Tutorial JPA (Java Persistence API)

Introducción

Java Persistence API (JPA) proporciona un estándar para gestionar datos relacionales en aplicaciones Java SE o Java EE, de forma que además se simplifique el desarrollo de la persistencia de datos.

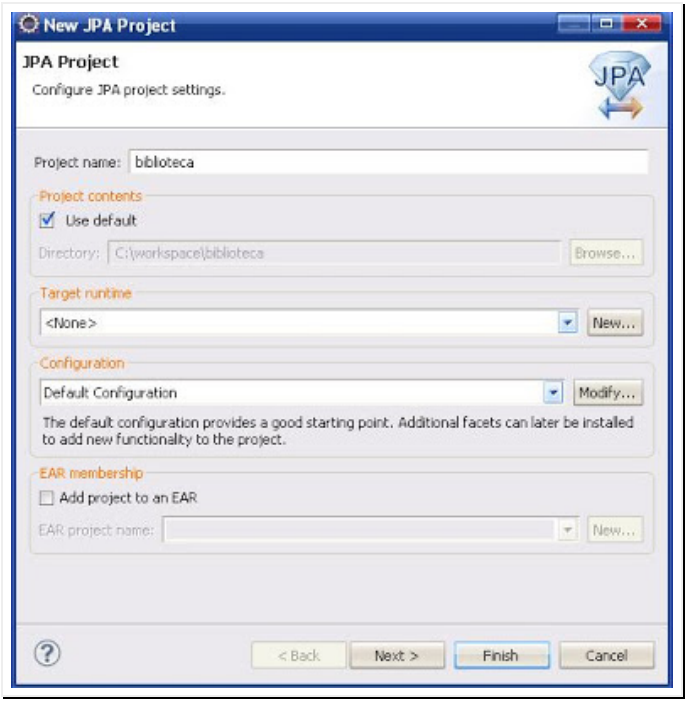
En su definición, ha combinado ideas y conceptos de los principales frameworks de persistencia, como Hibernate, Toplink y JDO, y de las versiones anteriores de EJB. Todos estos cuentan actualmente con una implementación JPA.

JPA se encarga de mapear una clase(Bean) a una tabla de la base de datos. de esta manera solo tenemos que escribir el código de nuestra clase con sus atributos y métodos y JPA se encarga de crear la tabla si esta no existe y relacionarla a nuestra base de datos.

JPA permite la persistencia no solo en bases de datos, sino también en otras formas como archivos de texto, planos y xml.

A través del tutorial iremos desarrollando un proyecto en eclipse sobre una base de datos MySQL. El proyecto consiste en crear una biblioteca en la cual tendremos las entidades libro, biblioteca, lector y bibliotecario las cuales representarán las tablas de la base de datos; analizaremos los conceptos de entidades, relaciones, herencia, manejo de excepciones y posteriormente un ejemplo de como se acceden, borran, y actualizan campos de la base de datos.

