# -Manual de Teoría-
# JPA e Hibernate

# INDICE

# 1. PERSISTENCIA

A lo largo de este modulo vamos a trabajar con 3 frameworks de persistencia o también conocidos como ORM (Object Relational Mapping).

Toda aplicación empresarial necesita tener sus datos almacenados en un sistema persistente como una base de datos.

La persistencia de datos se puede llevar a cabo de múltiples formas, la más conocida es JDBC. El problema que nos encontramos con JDBC es que los datos que manejamos en la aplicación son objetos, un cliente, una factura, ...etc. Sin embargo cuando queremos persistir estos datos la base de datos funciona en un paradigma diferente, el modelo relacional. Por esta razón nos vemos obligados a convertir nuestros objetos en registros para poder persistirlos y hacer lo contrario, convertir el registro leído en un objeto al realizar operaciones de lectura o recuperación de la BBDD.

Los ORM permiten realizar estas tareas de una forma totalmente transparente para el programador quien se encargará de persistir objetos y el propio framework será el encargado de transformarlo en registros y viceversa.

# 2. JPA

## INTRODUCCION

JPA es el acrónimo de Java Persistence API y se podría considerar como el estándar de los frameworks de persistencia.

En JPA utilizamos anotaciones como medio de configuración.

## ENTIDADES

Consideramos una entidad al objeto que vamos a persistir o recuperar de una base de datos. Se puede ver una entidad como la representación de un registro de la tabla.

Toda entidad ha de cumplir con los siguientes requisitos:

- Debe implementar la interface Serializable
- Ha de tener un constructor sin argumentos y este ha de ser público.
- Todas las propiedades deben tener sus métodos de acceso get() y set().

Para crear una entidad utilizamos la anotación @Entity, con ella marcamos un POJO como entidad.

```
@Entity
public class Persona implements Serializable
```

**Gráfico 1. Configuración de un POJO como entidad.**

## MAPEOS DE ENTIDADES

En este apartado vamos a ver las anotaciones más utilizadas para mapear cada una de las propiedades de la entidad contra la BBDD.

### @ID

Se utiliza para marcar una propiedad como clave primaria tanto si es una PK (Primary Key) simple, de un solo cambio o si forma parte de una PK compuesta, de varios campos.

```
@Entity
public class Persona implements Serializable {

	@Id
	private String nif;
```

**Gráfico 2. Marcamos la propiedad nif como PK**

Como sabemos las PK no pueden aceptar valore repetidos, estos han de ser únicos y no nulos.

En este caso en la clase String está implementado el método equals() por lo que no tenemos que hacer nada más.

Toda clave primaria debe tener implementado este método para poder comprobar si ya existe este valor de la PK en la tabla.

## @IDCLASS

Esta anotación nos sirve para poder anotar una clave primaria compuesta.

En una clase aparte definimos la clave primaria compuesta cumpliendo los siguientes requisitos:

- La clase ha de implementar la interface Serializable.
- En la clase creamos una propiedad por cada campo que forma parte de la PK.
- Estas propiedades deben tener un método get() y set() cada una de ellas.
- También debemos facilitar un constructor sin argumentos.
- Debemos implementar los métodos equals() y hashCode().

```java
public class PersonaPK implements Serializable{

    private Long telefono;
    private String nif;

    public PersonaPK() {
    }

    @Override
    public boolean equals(Object obj) {...}

    @Override
    public int hashCode() {...}
```

**Gráfico 3. Clase que define la PK compuesta**

En la entidad utilizamos la anotación @IdClass para especificar la clase que utilizamos como clave primaria compuesta.

```java
@Entity
@IdClass(PersonaPK.class)
public class Persona implements Serializable {

    @Id
    @Column(name = "id_telefono", nullable = false)
    private Long telefono;


    @Id
    @Column(name = "id_nif", nullable = false)
    private String nif;
```

**Gráfico 4. Creación de entidad con PK compuesta**

## @EMBEDDABLE

Cada una de las propiedades definida en la entidad, pasará a ser un campo en la tabla.

Esta anotación nos permite embeber una clase. Veámoslo con un ejemplo.

```java
@Embeddable
public class Direccion implements Serializable{

    private String calle;
    private String localidad;

    @Column(name="codigo_postal")
    private int cp;

    public Direccion() {
    }
```

**Gráfico 5. Creación de una clase Embeddable**

La clase Direccion declara tres propiedades calle, localidad y cp. La anotación @Embeddable especifica que toda propiedad que utilice esta clase como tipo en la entidad, será persistida con este formato.

| LOCALIDAD | CALLE | CODIGO_POSTAL |
|---|---|---|
| Madrid | Gran Via | 28014 |
| Madrid | Mayor | 28014 |

**Gráfico 6. Campos que se crearán en la tabla**

## @EMBEDDED

Con esta anotación hacemos que la propiedad de tipo Dirección se persista según la clase Embeddable creada anteriormente.

```java
@Embedded
private Direccion direccion;
```

**Gráfico 7. La propiedad direccion se persiste con las propiedades de la clase Direccion**

Si no utilizásemos la anotación se generaría un campo direccion con el identificador del objeto.

