JPA

Comienzo de la práctica	22-10-2020
Fecha de entrega aconsejada	5-11-2020

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 desarrollo *Standard Edition* (JSE) y *Enterprise Edition* (JEE). JPA permite una correspondencia entre objetos y bases de datos relacionales que hace más fácil convertir los objetos dependientes de la lógica de negocio de una aplicación a entidades de 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* y, finalmente, la especificación denominada 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` incluido en las *distro* actuales del lenguaje Java (`https://docs.oracle.com/javaee/7/api/javax/persistence/package-summary.html#package.description`)
- 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* que pertenecen al nivel de proceso de negocio.
- JPA puede realizar la correspondencia entre los mencionados objetos, en el nivel de diseño de una aplicación, y las filas de una tabla de una base de datos relacional automáticamente.
- 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 en Java tener que escribir código JDBC y SQL.

JPA se lanzó por primera vez como un subconjunto de la especificación EJB 3.0 (JSR 220) en Java EE 5. Desde entonces ha evolucionado como su propia especificación, comenzando con el lanzamiento de JPA 2.0 en Java EE 6 (JSR 317). En la actualidad JPA 2.2 se ha adoptado para la continuación de la API de persistencia de Java como parte de Jakarta EE.

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*, ya que este marco es compatible con JPA 2.2 con NoSQL

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 sea conforme con 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 de un lenguaje de base de datos y cómo hacer uso de ellos programando sólo en Java desde nuestro IDE. Este enfoque es puede considerar alternativo a lo que haría un servidor de Java EE completo, de tal forma que en el futuro podremos adoptar esta forma de trabajar para desarrollar cualquier aplicación que necesite persistencia. Sólo necesitaremos utilizar Tomcat o un contenedor de aplicaciones Web similar a este para poder llevarlo a cabo.

Para convertir una clase programada con un lenguaje de POO normal en una *entidad*, según el sentido indicado anteriormente, sólo tenemos que escribir anotaciones en el código de esta clase. Dichas anotaciones sirven para indicar cómo se debe guardar la clase mencionada en una base de datos y también cómo podemos relacionar las distintas clases entre sí. El único problema que hemos de resolver para poder utilizar JPA independientemente de un servidor de aplicaciones Java EE completo consiste en que tendremos que crearnos nuestro propio gestor de entidades con una *factoría* de la clase EntityManagerFactory, incluida en el paquete de persistencia de *javax*.

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*. A continuación, JPA creará una tabla para cada entidad, obtenida de la forma anterior, en una base de datos. Las distintas instancias de dicha *entidad* serían ahora las filas de esta tabla en la base de datos de soporte.

Todas las tablas que representan a las mencionadas 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 primarias aludidas pueden ser un campo simple, o 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 queremos, utilizando la anotación: `@Table(name="NEWTABLENAME")`. Dentro de la anotación `@GeneratedValue` se pueden indicar varias estrategias:

- `IDENTITY`: con este tipo de estrategia el proveedor de persistencia debe asignar claves primarias para la entidad usando una columna de identidad de base de datos. Las instancias sucesivas de la entidad son filas que se añaden a la tabla de la base de datos y se autoincrementa el `Id`. La declaración siguiente del ejemplo tutorial `jpa`

```
@Entity
public class Todo {
@Id
@GeneratedValue(strategy= GenerationType.IDENTITY)
private Long id;
private String resumen;
private String descripcion;
```

  Produce la siguiente salida:

```
CREATE TABLE TODO (ID BIGINT GENERATED BY DEFAULT AS IDENTITY NOT NULL,
DESCRIPCION VARCHAR(255), RESUMEN VARCHAR(255), PRIMARY KEY (ID))
```

- `TABLE`: con este tipo de estrategia el proveedor de persistencia subyacente debe usar una tabla de la base de datos para generar/mantener una clave primaria única para las entidades. La declaración siguiente del ejemplo tutorial `jpa.familias`:

```
@Id
@GeneratedValue(strategy = GenerationType.TABLE)
...
```