Crear un nuevo proyecto JPA

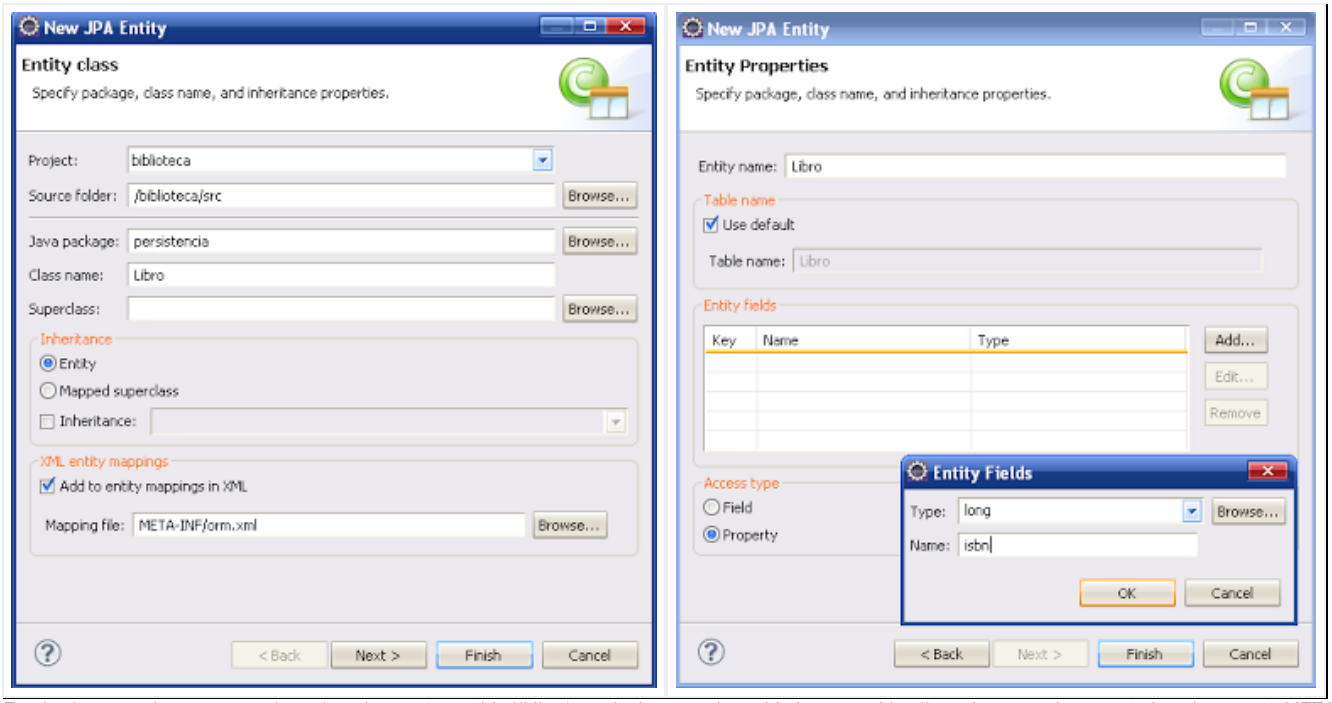


por ahora solo introduciremos el nombre del proyecto, luego organizaremos la configuración de la base de datos.

Creación de entidades

La creación de entidades nos permite relacionar una tabla de nuestra base de datos con una clase(Bean) de nuestro proyecto , para esto debemos hacer uso de las librerías que se encuentran en **javax.persistance.***

Eclipse nos permite generar una entidad de con sus atributos de la siguiente forma:



En el primer cuadro ponemos el nombre de nuestra entidad(Libro) y relacionamos la entidad a un archivo llamado orm.xml que estará en la carpeta META-INF, más adelante miraremos en que consiste este archivo y para que sirve. En el segundo cuadro crearemos los atributos de nuestra entidad y la escogemos forma de acceso.

Código generado:

```
@Entity
public class Libro implements serializable{

private Long isbn;
private String nombre;

public Libro();

public Libro(Long isbn, String nombre){
this.isbn = isbn;
this.nombre = nombre;
}

@Id
public Long getIsbn(){ return this.isbn; }
public void setIsbn(Long isbn){ this.isbn = isbn; }
public String getNombre(){ return this.nombre; }
public void setNombre(String nombre){ this.nombre = nombre; }
}
```

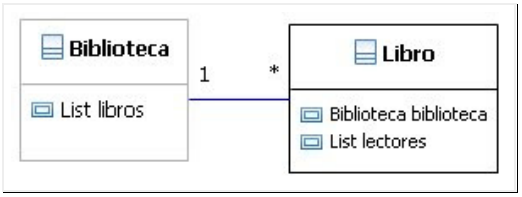
@Entity declara que la clase es una entidad.
@Id declara el identificador de nuestra entidad (llave primaria en nuestra tabla), la cual corresponde al atributo isbn de nuestra clase, sin embargo la especificación EJB3 sugiere que utilicemos la anotación en el elemento al cual se tiene acceso en este caso sobre el método; también es posible usar en vez de `@Id` la anotación `@EmbeddedId`.
La otra declaración (nombre) esta implícita (JPA asumirá por defecto que el nombre de ese atributo se representa con el mismo nombre en la declaración del campo de la base de datos) .

Dado el caso que necesitaramos accesar nuestra entidad de manera remota, se nos sugiere implementar la clase `Serializable`.

