un *gestor de entidades*. Un servidor completo que contemple Java EE suele proporcionar un gestor de entidades ya incorporado, que incluirá, entre otras características avanzadas, el poder deshacer ("rollback") las transacciones de forma automática.

En esta práctica vamos a enseñar cómo utilizar los gestores de entidades de forma independiente y cómo hacer uso de ellos desde nuestro IDE, alternativamente a lo que haría un servidor de Java EE completo, de tal forma que podamos en el futuro adoptar esta forma de trabajar para desarrollar en nuestro computador cualquier aplicación que necesite persistencia, sólo haciendo uso de Tomcat o un contenedor de aplicaciones Web similar.

Para convertir una clase programada con un lenguaje de POO normal, que refleja la lógica de negocio de una aplicación, en una *entidad* tenemos que escribir anotaciones en el código de esta clase. Dichas anotaciones sirven para indicar cómo se debe ser guardada la clase mencionada en una base de datos y también cómo podemos relacionar las clases entre sí. El único problema que tiene utilizar JPA independientemente de un servidor de aplicaciones Java EE completo consiste en que tendríamos que crearnos nuestro propio gestor de entidades.

2 Las anotaciones que propone JPA para las *clases de negocio*

Una clase que quiera hacerse persistente en una base de datos debería ser anotada con `javax.persistence.entity` y entonces esta clase pasaría a denominarse *entidad*.

JPA creará una tabla para cada entidad realizada de la forma anterior en la base de datos y las distintas instancias de esta *entidad*serían ahora las filas de esta tabla.

Todas las tablas que representan a entidades deben definir una *clave primaria* y las clases han de poseer un constructor sin argumentos, que no se podrá nunca declarar como `final`. Las claves aludidas pueden ser un campo simple una combinación de campos de la clase que hemos convertido en una *entidad*.

JPA permite generar una clave primaria en la base de datos mediante la anotación `@GeneratedValue`. Por defecto, el nombre de la tabla se corresponderá con el nombre de la clase, aunque podríamos cambiarlo si utilizamos además la anotación: `@Table(name="NEWTABLENAME")`.

Los campos pertenecientes a una *entidad* se han de salvar en la base de datos. JPA puede utilizar las propias variables de instancia de una clase o los correspondientes métodos "getter" y "setter" para acceder a dichos campos, pero nunca podremos mezclar ambos estilos cuando programemos una clase–entidad.

Por defecto, JPA convierte en persistentes a todos los campos de una *entidad*, pero si no nos interesa que esto ocurra, es decir, queremos que unos determinados campos no sean salvados, entonces los marcaremos con la anotación `@Transient`.

Cada uno de los campos de una clase-entidad se hace corresponder con una columna que posee el mismo nombre del campo, pero si nos interesa, también podríamos cambiarle nombre por defecto, utilizando la anotación: `@Column(name="newColumnName")`.

El resto de anotaciones relacionadas con la persistencia de campos y de métodos getter/setter que se pueden utilizar cuando queremos transformar una clase en una *entidad* vienen descritos en la siguiente tabla:

`@Id`	Identifica el único ID de la entrada en la base de datos
`@GeneratedValue`	Junto con ID sirve para definir que este valor se genera automáticamente
`@Transient`	Este campo no será salvado en la base de datos

JPA permite definir relaciones entre clases, p.e.: se podría definir que una clase fuera parte de otra. Las clases entre sí pueden establecer relaciones del tipo: uno–a–uno, muchos–a–uno y muchos–a–muchos. Una relación puede ser *bidireccional* o *unidireccional*, p.e.: en una relación bidireccional ambas clases guardan una referencia hacia la otra, mientras que en una relación unidireccional, sólo una clase mantendrá una referencia hacia la otra. En una relación *bidireccional* necesitamos especificar el lado propietario de dicha relación en la otra clase con el atributo `mappedBy` (`@ManyToMany(mappedBy="atributoDeLaClaseAdquirida")`). Las anotaciones relacionadas con las relaciones entre clases vienen dadas por la siguiente lista:

- `@OneToOne`
- `@OneToMany`
- `@ManyToOne`
- `@ManyToMany`

2.0.1 Configuración de una unidad persistente

El gestor de entidades `javax.persistence.EntityManager` proporciona las operaciones para la base de datos: encontrar los objetos, hacerlos persistentes, eliminarlos, etc. Las entidades gestionadas propagarán automáticamente los cambios a la base de datos con la orden `commit`. Si cerramos el gestor de entidades con la orden `close()`, entonces las entidades gestionadas se quedarán en un estado independiente (o *"detached"*), pero las podemos volver a sincronizar con la base de datos utilizando el método `merge()` del gestor de entidades.