  Produce la siguiente salida:

```
CREATE TABLE FAMILIA (ID INTEGER NOT NULL, DESCRIPCION VARCHAR(255), PRIMARY KEY (ID))
CREATE TABLE PERSONA (ID INTEGER NOT NULL, APELLIDO VARCHAR(255), NOMBRE VARCHAR(255),
NONSENSEFIELD VARCHAR(255), FAMILIA_ID INTEGER, PRIMARY KEY (ID))
CREATE TABLE EMPLEO (ID INTEGER NOT NULL, DESCREMPLEO VARCHAR(255), SALARIO FLOAT,
PRIMARY KEY (ID))
...
```

- Existen otras estrategias, que también se usan, tales como:

  - `GenerationType.SEQUENCE`, con esta estrategia el proveedor de persistencia subyacente debe usar una secuencia de base de datos para obtener la siguiente clave primaria única para las entidades.

- `GenerationType.AUTO`, esta estrategia indica que el proveedor de persistencia debe elegir automáticamente una estrategia adecuada para la base de datos en particular. Este es el `GenerationType` predeterminado.

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")`.

En resumen, las 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")`). Por ejemplo, en la clase `Familia` se incluye el atributo `mappedby` para indicar que se ha declarado el atributo `familia` en la clase *adquirida* `Persona`:

```
1 public class Familia {
2         ...
3     @OneToMany(mappedBy = "familia")
4     private final List<Persona> miembros = new ArrayList<Persona>();
5     ...
```

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="gente" transaction-type="RESOURCE_LOCAL">
         <class>jpa.familias.modelo.Familia</class>
         <class>jpa.familias.modelo.Persona</class>
         <class>jpa.familias.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:C:\Users\software\db-derby-10.14.2.0-bin
    \basesDatos\relacionesDB;create=true" />
  <property name="javax.persistence.jdbc.user" value="test" />
  <property name="javax.persistence.jdbc.password" value="test" />

  <!-- EclipseLink should create the database schema automatically -->
  <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. Aunque también podríamos asignarle el valor "`drop-and-create-tables`" a esta propiedad, pero esto ocasionaría que se destruyera el esquema de la base datos en cada nueva ejecución del programa.

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 }
```

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  import java.util.List;
3  import javax.persistence.EntityManager;
4  import javax.persistence.EntityManagerFactory;
5  import javax.persistence.Persistence;
6  import javax.persistence.Query;
7  import jpa.simple.modelo.Completo;
8  import jpa.simple.modelo.Todo;
9  public class Principal {
10         private static final String PERSISTENCE_UNIT_NAME = "tutorialJPA";
11         private static EntityManagerFactory factoria;
12
13         public static void main(String[] args) {
14                 // TODO Auto-generated method stub
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                 List<Completo> listaCompleto = q.getResultList();
24         for (Completo completo : listaCompleto) {
25             System.out.println(completo);
26         }
27
28                 System.out.println("Tamano:␣" + listaCompleto.size());
29
30                 //Ahora vamos a trabajar con una transaccion en la base de datos
31                 em.getTransaction().begin();
32                 //Crearse una instancia de completo y utilizar los metodos "setResumen()" y "setDescripcion()"
33                 Completo completo = new Completo();
34         completo.setResumen("Esto␣es␣una␣prueba");
35         completo.setDescripcion("Esto␣es␣una␣prueba");
36                 //Posteriormente hay que decir al gestor de entidad (em) que la instancia va a ser persistente;
37                 //conseguir la transaccion ("em.getTransaction()") y hacerla definitiva ("commit()")
38         em.persist(completo);
39         em.getTransaction().commit();
40       //Por ultimo, hay que cerrar al gestor de entidad
41                 em.close();
42         }
43
44 }
```

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 El paquete `jpa.eclipselink.modelo`