Con solo esto ya tenemos creada una “entidad” llamada “Libro” y podríamos insertar, actualizar o eliminar entradas en una tabla llamada “Libro” aunque esta aún no existiera.

Relaciones entre entidades:

Ahora si en nuestra base de datos quisieramos tener una entidad llamada biblioteca que tuviera relacionados muchos libros, entonces podríamos denotar la relación de varias maneras:



Unidireccional: Establece la relación en una sola dirección, en este caso desde libro hacia biblioteca.

```
@Entity
public class Libro implements serializable{

private Long isbn;
private String nombre;
private Biblioteca biblioteca;

public Libro();

public Libro(Long isbn, String nombre, Biblioteca biblioteca){
this.isbn = isbn;
this.nombre = nombre;
this.biblioteca = biblioteca;
}

@Id
public Long getIsbn(){ return this.isbn; }
public void setIsbn(Long isbn){ this.isbn = isbn; }
public String getNombre(){ return this.nombre; }
public void setNombre(String nombre){ this.nombre = nombre; }

@ManyToOne
@JoinColumn(name = "biblioteca_id")
public Biblioteca getBiblioteca(){ this.biblioteca }
}
```

@ManyToOne Define que tenemos una relacion de muchos a uno, en la cual tenemos muchos libros pertenecen a una biblioteca.
@JoinColumn Establece la columna biblioteca_id en nuestra tabla libro.

Bidireccional: Además de tener la relación anterior podríamos anexar una nueva relación desde biblioteca hacia libro, entonces tendríamos dos relaciones.

Creamos una nueva entidad (Biblioteca) con el siguiente contenido:

```
@Entity
public class Biblioteca implements serializable
{
private Long id;
private String nombre;
private List libros;


 public Biblioteca(){}

 public Biblioteca(String nombre,List libros){
 this.nombre = nombre;
 this.libros = libros;
 }

 @Id
 @GeneratedValue(strategy=GenerationType.SEQUENCE)
 public Long getId(){ return this.id; }

 public String getNombre(){ return this.nombre; }

 @OneToMany(mappedBy="biblioteca")
 public List getLibros(){ return this.libros; }
}
```

@GeneratedValue indica que el valor se generará automáticamente por secuencia, de esta manera podremos tener una columna auto increment que corresponderá a la llave primaria.

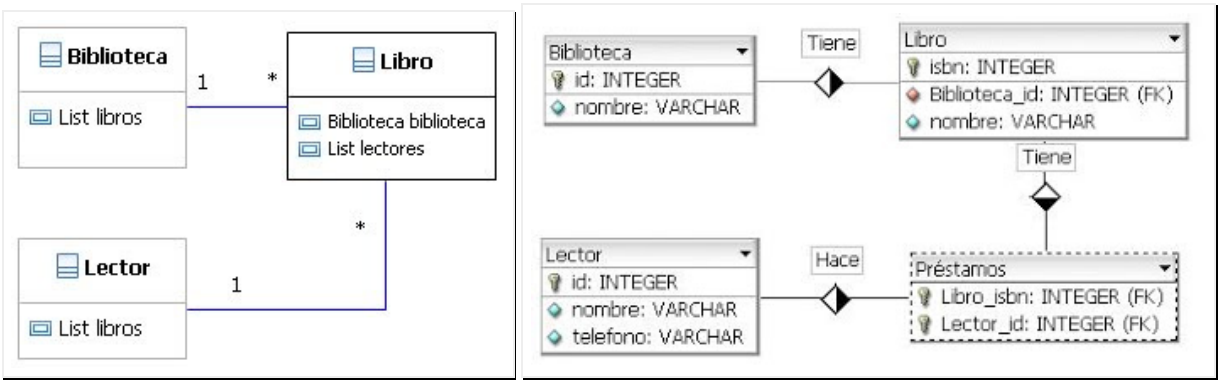
@OneToMany establece la relación de una bibliteca con muchos libros en donde cada objeto libro tendra una referencia a la entidad biblioteca.

De esta forma tendríamos dos relaciones, una desde libro hacia biblioteca, y otra desde biblioteca hacia libro.