El gestor de entidades (`EntityManager`) se crea con una clase *factoría*: `EntityManagerFactory`, que viene incluído en el paquete de persistencia de `javax`.

Un conjunto de entidades que estén conectadas lógicamente se pueden agrupar dentro de una unidad persistente, que definirá los datos de conexión necesarios con la base de datos, tales como: el *driver*, el *usuario* y la *palabra de paso*.

Para convertir a una unidad de nuestra aplicación en persistente tendremos que programar el archivo "persistence.xml" dentro del subdirectorio `META-INF` del directorio `/src` de nuestro proyecto en el IDE que estemos utilizando, tal como se muestra en el siguiente código:

```
<?xml version="1.0" encoding="UTF-8" ?>
<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_2_0.xsd"
  version="2.0" xmlns="http://java.sun.com/xml/ns/persistence">
  <persistence-unit name="relaciones_persistentes"
   transaction-type="RESOURCE_LOCAL">
  <class>jpa.eclipselink.modelo.Persona</class>
  <class>jpa.eclipselink.modelo.Familia</class>
  <class>jpa.eclipselink.modelo.Empleo</class>
    <properties>
      <property name="javax.persistence.jdbc.driver"
       value="org.apache.derby.jdbc.EmbeddedDriver" />
      <property name="javax.persistence.jdbc.url"
  value="jdbc:derby:D:\software\db-derby-10.12.1.1-bin\basesDatos\relacionesBD;
   create=true"/>
      <property name="javax.persistence.jdbc.user" value="prueba" />
      <property name="javax.persistence.jdbc.password" value="prueba" />

      <!--EclipseLink debe crear este esquema de base de datos automaticamente-->
    <property name="eclipselink.ddl-generation" value="drop-and-create-tables" />
      <property name="eclipselink.ddl-generation.output-mode"
        value="both" />
    </properties>
  </persistence-unit>
</persistence>
```

Los primeros 4 elementos de `<properties>` configuran los diferentes parámetros de conexión de JDBC.

El *elemento-property* con el nombre `eclipselink.ddl-generation` indica lo que hará JPA cuando encuentre tablas perdidas o que no existen. Después de la primera ejecución del ejemplo, habrá que eliminar la propiedad anterior del archivo `persistence.xml`, ya que en caso contrario recibiríamos un error, porque `EclipseLink` volverá a intentar crear dicho esquema en la base de datos.

3 Ejemplo tutorial

Lo primero de todo es descargar la distribución en "EclipseLink Installer.zip" del sitio (ver: `http://www.eclipse.org/eclipselink/downloads/`), que contiene los JARs de la implementación que vamos a necesitar, en concreto hay que descomprimir el `.zip` y buscar en la distribución los siguientes archivos:

- `eclipselink.jar`
- `javax.persistence_*.jar`

Los archivos anteriores hay que incluirlos en el `buildpath` del proyecto que tengamos creado en nuestro IDE para que funcione la demostración. En el ejemplo–demostración que sigue se va a utilizar la base de datos Apache Derby (ver `http://db.apache.org/derby/`). Para lo cual vamos a descargar la última distribución Derby del sitio de descargas (ver `http://db.apache.org/derby/derby_downloads.html`) y necesitaremos incluir en el `buildpath` del proyecto que hemos creado en nuestro IDE el archivo `derby.jar`.

3.1 Desarrollo del proyecto *jpa.simple* paso a paso

En nuestro IDE crearemos un nuevo proyecto java, que nombraremos "`jpa.simple`".

- a continuación, crearemos un subdirectorio que llamaremos "`lib`" donde ubicaremos los JARs que hemos mencionado anteriormente (3)
- añadiremos los contenidos del citado subdirectorio "`lib`" al `buildpath` de nuestro proyecto
- ahora nos creamos el paquete de Java: "`jpa.simple.modelo`" donde ubicaremos la clase "`Completo.java`", que también nos crearemos

```
1  package jpa.simple.modelo;
2  import javax.persistence.Entity;
3  import javax.persistence.GeneratedValue;
4  import javax.persistence.GenerationType;
5  import javax.persistence.Id;
6  @Entity
7  public class Completo {
8    @Id
9    @GeneratedValue(strategy = GenerationType.IDENTITY)
10   private Long id;
11   private String resumen;
12   private String descripcion;
13   public String getResumen() {
14     return resumen;
15   }
16   public void setResumen(String resumen) {
17     this.resumen = resumen;
18   }
19   public String getDescripcion() {
20     return descripcion;
21   }
22   public void setDescripcion(String descripcion) {
23     this.descripcion = descripcion;
24   }
25   @Override
26   public String toString() {
27     return "Completo [resumen=" + resumen + ", descripcion=" + descripcion
28         + "]";
29   }
30 }
```

Ahora nos crearemos el subdirectorio "META-INF" en la carpeta "src" del proyecto que hemos creado en nuestro IDE e incluiremos allí el archivo persistence.xml que hemos descrito anteriormente (2.0.1).