## @EMBEDDEDID

Es otra alternativa a la creación de claves primarias compuestas. Utilizamos esta anotación para especificar la propiedad que actúa como PK.

```
@Embeddable
public class PersonaPK implements Serializable{

    private Long telefono;
    private String nif;

    public PersonaPK() {
    }

    @Override
    public boolean equals(Object obj) {...}

    @Override
    public int hashCode() {...}
```

**Gráfico 8. Marcamos la clase PK como Embeddable**

A la hora de persistir la propiedad personapk, se generará la misma estructura que la clase PersonaPK.

```
@Entity
public class Persona implements Serializable {

    @EmbeddedId
    PersonaPK personapk;
```

**Gráfico 9. Declaración de clave primaria compuesta**

## @TABLE

Mediante esta anotación indicamos el nombre de la tabla donde se persistirán o recuperaran los datos de la entidad.

Si no la utilizásemos coge como nombre de la tabla el nombre de la clase. Según el ejemplo sería Persona.

```
@Entity
@IdClass(PersonaPK.class)
@Table(name = "Ejemplo1_PERSONAS")
public class Persona implements Serializable {
```

**Gráfico 10. Especificar nombre de la tabla**

## @COLUMN

Lo mismo ocurre con las columnas, podemos especificar su nombre, tamaño, ...etc. Cada propiedad de la entidad puede utilizar una anotación @Column para establecer las propiedades de la columna en la tabla.

Si no se utiliza, coge como nombre de columna el nombre de la propiedad.

```
@Id
@Column(name = "id_nif", nullable = false)
private String nif;
```

**Gráfico 11. La propiedad nif se mapea al campo con nombre id_nif y no acepta valores nulos**

```
@Column(length = 1)
private char sexo;
```

**Gráfico 12. El campo sexo tiene un solo carácter como longitud**

```
@Lob
@Basic(fetch = FetchType.LAZY)
@Column(table = "Ejemplo1_CV")
private String cv;
```

**Gráfico 13. El campo cv se genera en otra tabla con nombre Ejemplo1_CV**

## @SECONDARYTABLE

Como vemos en el último ejemplo una entidad puede utilizar tablas secundarias para almacenar datos, en el ejemplo el curriculum vitae de la persona se almacena en otra tabla.

Veamos cómo utilizar la anotación @SecondaryTable para definir tablas secundarias.

En esta anotación utilizamos dos atributos:

- El atributo name nos permite poner un nombre a la tabla.
- El atributo pkJoinColumns unimos la tabla primaria y la tabla secundaria.

Para esta última operación utilizamos otra anotación @PrimaryKeyJoinColumn cuyos atributos son:

- name; es el nombre del campo generado en la tabla secundaria
- referencedColumnName; especificamos el nombre del campo de la tabla primaria por el cual se unen.

```
@Entity
@IdClass(PersonaPK.class)
@Table(name = "Ejemplo1_PERSONAS")
@SecondaryTable(name = "Ejemplo1_CV",
                pkJoinColumns = {
                    @PrimaryKeyJoinColumn(name = "id_telefono",
                                          referencedColumnName = "id_telefono"),
                    @PrimaryKeyJoinColumn(name = "id_nif",
                                          referencedColumnName = "id_nif")
})
public class Persona implements Serializable {

    @Id
    @Column(name = "id_telefono", nullable = false)
    private Long telefono;

    @Id
    @Column(name = "id_nif", nullable = false)
    private String nif;
```

**Gráfico 14. Declaración de tabla secundaria**

Veamos la estructura de ambas tablas:

| ID_TELEFONO | ID_NIF | ESTADO | SEXO | NOMBRE | EDAD | FECHANACIMIENTO | LOCALIDAD | CALLE | CODIGO_POSTAL |
|---|---|---|---|---|---|---|---|---|---|
| 616111222 | 22222221-A | CASADO | M | Maria | 31 | 1998-03-01 | Madrid | Gran Via | 28014 |
| 616111111 | 11111111-A | CASADO | V | Pepe | 31 | 1999-01-01 | Madrid | Mayor | 28014 |

**Gráfico 15. Estructura de la tabla principal**

Como vemos el campo cv no está en la tabla principal.

| ID_TELEFONO | ID_NIF | CV |
|---|---|---|
| 616111222 | 22222221-A | Periodista, licenciado en Complutense ..... |
| 616111111 | 11111111-A | Arquitecto, licenciado en Complutense ..... |

**Gráfico 16. Estructura de la tabla secundaria**

Este campo está en la tabla secundaria junto a los dos campos que forman parte de la PK y que sirven para unir ambas tablas.

