# Chuleta JPA

*Rodrigo García Carmona* (v1.1.2)


Esta guía está pensada para SQLite y EclipseLink.

## Preparar el proyecto

Para poder usar JPA tenemos que añadir el proveedor de JPA (en nuestro caso EclipseLink) al proyecto. Para hacer eso importamos estos dos archivos jar:

- *eclipselink.jar*
- *javax.persistence.jar*

También debemos añadir una carpeta *META-INF* a nuestra carpeta *src* (código fuente) y poner dentro un archivo *persistence.xml* que debe ser similar a éste:

```
<?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="provider-name" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>

    <!-- Entity Classes -->
    <class>a.java.package.EntityClass1</class>
    <class>a.java.package.EntityClass2</class>
    <class>a.java.package.EntityClass3</class>

    <properties>
        <!-- Connection properties -->
        <property name="javax.persistence.jdbc.driver"
            value="org.sqlite.JDBC" />
        <property name="javax.persistence.jdbc.url"
            value="jdbc:sqlite:database-url" />
        <!-- Fill if we need user and password -->
        <property name="javax.persistence.jdbc.user" value="" />
        <property name="javax.persistence.jdbc.password" value="" />

        <!-- JPA creates the database schema if it doesn't exist -->
        <property name="eclipselink.ddl-generation" value="create-tables" />
    </properties>

</persistence-unit>
```

```
</persistence>
```

En este archivo debemos cambiar:

- El *provider-name*. Haremos referencia a este nombre durante la creación del Entity Manager.
- Las *Entity Classes* (clases de entidad, clases que representan entidades de la base de datos) que vamos a modelar.
- La *database-url* que apunta al archivo que contiene la base de datos.

Por último, debemos anotar (ver más adelante en este mismo documento) todas las clases de entidad que vayamos a gestionar usando JPA.

**Un aviso importante:** Antes de ejecutar código JPA tenemos que estar **completamente seguros** de que hemos creado las tablas de la base de datos.

# Trabajando con JPA

Para empezar a trabajar con JPA, lo primero que tenemos que hacer es crear un Entity Manager, que cumple el rol equivalente a la conexión en JDBC:

```
EntityManager em = Persistence.createEntityManagerFactory("provider-name").
                    createEntityManager();
```

Como con JDBC, si trabajamos con SQLite tenemos que activar el soporte para restricciones en foreign keys. Para ello debemos ejecutar las siguientes líneas de código inmediatamente después de crear el Entity Manager:

```
em.getTransaction().begin();
em.createNativeQuery("PRAGMA foreign_keys=ON").executeUpdate();
em.getTransaction().commit();
```

Tras acabar de trabajar con la base de datos, tenemos que acordarnos de cerrar el Entity Manager:

```
em.close();
```

Las transacciones se utilizan con las operaciones que modifican la base de datos, es decir: *create*, *update* y *delete*.

Para iniciar una transacción escribimos:

```
em.getTransaction().begin();
```

Y para finalmente ejecutarla escribimos:

```
em.getTransaction().commit();
```

Si queremos deshacer todos los cambios (*rollback*) desde el último commit escribimos:

```
em.getTransaction().rollback();
```

La mayor parte del tiempo la base de datos y los objetos tendrán el mismo estado, pero cuando trabajemos en un entorno multiusuario o en remoto, podría darse el caso de que quisiéramos asegurarnos de que están sincronizados. Para asegurarnos de que es así, disponemos de los siguientes dos métodos:

Para obligar a la base de datos a que refleje el estado del modelo de objetos:

```
em.flush();
```

Para obligar al modelo de objetos a que refleje el estado de la base de datos:

```
em.refresh();
```

# Operaciones CRUD

Podemos llevar a cabo las cuatro operaciones CRUD usando JPA.

**Create (Crear):**

```
em.persist(object);
```

**Read (Leer):**

```
Query q = em.createNativeQuery("SELECT Query", ClassName.class);
```

Y después o esto:

```
List<ClassName> list = q.getResultList();
```

O esto (únicamente si estamos seguros de que sólo vamos a obtener un resultado):

```
ClassName object = (ClassName) q.getSingleResult();
```

**Update (Actualizar):**

```
object.setMethod(setValue);
```

Es importante recordar que los cambios en una asociación tienen que llevarse a cabo en ambos lados de la misma:

```
employee.setDepartment(department);
department.addEmployee(employee);
```

**Delete (Borrar):**

```
em.remove(object);
```

# Clases Entidad JPA

Todas las clases entidad deben tener las siguientes características:

- Implementar la clase *Serializable*.
- Un constructor sin parámetros.
- Métodos accesores y modificadores para todos los atributos.
- Métodos equals y hashcode que sólo usen el atributo de la clave primaria.
- Asociaciones deben ser bidireccionales.