Es importante anotar que si se desea tener una relación bidireccional se hace necesario especificar las anotaciones correspondientes en ambas Entidades (Biblioteca, Libro).

Las anotaciones de cardinalidad existentes para relacionar entidades pueden ser **OneToMany, OneToOne, ManyToMany, ManyToOne**.

Dado el caso que necesitaramos establecer una relación de muchos a muchos podríamos verlo de la siguiente manera.



Un lector puede prestar muchos libros, y un libro puede ser prestado por muchos lectores. Esta relación muchos a muchos tiene diferentes formas de implementación según javax.persistance, sin embargo usaremos la más comúmente usada que es bidireccional en la cual existe una entidad dueña de la relación y una no dueña.

Nota: *Si en la relación no se tienen tributos descriptivos como una fecha del préstamo, una fecha de entrega etc, no se hace necesario crear una tercer entidad, ya que en la entidad dueño se infiere el uso de esta.*

Creamos una nueva entidad (Lector) con el siguiente contenido:

```
@Entity
public class Lector implements serializable
{
private Long id;
private String nombre;
private Long carnetLector;
private List libros;


 public Lector(){}

 public Lector(Long id,String nombre,Long carnetLector,List libros){

 this.id = id;
 this.nombre = nombre;
 this.carnetLector = carnetLector;
 this.libros = libros;
 }

 @Id
 public Long getId(){ return this.id; }
 public void setId(Long id){ this.id = id; }
 public String getNombre(){ return this.nombre; }
 public Long getCarnetLector(){ return  this.carnetLector; }
 public void setCarnetLector(Long carnetLector){  this.carnetLector = carnetLector; }

 @ManyToMany
 @JoinTable(name="prestamos",
 joinColumns={
 @JoinColumn(name="Lector_id", nullable=false)
```

```
 }
 , inverseJoinColumns={
 @JoinColumn(name="Libro_isbn", nullable=false)
 }
 )
 public List getLibros(){ return this.libros; }
}
```

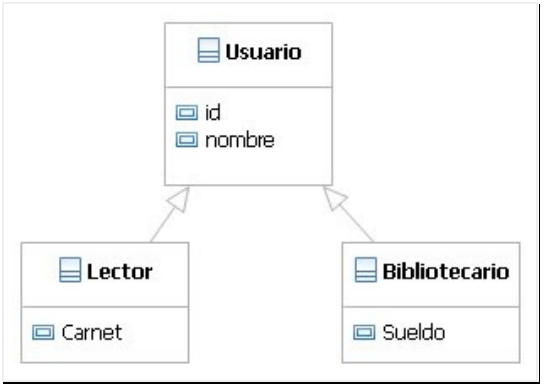
Lector tiene una lista de los libros que ha prestado.
@ManyToMany indica el tipo de relación.
@JoinTable(name = "prestamos" ... crea una tabla (prestamos) con el id del lector(Lector_id) y el id del libro(Libro_isbn) para obtener una fila única que represente la relación dentro de la tabla lector.

```
@Entity
public class Libro implements serializable{

private Long isbn;
private String nombre;
private Biblioteca biblioteca;
private List lectores;

public Libro();

public Libro(Long isbn, String nombre, Biblioteca biblioteca, List lectores){
this.isbn = isbn;
this.nombre = nombre;
this.biblioteca = biblioteca
this.lectores = lectores;
}

@Id
public Long getIsbn(){ return this.isbn; }
public void setIsbn(Long isbn){ this.isbn = isbn; }
public String getNombre(){ return this.nombre; }
public void setNombre(String nombre){ this.nombre = nombre; }

@ManyToOne
@JoinColumn(name = "biblioteca_id")
public Biblioteca getBiblioteca(){ this.biblioteca }

@ManyToMany(mappedBy="libros")
public List getLectores(){ return this.lectores; }
public void setLectores(List lectores){ this.lectores = lectores; }
}
```

@ManyToMany(mappedBy="libros") cada objeto lector tendrá una referencia a la lista de sus libros.

Herencia entre entidades

Resulta que en nuestro proyecto deseamos guardar la información de los lectores, pero también deseamos guardar la información de los empleados de la bliblioteca. Pueden existir datos que sean comunes para ambos tipos de usuarios, esto nos hace pensar en no repetir información innecesariamente.