## @ENUMERATED

En Java utilizamos los tipos enumerados. Este tipo de datos no lo reconoce la base de datos, por lo cual, debemos utilizar esta anotación para indicar como persistir una propiedad de tipo enumerado. Lo podemos hacer de dos formas:

- Almacenando su índice; EnumType.ORDINAL
- Almacenando su valor; EnumType.STRING

```
public enum EstadoCivil {
    SOLTERO, CASADO, VIUDO, DIVORCIADO
}
```

**Gráfico 17. Declaración del tipo enumerado Estado Civil**

```
@Enumerated(EnumType.STRING)
private EstadoCivil estado;
```

**Gráfico 18. Marcamos la propiedad estado para que almacene su valor**

| ID_TELEFONO | ID_NIF | ESTADO |
|---|---|---|
| 616111222 | 22222221-A | CASADO |
| 616111111 | 11111111-A | CASADO |

**Gráfico 19. El campo estado almacena el valor del tipo enumerado**

## @LOB

Designa la propiedad de un campo como:

- CLOB (Character Large Object); se utiliza para introducir un campo de texto muy grande. En nuestro ejemplo lo utilizamos para el curriculum de la persona.
- BLOB (Binary Large Object); permite almacenar archivos binarios como por ejemplo una imagen.

```
@Lob
@Basic(fetch = FetchType.LAZY)
@Column(table = "Ejemplo1_CV")
private String cv;
```

**Gráfico 20. Ejemplo @Lob**

## @BASIC

Cuando utilizamos la anotación anterior es recomendable especificar el tipo de recuperación. Esto se denomina el tipo fetch.

En nuestro ejemplo cuando queremos leer una entidad Persona de la BBDD se recuperarán todos los datos referentes a ella.

Ahora bien, es necesario recuperar el curriculum de la persona cuando quizás solo hemos recuperado leído la entidad para conocer su dirección?

Para este fin utilizamos la anotación @Basic, mediante la cual especificamos si recuperamos automáticamente los datos o no.

- FetchType.LAZY; Con este tipo fetch estamos indicando que al recuperar los datos de la entidad, el campo marcado de esta forma no se recupera hasta que no se solicite específicamente.
- FetchType.EAGER; Este otro tipo indica que los datos referentes a este campo se recuperan de forma implícita.

```
@Lob
@Basic(fetch = FetchType.LAZY)
@Column(table = "Ejemplo1_CV")
private String cv;
```

**Gráfico 21. El campo cv no se recupera de la tabla hasta que no se especifique.**

## @TEMPORAL

Utilizamos esta anotación para indicar como queremos persistir los objetos de tipo fecha. Puede adoptar las siguientes constantes:

- TemporalType.DATE; almacena la fecha como día, mes y año.
- TemporalType.TIME; se almacena la hora como horas, minutos y segundos.
- TemporalType.TIMESTAMP; almacena fecha y hora juntas.

```
@Temporal(TemporalType.DATE)
private Date fechaNacimiento;
```

**Gráfico 22. En el campo fechaNacimiento se almacenará únicamente día, mes y año.**

# PERSISTENCE.XML

En este archivo vamos a definir las unidades de persistencia. Se necesita una unidad de persistencia por cada base de datos.

También se necesitan distintas unidades de persistencia si elegimos diferentes estrategias de generación de tablas. Por ejemplo: si elegimos eliminar y crear la tabla para persistir datos y elegimos ninguna para efectuar lecturas.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
             http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">

  <persistence-unit name="PU" transaction-type="RESOURCE_LOCAL">
    <provider>oracle.toplink.essentials.PersistenceProvider</provider>
    <class>jpaentidades.Persona</class>
    <properties>
      <property name="toplink.jdbc.user" value="app"/>
      <property name="toplink.jdbc.password" value="app"/>
      <property name="toplink.jdbc.url" value="jdbc:derby://localhost:1527/sample"/>
      <property name="toplink.jdbc.driver" value="org.apache.derby.jdbc.ClientDriver"/>
      <property name="toplink.ddl-generation" value="drop-and-create-tables"/>
    </properties>
  </persistence-unit>

  <persistence-unit name="PU2" transaction-type="RESOURCE_LOCAL">
    <provider>oracle.toplink.essentials.PersistenceProvider</provider>
    <class>jpaentidades.Persona</class>
    <properties>
      <property name="toplink.jdbc.user" value="app"/>
      <property name="toplink.jdbc.password" value="app"/>
      <property name="toplink.jdbc.url" value="jdbc:derby://localhost:1527/sample"/>
      <property name="toplink.jdbc.driver" value="org.apache.derby.jdbc.ClientDriver"/>
    </properties>
  </persistence-unit>
</persistence>
```

**Gráfico 23. persistence.xml**

Cada unidad de persistencia ha de contener los siguientes datos:

- **name**; nombre de la unidad de persistencia. Necesitamos un nombre para luego poder generar el objeto EntityManagerFactory del cual hablaremos más adelante.
- **transaction-type**; especificamos el tipo de transaccionalidad elegida.
- **provider**; necesitamos de un proveedor de persistencia. JPA actúa igual que JDBC, esto es, necesitamos una implementación del API para poder trabajar con la BBDD. En JDBC necesitábamos el driver de la BBDD y en JPA necesitamos de un proveedor de persistencia. En este ejemplo hemos optado por elegir TopLink.
- **class**; aquí deberán especificarse todas las entidades que vamos a manejar. En nuestro ejemplo detallamos la entidad Persona.

- **properties**; Como propiedades introducimos los datos necesarios para poder establecer una conexión a la base de datos. También como propiedad elegimos la estrategia de generación de tablas, pudiendo tomar estos valores:
    - **create-tables**; Intentará crear las tablas para cada operación.
    - **drop-and-create-tables**; Eliminará y creará las tablas de nuevo.
    - **Ninguno**; Ni elimina, ni crea tablas. Si optamos por este valor no aparece la propiedad en la unidad de persistencia.

## ENTITYMANAGER

El EntityManager es el puente entre el mundo orientado a objetos y bases de datos relacionales. El EntityManager realiza las operaciones CRUD (crear, leer, actualizar y eliminar) sobre las entidades.

Además, el EntityManager también trata de mantener las entidades sincronizadas con la base de datos automáticamente.

Podriamos decir que EntityManager es el objeto clave en JPA.

Para obtener el EntityManager necesitamos previamente obtener un objeto de tipo EntityManagerFactory. Este se creará a partir del nombre de la unidad de persistencia facilitado como argumento.

Una vez creado el EntityManager podemos afirmar que tenemos una conexión abierta a la base de datos.