Y se recomienda que tengan las siguientes características:

- Un método toString.
- Métodos add y remove para los atributos que sean listas.

Los atributos deben ser de uno de los siguientes tipos:

- *Integer* (INTEGER en SQLite)
- *Double* (REAL en SQLite)
- *String* (TEXT en SQLite)
- *java.sql.Date* (DATE en SQLite)
- *byte[]* (BLOB en SQLite)
- Otra clase entidad o una lista de elementos de otra clase entidad (representando una relación con foreign keys en SQL)

La tabla que corresponde a la clase entidad debe cumplir:

- Poseer una PRIMARY KEY con la restricción AUTOINCREMENT (en SQLite).

Debemos anotar todas las clases de entidad de la siguiente forma:

```
@Entity
@Table(name = "table_name")
public class EntityClass implements Serializable {

    @Id
    @GeneratedValue(generator="generator_name")
    @TableGenerator(name="generator-name", table="sqlite_sequence",
           pkColumnName="name", valueColumnName="seq",
           pkColumnValue="tabl_name")
    private Integer id;
    ...
}
```

- *table_name* debe ser el nombre de la tabla.
- *generator_name* puede ser cualquier nombre que escojamos, pero debe ser diferente a todos los demás *generator_names*.

Ten en cuenta que la estructura de la anotación TableGenerator que aparece aquí es específica de SQLite.

Anotamos BLOBs de la siguiente forma:

```
@Basic(fetch=FetchType.LAZY)
@Lob
```

```
private byte[] photo;
```

El tipo de recuperación LAZY indica que este atributo sólo va a ser recuperado de la base de datos cuando verdaderamente es accedido, es decir, cuando se invoque al método accesor correspondiente.

El resto de atributos son mapeados automáticamente con las columnas del mismo nombre en la tabla correspondiente. Si queremos enlazar un atributo con una columna de diferente nombre debemos usar la anotación Column.

```
@Column(name="surname)
private String familyName;
```

# Relaciones Uno a Uno

Aquí tienes un ejemplo de cómo implementar una relación uno a uno usando JPA.

## Tablas

**employees**

| id | name | surname | salary | address_id |
|---|---|---|---|---|
| 1 | Bob | Way | 50000 | 6 |
| 2 | Sarah | Smith | 60000 | 7 |

**addresses**

| id | street | city | country |
|---|---|---|---|
| 6 | Mayor 5 | Madrid | Spain |
| 7 | Fernán 17 | Burgos | Spain |

## Clases

```
@Entity
@Table(name="employees")
public class Employee implements Serializable {
    @Id
    // Other Id annotations as needed
    private Integer id;
    ...
    @OneToOne(fetch=FetchType.LAZY)
    @JoinColumn(name="address_id")
    private Address address;
    ...
}
```

```
@Entity
@Table(name="addresses")
public class Address implements Serializable {
    @Id
    // Other Id annotations as needed
    private Integer id;
    ...
    @OneToOne(fetch=FetchType.LAZY, mappedBy="address")
    private Employee owner;
    ...
}
```

# Relaciones Muchos a Uno

Aquí tienes un ejemplo de cómo implementar una relación muchos a uno usando JPA.

## Tablas

### employees

| id | name | surname | salary |
|---|---|---|---|
| 1 | Bob | Way | 50000 |
| 2 | Sarah | Smith | 60000 |

### phones

| id | number | type | owner_id |
|---|---|---|---|
| 1 | 917555555 | home | 1 |
| 2 | 669696969 | work | 1 |
| 3 | 666666666 | work | 2 |

## Clases

```
@Entity
@Table(name="employees")
public class Employee implements Serializable {
     @Id
    // Other Id annotations as needed
    private Integer id;
    ...
    @OneToMany(mappedBy="owner")
    private List<Phone> phones;
    ...
}
```

```
@Entity
@Table(name="addresses")
public class Phone implements Serializable {
    @Id
    // Other Id annotations as needed
    private Integer id;
    ...
    @ManyToOne(fetch=FetchType.LAZY)
    @JoinColumn(name="owner_id")
    private Employee owner;
    ...
}
```

# Relaciones Muchos a Muchos

Aquí tienes un ejemplo de cómo implementar una relación muchos a muchos usando JPA.