La ventaja de trabajar con objetos nos permite hacer uso extensivo de la herencia. Tanto en Hibernate como en JPA se definen tres estrategias para mapear relaciónes de herencia entre clases a tablas de nuestra base de datos:

- **Una sola tabla para guardar toda la jerarquía de clases:** Tiene la ventaja de ser la opción que mejor rendimiento da, ya que sólo es necesario acceder a una tabla (está totalmente desnormalizada). Tiene como inconveniente que todos los campos de las clases hijas tienen que admitir nulos, ya que cuando guardemos un tipo, los campos correspondientes a los otros tipos de la jerarquía no tendrán valor.
- **Una tabla para el padre de la jerarquía, con las cosas comunes, y otra tabla para cada clase hija con las cosas concretas.**Es la opción más normalizada, y por lo tanto la más flexible (puede ser interesante si tenemos un modelo de clases muy cambiante), ya que para añadir nuevos tipos basta con añadir nuevas tablas y si queremos añadir nuevos atributos sólo hay que modificar la tabla correspondiente al tipo donde se está añadiendo el atributo. Tiene la desventaja de que para recuperar la información de una clase, hay que ir haciendo join con las tablas de las clases padre.
- **Una tabla independiente para cada tipo.** En este caso cada tabla es independiente, pero los atributos del padre (atributos comunes en los hijos), tienen que estar repetidos en cada tabla. En principio puede tener serios problemas de rendimiento, si estamos trabajando con polimorfismo, por los SQL UNIONS que se tienen que hacer para recuperar la información. Sería la opción menos aconsejable. Tanto es así que en la versión 3.0 de EJBs, aunque está recogida en la especificación, no es obligatoria su implementación.

Nos limitaremos a explicar las dos primeras implementaciones, uds pueden escoger cual usar para su proyecto.

Una sola tabla para guardar toda la jerarquía de clases: una sola tabla para toda la jerarquía de clases. Es, posiblemente, la implementación más sencilla.

La tabla tiene el campo "**discriminador**", este campo será el que se use como discriminador para saber a que tipo concreto se refiere el registro. Por defecto en este campo se guardará el nombre de la clase (sin el nombre del paquete).

Implementación

Creamos una nueva entidad (Usuario) con el siguiente contenido:

```
@Entity
//@Inheritance(Strategy=InheritanceType.SINGLE_TABLE)
public class Usuario implements serializable{

private Long id;
private String nombre;

public Usuario();

public Usuario(Long id, String nombre){
this.id = id;
this.nombre = nombre;
}

@Id
public Long getId(){ return this.id; }
public void setId(Long id){ this.id = id; }
public String getNombre(){ return this.nombre; }
public void setNombre(String nombre){ this.nombre = nombre; }
}
```

Nótese que la línea donde está definido el tipo de mapeo para la herencia (justo encima de la declaración de la clase) está comentada. Es decir, no es necesaria ya que `SINGLE_TABLE` es el valor por defecto. En el ejemplo se a puesto comentada la línea para que el lector vea como sería la definición de la estrategia que se va a utilizar para mapear la herencia de estas entidades.

Veamos las otras clases:

Nosotros ya habíamos creado una entidad (Lector), ahora solo debemos modificarla y heredar de la entidad Usuario, pueden ver como nos queda más corta:

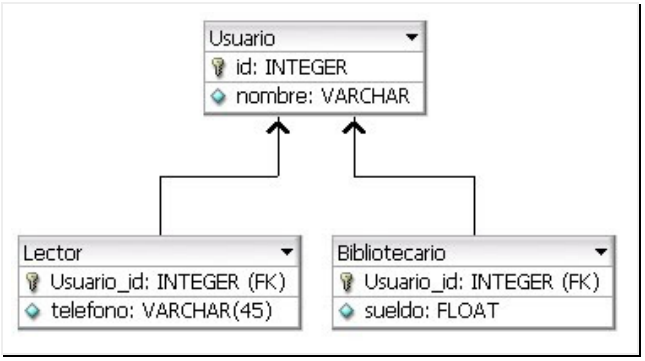
```
@Entity
public class Lector extends Usuario implements serializable{

private Long carnetLector;

public Long getCarnetLector(){ return this.carnetLector; }
public void setCarnetLector(Long carnetLector){ this.carnetLector = carnetLector; }
}
```