```
// 1.- EntityManagerFactory
EntityManagerFactory emf =
        Persistence.createEntityManagerFactory("PU");

// 2.- EntityManager es el eje central de JPA
EntityManager em = emf.createEntityManager();

// 3.- Obtener una transaccion
EntityTransaction et = em.getTransaction();
```

**Gráfico 24. Obtener EntityManager**

Dependiendo de la operación que vayamos a realizar necesitamos una transacción o no. En el caso de las operaciones de persistencia es obligatorio obtener una, sin embargo en las operaciones de lectura no es necesario.

Mediante el objeto EntityTransaction podemos obtener una transacción a partir del EntityManager.

En la siguiente figura vemos como se inicia la transacción mediante el método begin(). Con el método commit() damos por válida la transacción y con el método rollback() deshacemos la transacción.

El método persist() de la interface EntityManager realiza la operación de persistencia en la tabla.

En el bloque finally cerramos el objeto EntityManager, donde implícitamente estamos cerrando la conexión abierta a la base de datos.

```
try{
     et.begin();

     em.persist(p1);
     em.persist(p2);
     em.persist(p3);

     et.commit();

}catch(Exception ex){
     et.rollback();
     ex.printStackTrace();
}finally{
     em.close();
}
```

**Gráfico 25. Manejo de transacciones**

El código de este ejemplo lo encontrareis en **Ejemplo1_JPA_Entidades**

## MODELO DE DOMINIO

Que ocurre cuando queremos trabajar con varias entidades a la vez que están relacionadas entre ellas. Por ejemplo como podríamos hacer para persistir todos los datos a la vez o como recuperar los datos, ...etc. Lo primero que deberíamos desarrollar es un modelo de dominio.

Un modelo de dominio es la representación de las relaciones y cardinalidad de todas las entidades relacionadas.

Veamos un ejemplo de modelo de dominio:

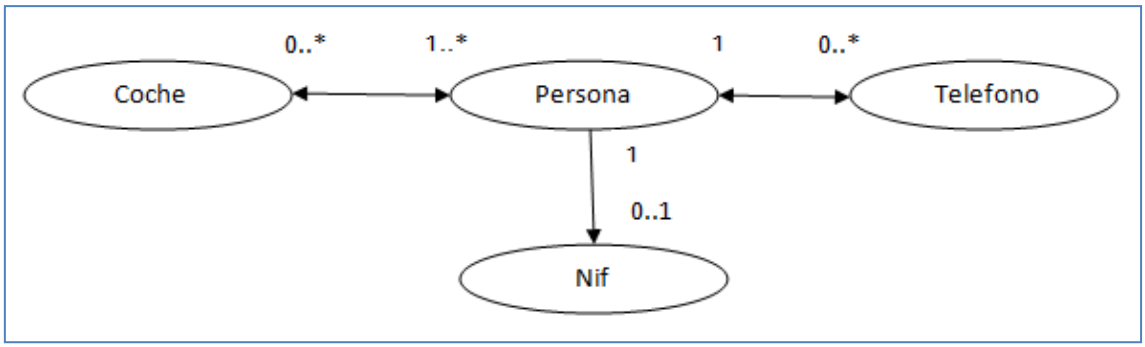


**Gráfico 26. Ejemplo modelo de dominio**

Cada uno de los círculos representa una entidad.

Las flechas indican la direccionalidad, esta puede ser de dos formas:

- **Unidireccional**; En el caso del Nif podemos decir que desde la entidad Persona se puede acceder a la entidad Nif pero no a la inversa
- **Bidireccional**; Según el ejemplo, desde Persona puedo acceder a la entidad Coche y a la inversa también.

Los números posicionados sobre las entidades indican la cardinalidad, estas pueden ser:

- **De uno a uno**; Una persona puede tener un solo nif.
- **De uno a varios**; Una persona puede tener ninguno o varios teléfonos.
- **De varios a uno**; Varios teléfonos pueden pertenecer a la misma persona.
- **De varios a varios**; Varios coches pueden pertenecer a varias personas a la vez, es decir, pueden tener varios propietarios.

## RELACION ENTRE ENTIDADES

Para establecer las relaciones entre entidades utilizamos anotaciones como:

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

### @ONETOONE

La entidad propietaria es la que consideramos más importante, en nuestro ejemplo será la entidad Persona.

La propiedad nif es la que anotamos como @OneToOne y definimos el tipo ALL de cascada para especificar que todas las operaciones que se hagan sobre la entidad Persona también se propagarán a la entidad Nif.

Con la anotación @JoinColumn especificamos la FK (Foreing Key). Se creará un campo llamado nif_id en la tabla de la entidad Persona con el valor del id de la tabla de Nif.