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

```
1 package jpa.familias.modelo;
2 import java.util.ArrayList;
3 import java.util.List;
4
5 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;
    }
}
```

```
package jpa.familias.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.ManyToOne;
import javax.persistence.OneToMany;
import javax.persistence.Transient;
@Entity
public class Persona {
        @Id
    @GeneratedValue(strategy = GenerationType.TABLE)
    private int id;
    private String nombre;
    private String apellido;

    private Familia familia;

    private String nonsenseField = "";

    private List<Empleo> listaEmpleos = new ArrayList<Empleo>();
    public int getId() {
        return id;
    }
    public void setId(int Id) {
        this.id = Id;
    }
    public String getNombre() {
        return nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    // Dejar el nombre standard de la columna de la tabal
```

```
38      public String getApellido() {
39          return apellido;
40      }
41      public void setApellido(String apellido) {
42          this.apellido = apellido;
43      }
44      @ManyToOne
45      public Familia getFamilia() {
46          return familia;
47      }
48      public void setFamilia(Familia familia) {
49          this.familia = familia;
50      }
51      @Transient
52      public String getNonsenseField() {
53          return nonsenseField;
54      }
55      public void setNonsenseField(String nonsenseField) {
56          this.nonsenseField = nonsenseField;
57      }
58      @OneToMany
59      public List<Empleo> getListaEmpleos() {
60          return this.listaEmpleos;
61      }
62      public void setListaEmpleos(List<Empleo> nickName) {
63          this.listaEmpleos = nickName;
64      }
65  }
```

```
1   package jpa.familias.modelo;
2   import javax.persistence.Entity;
3   import javax.persistence.GeneratedValue;
4   import javax.persistence.GenerationType;
5   import javax.persistence.Id;
6
7   @Entity
8   public class Empleo {
9            @Id
10              @GeneratedValue(strategy = GenerationType.TABLE)
11              private int id;
12              private double salario;
13              private String descrEmpleo;
14
15              public int getId() {
16                  return id;
17              }
18              public void setId(int id) {
19                  this.id = id;
20              }
21              public double getSalario() {
22                  return salario;
23              }
24              public void setSalario(double salario) {
25                  this.salario = salario;
26              }
27              public String getDescrEmpleo() {
28                  return descrEmpleo;
29              }
30              public void setDescrEmpleo(String descrEmpleo) {
31                  this.descrEmpleo = descrEmpleo;
32              }
33  }
```

4.4 Creación de una prueba con JUnit del programa

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

Para hacerlo, nos crearemos un subdirectorio `test` en el proyecto `jpa.familias` (seleccionar con el botón derecho el nombre del proyecto en el navegador de Eclipse, *BuildPath*, *ConfigureBuildPath*, seleccionar la pestaña `Source`, `Add Folder`, darle el nombre `test` y *Create New Folder*). Después, nos crearemos un paquete: `jpa.familias.main`, lo seleccionamos con el botón derecho y *New*, *JUnit Test Case*. Al nuevo test case le podemos llamar `JpaTest` y aceptamos la versión JUnit 4.0 cuando el asistente nos lo pida.

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.familias.main;
2  import static org.junit.Assert.*;
3  import org.junit.Before;
4  import org.junit.Test;
5  import javax.persistence.EntityManager;
6  import javax.persistence.EntityManagerFactory;
7  import javax.persistence.Persistence;
8  import javax.persistence.Query;
9  import jpa.familias.modelo.Familia;
10 import jpa.familias.modelo.Persona;
11 import java.util.List;
12
13 public class JpaTest {
14         private static final String PERSISTENCE_UNIT_NAME = "gente";
15         private EntityManagerFactory factoria;
16         @Before
17         public void setUp() throws Exception {
18                 factoria = Persistence.createEntityManagerFactory(PERSISTENCE_UNIT_NAME);
19         EntityManager em = factoria.createEntityManager();
20         //Comenzar una nueva transaccion de tal forma que podamos hacer persistente una nueva entidad
21         em.getTransaction().begin();
22         //leer las entradas existentes
23         Query q = em.createQuery("select m from Persona m");
24         // En este punto Personas debe estar vacio
25         // tenemos entradas?
26         boolean createNewEntries = (q.getResultList().size() == 0);
27         // No, pues vamos a crearnos nuevas entradas
28         if (createNewEntries) {
29             assertTrue(q.getResultList().size() == 0);
30             Familia familia = new Familia();
31             familia.setDescripcion("Familia de los Martinez");
32             em.persist(familia);
33             for (int i = 0; i < 40; i++) {
34                 Persona persona = new Persona();
35                 persona.setNombre("Jaime_" + i);
36                 persona.setApellido("Martinez_" + i);
37                 em.persist(persona);
38                 // ahora que persista la relacion entre familia y persona
39                 familia.getMiembros().add(persona);
40                 em.persist(persona);
41                 em.persist(familia);
42             }
43         }
44         // Comprometer (commit) la transaccion, lo que ocasionara que la entidad sea almacenada en
45         //la base de datos
46         em.getTransaction().commit();
47         //Siempre hay que cerrar el EntityManager para que se guarden los recursos
48         em.close();
49         }
```