A través del parámetro eclipselink.ddl-generation, incluido en persistence.xml, estamos utilizando los conmutadores de EclipseLink para indicar que el esquema de la base de datos será descartado y creado automáticamente en cada ejecución.

Podemos inspeccionar el código SQL que se genera y que se aplicará a la base de datos creada automáticamente por el marco de trabajo y que sirve de soporte a la aplicación si accedemos al directorio local (por ejemplo, D : \software\db − derby − 10.12.1.1 − bin\basesDatos\miBD) donde hayamos configurado Derby.

Es necesario cambiar el valor del parámetro eclipselink.ddl-generation.output-mode de "database" a "sql-script" o al valor "both". Si lo hacemos de esta manera, podemos comprobar que se crearán los 2 archivos siguientes: "createDDL.jdbc" y "dropDDL.jdbc".

Por último, nos crearemos la clase que llamaremos Principal.java. Cada vez que se ejecute su método main(...), se va a crear una nueva entrada en la base de datos de soporte.

Como ya se indicó antes (2.0.1), hay que eliminar la propiedad eclipselink.ddl-generation después de la primera ejecución de nuestro programa y para las ejecuciones posteriores.

```
1  package jpa.simple.main;
2
3  import java.util.List;
4  import javax.persistence.EntityManager;
5  import javax.persistence.EntityManagerFactory;
6  import javax.persistence.Persistence;
7  import javax.persistence.Query;
8  import jpa.simple.modelo.Completo;
9
10 public class Principal {
11   private static final String PERSISTENCE_UNIT_NAME = "tutorialJPA";
12   private static EntityManagerFactory factoria;
13
14   public static void main(String[] args) {
15     factoria = Persistence.createEntityManagerFactory(PERSISTENCE_UNIT_NAME);
16     EntityManager em = factoria.createEntityManager();
17     // leer las entradas existentes y escribir en la consola
18     Query q = em.createQuery("select␣t␣from␣Completo␣t");
19     //Crearse una lista con template: "Completo" a la que asignameros el resultado de la consulta
20     //en la base de datos ("q.getResultList()"
21     //Iterar en la lista e imprimir las instancias "completo"
22     //Ahora imprimimos el numero de registros que tiene ya la base de datos
23     System.out.println("Tamano:␣" + listaCompleto.size());
24
25     //Ahora vamos a trabajar con una transaccion en la base de datos
26     em.getTransaction().begin();
27     //Crearse una instancia de completo y utilizar los metodos "setResumen()" y "setDescripcion()"
28     //Posteriormente hay que decir al gestor de entidad (em) que la instancia va a ser persistente;
29     //conseguir la transaccion ("em.getTransaction()") y hacerla definitiva ("commit()")
30
31     //Por ultimo, hay que cerrar al gestor de entidad
32     em.close();
33   }
34 }
```

4 Creación de relaciones entre entidades persistentes

El objetivo que pretendemos alcanzar siguiendo esta sección consiste en aprender a utilizar relaciones entre entidades persistentes que se han programado conforme a la interfaz JPA de programación de aplicaciones con datos relacionales para Java.

Los pasos que hay que seguir para programar correctamente aplicaciones que incluyen las mencionadas relaciones se describen en las subsecciones que aparecen a continuación.

4.1 Creación del proyecto en nuestro IDE

Nos creamos un nuevo proyecto Java y, dentro de él, un subdirectorio que llamaremos "`lib`" donde ubicaremos los archivos "`.jar`" que describimos anteriormente (3) y que hacen accesible la interfaz de programación JPA a nuestro programa.

Posteriormente, nos crearemos un paquete Java que denominaremos "`modelo`", que he de incluir las siguientes clases:

- `Familia`: contiene métodos para obtener la identificación, descripción de una familia y un método que permita listar a todos los miembros de una familia (que son instancias de la clase "`Persona`"); además tendremos que establecer una relación uno-a-muchos con "`Persona`".