```
@OneToOne(cascade=CascadeType.ALL)//cascade en entidad propietaria
@JoinColumn(name="nif_id",referencedColumnName="id")
private Nif nif;
```

**Gráfico 27. Ejemplo OneToOne en la entidad propietaria.**

En este caso la entidad Nif actúa como subordinada. Aquí se especifica el atributo mappedBy para referenciar la propiedad nif de la entidad Persona.

```
@OneToOne(mappedBy="nif") // igual que en la entidad Persona
private Persona p;
```

**Gráfico 28. Mapeo OneToOne en la entidad subordinada**

## @ONETOMANY Y @MANYTOONE

La entidad que tiene la anotación @OneToMany es la entidad subordinada. En este caso estamos indicando que una persona puede tener varios teléfonos.

Para mantener sincronizadas ambas entidades utilizamos los métodos addXXX().

```
// quien tiene oneToMany es la entidad subordinada
@OneToMany(mappedBy="p",cascade=CascadeType.ALL)
private Set<Telefono> telefonos = new HashSet<Telefono>();

// metodos de sincronizacion
// uno por cada propiedad de tipo collection
public void addTelefono(Telefono t){
    telefonos.add(t);
    t.setP(this);
}
```

**Gráfico 29. Relación de uno a varios en la entidad subordinada**

El propietario es la entidad que contiene la anotación @ManyToOne.

```
@ManyToOne // la entidad principal es la que contiene ManyToOne
@JoinColumn(name="persona_id",referencedColumnName="id")
private Persona p;
```

**Gráfico 30. Relación de varios a uno en la entidad propietaria**

## @MANYTOMANY

Esta anotación permite anotar relaciones muchos a muchos.

Para poder configurar esta relación es necesario crear una tabla intermedia mediante la anotación @JoinTable. La tabla se creará con el id de persona y coche.

También son necesarios los métodos de sincronización.

```
@ManyToMany(cascade={CascadeType.MERGE,
                     CascadeType.PERSIST,
                     CascadeType.REFRESH})
@JoinTable(name="ejemplo2_personas_coches",
     joinColumns=@JoinColumn(name="persona_id",
                             referencedColumnName="id"),
     inverseJoinColumns=@JoinColumn(name="coche_id",
                             referencedColumnName="id"))
private Set<Coche> coches = new HashSet<Coche>();

public void addCoche(Coche c){
    coches.add(c);
    c.getPropietarios().add(this);
}
```

**Gráfico 31. Anotación @ManyToMany en la entidad Persona**

```
@ManyToMany(mappedBy="coches",
            cascade={CascadeType.MERGE,
                     CascadeType.PERSIST,
                     CascadeType.REFRESH})
private Set<Persona> propietarios = new HashSet<Persona>();

// metodo de sincronizacion
public void addPropietario(Persona p){
    propietarios.add(p);
    p.getCoches().add(this);
}
```

**Gráfico 32. Anotación @ManyToMany en la entidad Coche**

## OPERACIONES EN CASCADA

La siguiente tabla reúne todas las operaciones que se pueden especificar en cascada.

| Tipo de cascada | Efecto |
|---|---|
| CascadeType.MERGE | Solo las operaciones EntityManager.merge serán propagadas a las entidades relacionadas. |
| CascadeType.PERSIST | Solo las operaciones EntityManager.persist serán |

| | |
|---|---|
| | propagadas a las entidades relacionadas. |
| CascadeType.REFRESH | Solo las operaciones EntityManager.refresh serán propagadas a las entidades relacionadas. |
| CascadeType.REMOVE | Solo las operaciones EntityManager.remove serán propagadas a las entidades relacionadas. |
| CascadeType.ALL | Todas las operaciones EntityManager serán propagadas a las entidades relacionadas. |

El código de este ejemplo lo encontrareis en **Ejemplo2_JPA_Relaciones**

# ESTRATEGIAS DEL MAPEO DE HERENCIA

Podemos crear una entidad extendiendo de otra y reutilizando sus propiedades.

Lo vemos con un ejemplo, la entidad Persona declara tres propiedades: ID, NOMBRE y APELLIDO.

Utilizando herencia creamos otras dos entidades Profesor y Alumno. En la primera de ellas añadimos la propiedad TITULACION y en la segunda creamos la nueva propiedad CURSO.

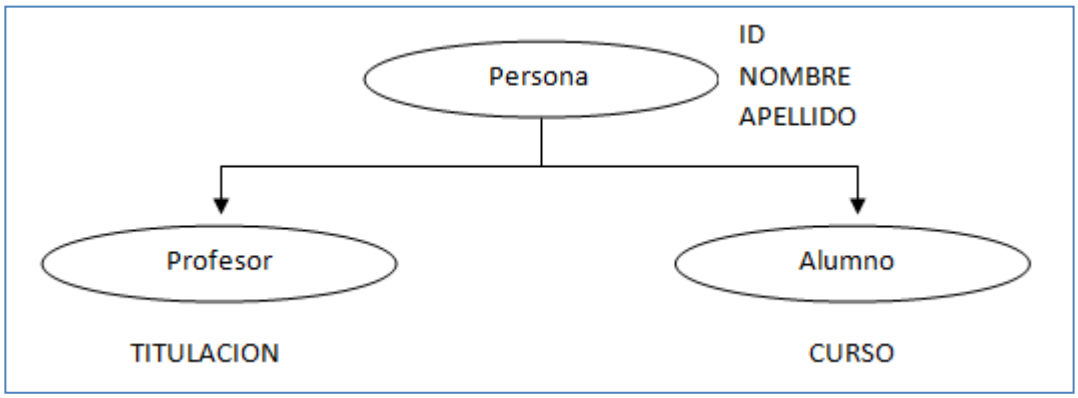


**Gráfico 33. Herencia entre entidades**

Para mapear la herencia utilizamos tres estrategias:

- SINGLE_TABLE
- JOINED
- TABLE_PER_CLASS

## SINGLE_TABLE

Se recogen en una sola tabla los datos de la entidad superclase y las entidades superclase.

El campo TIPO se conoce con el nombre de discriminador y lo utilizamos para especificar si el registro es un Alumno (A) o un Profesor (P).

| ID | TIPO | NOMBRE | APELLIDO | CURSO | TITULACION |
|---|---|---|---|---|---|
| 3 | A | Ana | Lopez | Spring | *<NULL>* |
| 2 | P | Jose | Sanchez | *<NULL>* | Ingeniero |

**Gráfico 34. Tabla generada con estrategia Single_Table**

La ventaja de esta estrategia es que al tener una sola tabla el rendimiento es muy bueno.

El inconveniente que presenta es que no utilizo todos los campos de la tabla por lo cual se generan muchos valores nulos.

Veamos que código necesitamos para implementar esta estrategia. Comenzamos por la entidad Persona:

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="TIPO", discriminatorType=DiscriminatorType.STRING,
                     length=1)
@Table(name="ejemplo4_personas")
public class Persona implements Serializable {
```