```
@Test
    public void comprobarGenteDisponible() {
        //Ahora vamos a comprobar la base de datos y ver si las entradas creadas estan alli
        //crer un nuevo EntityManager "fresco"
        EntityManager em = factoria.createEntityManager();
        //Realizar una consulta simple a todas las entidades Message
        Query q = em.createQuery("select m from Persona m");
        //Debemos tener 40 personas en la base de datos
        assertTrue(q.getResultList().size() == 39);
        em.close();
    }
```

```
@Test
    public void comprobarFamilia() {
        EntityManager em = factoria.createEntityManager();
        // Recorrer cada una de las entidades e imprimir cada uno de sus mensajes,
        //asi como la fecha de creacion
        Query q = em.createQuery("select f from Familia f");
        //deberiamos tener una sola familia con 40 personas
        assertTrue(q.getResultList().size() == 1);
        assertTrue(((Familia) q.getSingleResult()).getMiembros().size() == 40);
        em.close();
    }
```

```
@Test(expected = javax.persistence.NoResultException.class)
    public void eliminarPersona() {
        EntityManager em = factoria.createEntityManager();
        // Comenzar una nueva transaccioin local de tal forma que podamos hacer persistir una nueva entidad
        em.getTransaction().begin();
        Query q = em
                .createQuery("SELECT p FROM Persona p WHERE p.nombre = :nombre AND p.apellido = :apellido");
        q.setParameter("nombre", "Jaime_1");
        q.setParameter("apellido", "Martinez_1");
        Persona usuario = (Persona) q.getSingleResult();
        em.remove(usuario);
        em.getTransaction().commit();
        Persona persona = (Persona) q.getSingleResult();
        em.close();
    }
}
```

4.5 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), para lo cual sólo hay que ejecutar el proyecto *RunAs* como un *Junit Test*.

El método `setup()` creará unas cuentas entradas del test. Después de que se creen estas entradas, se leerán y se cambiará un campo de las entradas que después se salva en la base de datos.

4.6 Comprobación de la creación de la base de datos en Derby

Usaremos la herramienta `ij` incorporada en la distribución de Derby para cargar el motor de base de datos. El controlador de Derby embebido en esta herramienta se utilizará primero para crear y conectarnos a la base de datos ’relacionesDB’. También vamos a utilizar algunas declaraciones SQL básicas para consultar una tabla. Después de incluir el directorio `%DERBY_HOME%\bin` en la variable de entorno `PATH`, tenemos que ejecutar:
`java -jar %DERBY_HOME%\lib\derbyrun.jar ij` (o bien
`java -jar $DERBY_HOME/lib/derbyrun.jar ij` en Linux).

Una vez dentro del shell de `ij`, tecleamos:

```
CONNECT 'jdbc:derby:C:\Users\software\db-derby-10.14.2.0-bin\basesDatos\relacionesDB';
```

obteniendo una conexión con la base de datos `'relacionesDB'` que anteriormente creamos en esta parte del tutorial (ver 3.1).

Si ahora tecleamos: `SELECT * FROM TEST.PERSONA;`, obtendremos:

```
ID | APELLIDO   |NOMBRE | NONSENSEFIELD | FAMILIA_ID
------------------------------------------------------
------------------------------------------------------
------------------------------------------------------
2  |Martinez_0  | Jaime_0  |            | NULL
5  |Martinez_3  | Jaime_3  |            | NULL
34 |Martinez_32 | Jaime_32 |            | NULL
29 |Martinez_27 | Jaime_27 |            | NULL
 ...
39 filas seleccionadas
```

Tecleando: `SELECT * FROM TEST.FAMILIA;`, se obtendrá:

```
ID         |DESCRIPCION
-------------------------------------------------------------
-------------------------------------------------------------
1          |Familia de los Martinez
1 fila seleccionada
```

Para salir de `ij` hay que introducir `exit;`

Ahora, podríamos querer ejecutar órdenes SQL directamente sobre el servidor utilizando el cliente `ij`, para lo cual arrancaremos el servidor de red de Derby tecleando:
`java -jar %DERBY_HOME%\lib\derbyrun.jar server start`, que responderá con:

```
Sun Nov 15 09:33:52 CET 2020 : Se ha instalado el gestor de seguridad con la politica
 de seguridad de servidor basica.
Sun Nov 15 09:33:53 CET 2020 : Servidor de red Apache Derby: Se ha iniciado 10.14.2.0 - (1828579) y
esta listo para aceptar las conexiones en el puerto 1527
```

Posteriormente, tendremos que arrancar el cliente tecleando:
`java -jar %DERBY_HOME%\lib\derbyrun.jar ij` en otro terminal. Y utilizando este cliente, vamos a crear y abrir una conexión con el servidor por el puerto 1527:
`CONNECT 'jdbc:derby://localhost:1527/relacionesDB;create=true';`

La URL de conexión del cliente contiene información de red (nombre de host y número de puerto) que no se encuentra en la URL de una conexión utilizando el servidor Derby embebido, como en el ejemplo anterior. En este ejemplo, el motor de base de datos de Derby está integrado en el servidor de red y regresará datos al cliente `ij` (tendremos una configuración cliente/servidor). Por el contrario, estableciendo una conexión utilizando una URL embebida (una sin //localhost:1527/) provocará que el *motor Derby* se integre en la aplicación `ij` (es decir, tendremos una configuración incrustada). En esta configuración, varios programas cliente pueden conectarse al servidor de red y acceder a la base de datos simultáneamente, pero no hemos demostraremos esta capacidad aquí.

Para probarlo, nos crearemos una tabla ficticia con: `CREATE TABLE SEGUNDATABLA (ID INT PRIMARY KEY, NOMBRE VARCHAR(14));`

Posteriormente, insertaremos 3 filas en la tabla:
`INSERT INTO SEGUNDATABLA VALUES (100, 'CIEN'),(200, 'DOSCIENTOS'), (300,'TRESCIENTOS');.`

Para comprobar que funciona adecuadamente, tecleraremos:
`SELECT * FROM SEGUNDATABLA;` y `SELECT * FROM SEGUNDATABLA WHERE ID=200;.`

Y obtendremos las siguientes salidas, respectivamente:

```
ID         |NOMBRE
--------------------------
100        |CIEN
200        |DOSCIENTOS
300        |TRESCIENTOS
3 filas seleccionadas
```

```
ID         |NOMBRE
--------------------------
200        |DOSCIENTOS
1 fila seleccionada
```

Por último, saldremos correctamente del cliente tecleando `exit;` y pararemos el Derby Network Server: `java -jar %DERBY_HOME%\lib\derbyrun.jar server shutdown .`

De este modo, terminrá correctamente el servidor y obtendremos un mensaje parecido a:
`Sun Nov 15 10:08:31 CET 2020 : Servidor de red Apache Derby: cierre de 10.14.2.0 - (1828579).`

4.7 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.

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