Creamos una nueva entidad (Bibliotecario) con el siguiente contenido:

```
@Entity
public class Bibliotecario extends Usuario implements serializable{

private Long salario;

public Long getSalario(){ return this.salario; }
public void setSalario(Long salario){ this.salario = salario; }
}
```

Se puede ver como las clases derivadas son extremadamente sencillas, y no especifican ninguna anotación especial. Tampoco ninguna de ellas tiene un atributo que especifique el id (la PK) del objeto. Esto no es necesario ya que esta propiedad se hereda del padre `Usuario`.

Una tabla para cada subclase ("join" de tablas): Recordamos que en esta estrategia tendremos una tabla para los atributos comunes (los atributos del padre), y una tabla por cada subclase (con exclusivamente los atributos nuevos que aporta la subclase). Para conseguir un objeto se hará join con las tablas necesarias.



Implementacion:

```
@Entity
@Inheritance(Strategy=InheritanceType.JOINED)
public class Usuario implements serializable{

private Long id;
private String nombre;

public Usuario();

public Usuario(Long id, String nombre){
this.id = id;
this.nombre = nombre;
}

@Id
public Long getId(){ return this.id; }
public void setId(Long id){ this.id = id; }
public String getNombre(){ return this.nombre; }
public void setNombre(String nombre){ this.nombre = nombre; }
}
```

Esta clase, el padre de la jerarquía, es igual que en el ejemplo anterior, salvo que está vez si es obligatorio definir la anotación para indicar la estrategia a seguir para mapear la herencia (`InheritanceType.JOINED`).

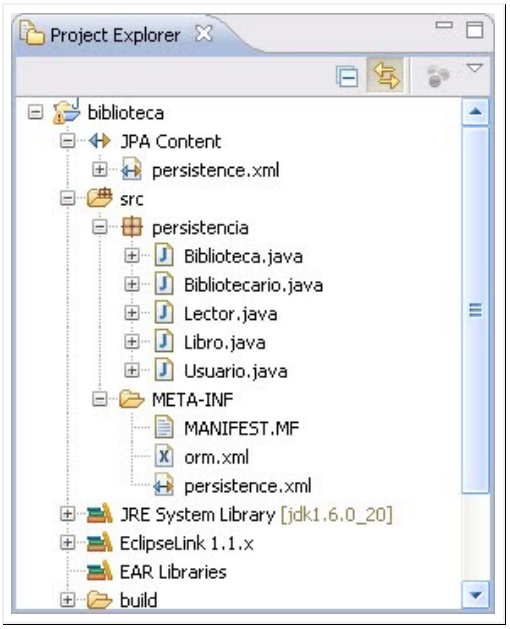
Veamos las clases hijas:

```
@Entity
@PrimaryKeyJoinColumn(name="Usuario_id")
public class Lector extends Usuario implements serializable{
```

```
private Long carnetLector = 0;

public Long getCarnetId(){ return this.carnetLector; }
public void setCarnetId(Long carnetLector){ this.carnetLector = carnetLector; }
}

@Entity
@PrimaryKeyJoinColumn(name="Usuario_id")
public class Bibliotecario extends Usuario implements serializable{

private Long salario = 0;

public Long getSalario(){ return this.salario; }
public void setSalario(Long salario){ this.salario = salario; }
}
```

Comparando con el ejemplo anterior, ambas clases lo único que hacen es indicar el nombre del campo que contiene la clave ajena (la FK) para hacer el join con el padre de la jerarquía.

Al final tendremos una estructura de archivos que se podría ver de la siguiente manera:



Independiente del entorno de desarrollo que usemos, básicamente lo que importa es que nuestro sistema posea un sistema de archivos de la siguiente forma:

En persistencia residen los archivos de nuestras entidades y en META-INF la información relacionada a ellos.

Ahora, como relacionamos esta entidad a una base datos?

Pues bien, para eso utilizaremos un archivo llamado persistence.xml que pondremos en la carpeta “src/META-INF” de nuestra aplicación.