**Gráfico 35. Declaración de la entidad que actúa de superclase.**

Utilizamos la anotación @Inheritance para elegir la estrategia del mapeo de herencia, en nuestro ejemplo SINGLE_TABLE

Con la anotación @DiscriminatorColumn creamos el campo discriminador con nombre TIPO de tipo String y un solo carácter como longitud.

Proseguimos viendo el código de la entidad sublclase Alumno:

```
@Entity
@DiscriminatorValue(value="A")
public class Alumno extends Persona implements Serializable {
```

**Gráfico 36. Declaración de la entidad Alumno que actúa de subclase.**

La anotación @DiscriminatorValue permite especificar el valor del campo discriminador.

Por último vemos el código de la entidad Profesor:

```
@Entity
@DiscriminatorValue(value="P")
public class Profesor extends Persona implements Serializable {
```

**Gráfico 37. Declaración de la entidad Profesor que actúa de subclase.**

El código de este ejemplo lo encontrareis en **Ejemplo4_JPA_SingleTable**

## JOINED

El modelo Joined es el más utilizado ya que se asemeja al modelo relacional utilizado en bases de datos. Sigue siendo necesario utilizar el campo discriminador.

Las ventajas de este modelo son que no se generan valores nulos y presenta un mejor diseño.

El inconveniente es que ahora tendremos tres tablas por lo que habrá un peor rendimiento.

La estructura de tablas generada es la siguiente:

| ID | TIPO | NOMBRE | APELLIDO |
|---|---|---|---|
| 3 | A | Ana | Lopez |
| 2 | P | Jose | Sanchez |

**Gráfico 38. Tabla que recoge las entidades de tipo Persona**

| ID | CURSO |
|---|---|
| 3 | Spring |

**Gráfico 39. Tabla que recoge las entidades de tipo Alumno**

| ID | TITULACION |
|---|---|
| 2 | Ingeniero |

**Gráfico 40. Tabla que recoge las entidades de tipo Profesor**

El código para implementar esta estrategia lo detallamos a continuación comenzando por la entidad Persona:

Esta vez utilizamos la anotación @Inheritance para elegir la estrategia Joined

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
@DiscriminatorColumn(name="TIPO", discriminatorType=DiscriminatorType.STRING,
                     length=1)
@Table(name="ejemplo5_personas")
public class Persona implements Serializable {
```

**Gráfico 41. Declaración de la entidad Persona**

En las entidades que actúan como subclases es necesario establecer el valor del campo discriminador y también el nombre de la tabla cosa que no hacíamos en el modelo anterior.

Veamos el código de la entidad Alumno.

```
@Entity
@DiscriminatorValue(value="A")
@Table(name="ejemplo5_alumnos")
public class Alumno extends Persona implements Serializable {
```

**Gráfico 42. Declaración de la entidad Alumno**

Por último, vemos el código de la entidad Profesor.