- `Persona`: ha de contener métodos para obtener el nombre y los apellidos de una persona, así como también otros para ID; una lista privada de los empleos que ha tenido cada persona y las siguientes relaciones: muchos-a-uno con "`familia`" a través del método `getFamilia()` y uno-a-muchos con "`lista de empleos`" a través del método `getListaEmpleos()`.

- `Empleo`: ha de incluir como atributos privados: ID (del empleo), salario y descripción del empleo y los métodos `setter-getter` asociados.

4.2 Creación del archivo `persistence.xml`

Dentro del subdirectorio "`src/META-INF`" del proyecto que anteriormente creamos en nuestro IDE (4.1), hemos de incluir ahora un archivo "`persistence.xml`". No hemos de olvidar cambiar el camino y el nombre a la base de datos de destino para evitar sobreescribir los datos salvados de otras aplicaciones, así como de indicar las nuevas *clases–entidad* persistentes de nuestro programa, que podríamos denominar como sigue:

```
<class>jpa.eclipselink.modelo.Persona</class>
<class>jpa.eclipselink.modelo.Familia</class>
<class>jpa.eclipselink.modelo.Empleo</class>
```

4.3 Creación de una prueba con JUnit del programa

Para probar la aplicación vamos a crearnos una clase Test, utilizando para ello el paquete de *pruebas unitarias* `JUnit` para Java.

El método `setUp()` de la clase `JpaTest` que vamos a programar va a crear primeramente unas cuantas entradas, como muestra el siguiente trozo de código. Posteriormente, se declararán como entidades persistentes de la base de datos las personas incluídas en las familias.

```
1  package jpa.eclipselink.principal;
2  import static org.junit.Assert.assertTrue;
3  import org.junit.Before;
4  import org.junit.Test;
5
6  import javax.persistence.EntityManager;
7  import javax.persistence.EntityManagerFactory;
8  import javax.persistence.Persistence;
9  import javax.persistence.Query;
10
11 import jpa.eclipselink.modelo.Familia;
12 import jpa.eclipselink.modelo.Persona;
13
14
15 public class JpaTest {
16         private static final String PERSISTENCE_UNIT_NAME = "relaciones_persistentes";
17         private EntityManagerFactory factoria;
18           @Before
19           public void setUp() throws Exception {
20             factoria = Persistence.createEntityManagerFactory(PERSISTENCE_UNIT_NAME);
21             EntityManager em = factoria.createEntityManager();
22         // Comenzar una nueva transaccion local de tal forma que pueda persistir
23         //como una nueva entidad.
24         // Leer las entradas que ya hay en la base de datos.
25         // Las personas no deben tener ningun atributo asignado todavia
26                 Query q = em.createQuery("select m from Persona m");
27
28
29         // Comprobar si necesitamos crear entradas en la base
30             boolean crearseNuevasEntradas = (q.getResultList().size() == 0);
31            if (crearseNuevasEntradas){
32                   // Vamos a ello...
33               assertTrue(q.getResultList().size() == 0);
34               Familia familia = new Familia();
35               familia.setDescripcion("Familia Martinez");
36               em.persist(familia);
37               for (int i = 0; i < 20; i++) {
38             //Crearse una "persona" y asignar sus atributos con: setNombre(...) y setApellidos(...)
39
40
41                 em.persist(persona);
42             //anadir "persona" al ListArray<Persona> que representa a "familia"
43
44
45                 // ahora hacemos que persista la relacion familia-persona
46                 em.persist(persona);
47                 em.persist(familia);
48               }
49            }
50              // Ahora hay que hacer "commit" de la transaccion, lo que causa que la
51              //entidad se salve en la base de datos.
52            em.getTransaction().commit();
53             //Ahora hay que cerrar el EntityManager o perderemos nuestras entradas.
54            em.close();
55     }//setUp()
```

```
@Test
  public void comprobarPersonas() {
    //ahora vamos a comprobar la base de datos para ver si las entradas que hemos creado estan alli
    //Para eso, nos crearemos un gestor de entidades "fresco"
    //Realizaremos una consulta simple que consistira en seleccionar a todas las personas
    //Si todo ha ido bien, en la lista de personas hemos de tener a 20 miembros:
    //utilizar "assertTrue()", "getResultList()" y "size()" para comprobarlo.
   //Acordarse de cerrar el gestor de entidades
  }
```

```
@Test
  public void comprobarFamilia() {
    //Para esto, nos crearemos un gestor de entidades "fresco"
    //Recorrer cada una de las entidades y mostrar cada uno de sus campos asi como la fecha de creacion
    //Deberiamos tener una familia con 20 personas
    //Utilizar "assertTrue()", "getResultList()", "size()" y "getSingleResult()" para determinarlo

//Acordarse de cerrar el gestor de entidades
  }
```

```
@Test(expected = javax.persistence.NoResultException.class)
  public void eliminarPersona() {
    //Para esto, nos crearemos un gestor de entidades "fresco"
    // Begin a new local transaction so that we can persist a new entity
    //Comenzar una nueva transaccion local de tal manera que podamos hacer persitente una nueva
    //entidad. Ahora me creare la consulta necesaria eliminar la persona de nombre y apellidos
    //que indicare despues.

Query q = em.createQuery("SELECT p FROM Persona p WHERE p.nombre=:nombre AND p.apellido=:apellido");

   //Ahora asigno los parametros
    q.setParameter("nombre", "Pepe_1");
    q.setParameter("apellido", "Martinez_!");

//Ahora utilizo el metodo: "getSingleResult()" para obtener a la persona que me interesa y
//los metodos: "remove(persona)" y "commit()" para eliminarla de la entidad y confirmar la
//eliminacion, respectivamente.
//Acordarse de cerrar el gestor de entidades
  }
```