## Tablas

### employees

| id | firstname | lastname |
|---|---|---|
| 1 | Bob | Way |
| 2 | Sarah | Smith |

### projects

| id | name |
|---|---|
| 1 | GIS |
| 2 | SIG |

### proj_emp

| emp_id | proj_id |
|---|---|
| 1 | 1 |
| 1 | 2 |
| 2 | 1 |

## Clases

```
@Entity
@Table(name="employees")
public class Employee implements Serializable {
```

```
public class Employee implements Serializable {
      @Id
    // Other Id annotations as needed

    private Integer id;
    ...
    @ManyToMany
    @JoinTable(name="proj_emp",
           joinColumns={@JoinColumn(name="emp_id", referencedColumnName="id")},
           inverseJoinColumns={@JoinColumn(name="proj_id", referencedColumnName="id")})
    private List<Project> projects;
    ...
}

@Entity
@Table(name="projects")
public class Project implements Serializable {
    @Id
    // Other Id annotations as needed
    private Integer id;
    ...
    @ManyToMany(mappedBy="projects")
    private List<Employee> employees;
    ...
}
```

# Relaciones Muchos a Muchos con Clase de Asociación

Cuando una relación muchos a muchos posee una clase de asociación, nos vemos obligados a representar esta información adicional en forma de una clase extra. Cuando ocurre esto, la relación muchos a muchos se simula usando dos relaciones muchos a uno entre la clase de asociación y las dos clases relacionadas.

## Tablas

**employees**

| id | firstname | lastname |
|---|---|---|
| 1 | Bob | Way |
| 2 | Sarah | Smith |

**projects**

| id | name |
|---|---|
| 1 | GIS |
| 2 | SIG |

**proj_emp**

| emp_id | proj_id | is_lead |
|---|---|---|
| 1 | 1 | true |
| 1 | 2 | false |
| 2 | 1 | false |

## Clases

```
@Entity
@Table(name="employees")
public class Employee implements Serializable {
      @Id
    // Other Id annotations as needed
    private Integer id;
    ...
    @OneToMany(mappedBy="employee")
    private List<ProjectAssociation> projects;
    ...
}

@Entity
@Table(name="projects")
public class Project implements Serializable {
      @Id
    // Other Id annotations as needed
    private Integer id;
    ...
    @OneToMany(mappedBy="project")
    private List<ProjectAssociation> employees;
    ...
}

@Entity
@Table(name="proj_emp")
@IdClass(ProjectAssociationId.class)
public class ProjectAssociation implements Serializable {

    @Id
    private Integer emp_id;
    @Id
    private Integer proj_id;
    @Column(name="is_lead")
    private boolean isTeamLead;
    @ManyToOne
    @PrimaryKeyJoinColumn(name="emp_id", referencedColumnName="id")
    private Employee employee;
    @ManyToOne
    @PrimaryKeyJoinColumn(name="proj_id", referencedColumnName="id")
    private Project project;
```

```
    ...
}

public class ProjectAssociationId implements Serializable {

    private Integer emp_id;
    private Integer proj_id;

    // equals() and hashcode() using both ids
    public int hashCode() {
           return (int)(employeeId + projectId);
    }

    public boolean equals(Object object) {
           if (object instanceof ProjectAssociationId) {
               ProjectAssociationId otherId = (ProjectAssociationId) object;
               return (otherId.employeeId == this.employeeId) &&
                (otherId.projectId == this.projectId);
           }
           return false;
    }

}
```

Fíjate que, en el ejemplo, hemos tenido que crear una clase especial llamada *ProjectAssociationID*, cuyo único propósito es calcular la clave primaria de la clase de asociación. Es necesario hacer esto porque dicha clave primaria es la combinación de dos foreign keys.

Aquí tienes un ejemplo de cómo crear un objeto de la clase de asociación en esta relación muchos a muchos:

```
public void addEmployee(Employee employee, boolean teamLead) {
       ProjectAssociation association = new ProjectAssociation();
       association.setEmployee(employee);
       association.setProject(this);
       association.setEmployeeId(employee.getId());
       association.setProjectId(this.getId());
       association.setIsTeamLead(teamLead);

       this.employees.add(association);
       // Also add the association object to the employee.
       employee.getProjects().add(association);
}
```

El código listado anteriormente está diseñado para formar parte de la clase *Project*.