```
@Entity
@DiscriminatorValue(value="P")
@Table(name="ejemplo5_profesores")
public class Profesor extends Persona implements Serializable {
```

**Gráfico 43. Declaración de la entidad Profesor**

El código de este ejemplo lo encontrareis en **Ejemplo5_JPA_Joined**

## TABLE_PER_CLASS

No todos los ORM implementan esta estrategia, por ejemplo TopLink no lo hace. Se genera una tabla por entidad.

No se reconocen ventajas de esta implementación a no ser que se quiera trabajar con tablas aisladas.

Los inconvenientes son varios:

- Peor rendimiento al tener varias tablas
- No tenemos un buen diseño
- No existe relación entre ninguna de las tablas
- No se aplican los conceptos de herencia en orientación a objetos

- Se duplican campos (Profesores y Alumnos duplican los campos de Persona)

En esta estrategia no se necesita un discriminador como en las otras anteriores.

La estructura de tablas generada será:

| ID | NOMBRE | APELLIDO |
|---|---|---|
| 3 | Ana | Lopez |
| 2 | Jose | Sanchez |

**Gráfico 44. Estructura de tabla para las entidades Persona**

| ID | NOMBRE | APELLIDO | CURSO |
|---|---|---|---|
| 3 | Ana | Lopez | Spring |

**Gráfico 45. Estructura de tabla para las entidades Alumno**

| ID | NOMBRE | APELLIDO | TITULACION |
|---|---|---|---|
| 2 | Jose | Sanchez | Arquitecto |

**Gráfico 46. Estructura de tabla para las entidades Profesor**

Este es el único código que debemos añadir en la entidad Persona:

```
@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
@Table(name="ejemplo6_personas")
public class Persona implements Serializable {
```

**Gráfico 47. Declaración de la entidad Persona**

```
@Entity
@Table(name="ejemplo6_alumnos")
public class Alumno extends Persona implements Serializable {
```

**Gráfico 48. Declaración de la entidad Alumno**

```
@Entity
@Table(name="ejemplo6_profesores")
public class Profesor extends Persona implements Serializable {
```

**Gráfico 49. Declaración de la entidad Profesor**

El código de este ejemplo lo encontrareis en **Ejemplo3_JPA_TablePerClass**

## LISTENERS DE CICLO DE VIDA

Un listener es un escuchador que detecta un evento. En este caso, el evento será la ejecución de un método del ciclo de vida. La siguiente tabla recoge los métodos de ciclo de vida.

| Métodos de ciclo de vida | Cuando se ejecutan |
|---|---|
| @PrePersist | Antes de que el EntityManager persista la entidad |
| @PostPersist | Después de que la entidad sea persistida |
| @PostLoad | Después de que la entidad sea cargada por una query, búsqueda o operación de actualización. |
| @PreUpdate | Antes de que se actualice la entidad en la base de datos. |
| @PostUpdate | Después de que se actualice la entidad en la base de datos. |
| @PreRemove | Antes de que el EntityManager elimine una entidad |
| @PostRemove | Después de que la entidad sea eliminada |

Vamos a generar un listener que detecte los eventos PrePersist y PreUpdate. Para ello generamos una clase con un método que reciba como argumento la instancia de la entidad a manejar. Este método los anotaremos con las anotaciones @ PrePersist y @ PreUpdate

Lo que hacemos con este listener es cambiar el nombre de cualquier persona a Pepito y luego mostrar un mensaje dependiendo si la entidad es de tipo Profesor o Alumno.

```
public class MiListener {

    @PrePersist
    @PreUpdate
    public void interceptar(Persona p){
        p.setNombre("Pepito");
        if (p instanceof Profesor){
            System.out.println("Se va a crear un nuevo registro profesor");
        }else{
            System.out.println("Se va a crear un nuevo registro de alumno");
        }
    }
}
```

**Gráfico 50. Creación de la clase que actúa como listener**

Con la anotación @EntityListeners asociamos el listener creado.

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="TIPO", discriminatorType=DiscriminatorType.STRING,
                     length=1)
@Table(name="ejemplo6_personas")
@EntityListeners(MiListener.class)
public class Persona implements Serializable {
```

**Gráfico 51. Asociamos el listener creado a la entidad Persona**

El código de este ejemplo lo encontrareis en **Ejemplo6_JPA_Listeners**

# JPQL

## TIPOS DE CONSULTAS

| Tipo de Consulta | Descripción |
|---|---|
| SELECT | Devuelve datos de una entidad o entidades relacionadas |
| UPDATE | Modifica una o varias entidades |
| DELETE | Elimina una o varias entidades |

## SELECT

La sintaxis es la siguiente:

```
SELECT [DISTINCT] expression1, expression2, .... expressionN

SELECT c
FROM Category c
WHERE c.categoryName LIKE :categoryName
ORDER BY c.categoryId
```

**Gráfico 52. Sintaxis y ejemplo de consultas SELECT**

En el ejemplo anterior, se seleccionan las entidades Category cuyo nombre sea igual al parámetro categoryName y ordenadas por su Id

En el siguiente ejemplo, se seleccionan todas las entidades de tipo Category.

```
SELECT c
FROM Category AS c

SELECT c.categoryName, c.createdBy
FROM Category c
```

**Gráfico 53. Ejemplos de consultas SELECT**

En el ejemplo anterior se seleccionan unicamente los campos categoryName y createdBy de las entidades de tipo Category.