4.4 Ejecutar como una prueba

La comprobación de que funciona se hace mediante una prueba del marco de trabajo para pruebas unitarias de Java: JUnit (ver “créditos” para tutorial). El método `setup()` creará unas cuentas entradas del test. Después de que se crean stas entradas, se leerán y se cambiará un campo de las entradas que después se salva en la base de datos.

4.5 Convertir en componente

Ahora podemos crearnos un componente utilizando para ello un proyecto Maven dentro de nuestro IDE y desplegarlo después para que pueda utilizar las entidades persistentes que hemos programado anteriormente.

4.6 El paquete `jpa.eclipselink.modelo`

Se ha supuesto un modelo conformado por: `Familia`, `Persona`, `Empleo`, tal como sigue

```
package jpa.eclipselink.modelo;
import java.util.ArrayList;
import java.util.List;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToMany;
@Entity
public class Familia {
        @Id
          @GeneratedValue(strategy = GenerationType.TABLE)
          private int id; private String descripcion;
          @OneToMany(mappedBy = "familia")
          private  final List<Persona> miembros = new ArrayList<Persona>();
          public int getId() {
            return id;
          }
          public void setId(int id) {
            this.id = id;
          }
          public String getDescripcion() {
            return descripcion;
          }
          public void setDescripcion(String descripcion) {
            this.descripcion = descripcion;
          }
          public List<Persona> getMiembros() {
            return miembros;
          }
      public void setMiembros(Persona persona){
          miembros.add(persona);
      }
}
```

```
package jpa.eclipselink.modelo;
import java.util.ArrayList;
import java.util.List;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import javax.persistence.ManyToOne;
import javax.persistence.Transient;
@Entity
public class Persona {
         @Id
          @GeneratedValue(strategy = GenerationType.TABLE)
          private String id;
          private String nombre;
          private String apellidos;
          private Familia familia;
          private List<Empleo> listaEmpleos = new ArrayList<Empleo>();
...
        @ManyToOne
          public Familia getFamilia() {
            return familia;
          }
...
  @OneToMany
          public List<Empleo> listaEmpleos() {
            return this.listaEmpleos;
          }
...
}
```

```java
package jpa.eclipselink.modelo;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Empleo {
         @Id
          @GeneratedValue(strategy = GenerationType.TABLE)
          private int id;
          private double salario;
          private String descripcionTrabajo;

         ...
}
```

Créditos

- Package javax.persistence:http://docs.oracle.com/javaee/7/api/javax/persistence/package-summary.html
- http://www.vogella.com/tutorials/JavaPersistenceAPI/article.html
- http://docs.oracle.com/javaee/5/tutorial/doc/bnbpy.html
- http://robertleggett.wordpress.com/2014/01/29/jersey-2-5-1-restful-webservice-jax-rs-with-jpa-2-1-and-derby-in-memory-database/
- https://en.wikipedia.org/wiki/Java_Persistence_API
- J. Murach, M. Urban. “Java Servlets and JSP”. Murach, 2014
- http://www.thejavageek.com/jpa-tutorials/
- Tutorial sobre pruebas unitarias con JUnit:http://www.vogella.com/tutorials/JUnit/article.html