## UPDATE

La sintaxis es la siguiente:

```
UPDATE entityName indentifierVariable
SET    single_value_path_expression1 = value1, ...
       single_value_path_expressionN = valueN
WHERE where_clause
```

**Gráfico 54. Sintaxis consulta de tipo UPDATE**

En el siguiente ejemplo se modifican las entidades de tipo Seller cuyo campo lastName contenga valores que comiencen por el prefijo 'PackRat' estableciendo sus campos status y commissionRate a valores 'G' y 10 respectivamente.

```
UPDATE Seller s
SET s.status = 'G', s.commissionRate = 10
WHERE s.lastName like 'PackRat%'
```

**Gráfico 55. Ejemplo consulta UPDATE**

## DELETE

La sintaxis es la siguiente:

```
DELETE entityName indentifierVariable
WHERE where_clause
```

**Gráfico 56. Sintaxis consulta de tipo DELETE**

```
DELETE Seller s
WHERE s.status = 'Silver'
```

**Gráfico 57. Ejemplo consulta DELETE**

En el ejemplo anterior se eliminarán todas las entidades de tipo Seller cuyo campo status contenga el valor Silver.

## CLAUSULA FROM

Define el dominio de la consulta, es decir, los nombres para las entidades que van a ser utilizados en la consulta. El operador AS es opcional.

La sintaxis es la siguiente:

```
FROM entityName [AS] identificationVariable
```

**Gráfico 58. Sintaxis de cláusula FROM**

## EXPRESAR RUTAS

Utilizamos el punto (.) como operador de navegación para expresar rutas.

Al igual que ocurre en el lenguaje Java el punto nos permite especificar una propiedad dentro del objeto, pues en JPA lo utilizamos para concretar un campo dentro de la entidad.

```
SELECT distinct c
FROM Category c
WHERE c.items is NOT EMPTY
```

**Gráfico 59. Ejemplo de operador de navegación**

En el ejemplo anterior se seleccionan todas las entidades diferentes de tipo Category cuyo campo items no este vacío.

## FILTRANDO DATOS CON WHERE

La cláusula WHERE le permite filtrar los resultados de una consulta. Únicamente las entidades que responden a la condición de consulta especificada se recuperarán de la base de datos.

En el siguiente ejemplo se seleccionan todas las entidades de tipo Category cuyo campo categoryId es mayor de 500.

```
SELECT c
FROM Category c
WHERE c.categoryId > 500
```

**Gráfico 60. Ejemplo cláusula WHERE**

## EXPRESIONES CONDICIONALES Y OPERADORES

| **Tipo de Operador** | **Operador** |
|---|---|
| Navegación | . |

| | |
|---|---|
| Signo unario | +,- |
| Aritméticos | *,/, +,- |
| Relacionales | =, >, >=, <, <=, <>,<br>[NOT] BETWEEN, [NOT] LIKE,<br>[NOT] IN, IS [NOT] NULL, IS [NOT] EMPTY, [NOT] MEMBER [OF] |
| Lógicos | NOT, AND, OR |

```
WHERE c.categoryName = 'Dumped Cars'
OR c.categoryName = 'Furniture from Garbage'
```

**Gráfico 61. Ejemplo de operadores**

En el ejemplo anterior se establece como condición que el campo categoryName sea igual a 'Dumped Cars' o 'Furniture from Garbage'.

## BETWEEN

Se puede utilizar el operador BETWEEN en una expresión aritmética para comparar una variable con un rango de valores.

Su sintaxis es la siguiente:

```
path_expression [NOT] BETWEEN lowerRange and upperRange
```

**Gráfico 62. Sintaxis cláusula BETWEEN**

Puede utilizar una cláusula WHERE y parámetros con nombre para el rango de la siguiente manera:

```
WHERE c.categoryId BETWEEN :lowRange AND :highRange
```

**Gráfico 63. Ejemplo cláusula BETWEEN**

## IN

El operador IN le permite crear una expresión condicional en función de si una expresión de ruta existe en una lista de valores. A continuación se muestra la sintaxis para el operador IN:

```
path_expression [NOT] IN (List_of_values)
```

**Gráfico 64. Sintaxis cláusula IN**

La lista de valores puede ser una lista estática de valores separados por comas, o una lista dinámica recuperada por una subconsulta.

```
WHERE u.userId IN ('viper', 'drdba', 'dumpster')
```

**Gráfico 65. Ejemplo cláusula IN**

Una subconsulta es una consulta dentro de otra consulta. Una subconsulta puede devolver un valor único o múltiple:

```
WHERE c.user IN (SELECT u
                 FROM User u
                 WHERE u.userType = 'A')
```

**Gráfico 66. Ejemplo de subconsultas con cláusula IN**

## LIKE

El operador LIKE permite determinar si una expresión de ruta de un solo valor coincide con un patrón de cadena. La sintaxis para el operador LIKE es la siguiente:

```
string_value_path_expression [NOT] LIKE pattern_value_
```

**Gráfico 67. Sintaxis de la cláusula LIKE**

Aquí pattern_value es una cadena literal o un parámetro de entrada. El pattern_value puede contener un guión bajo (_) o un signo de porcentaje (%). El guión bajo representa un solo carácter. El signo de porcentaje (%) representa cualquier número de caracteres.

```
WHERE c.itemName LIKE '_ike'
WHERE c.categoryName LIKE 'Recycle%'
WHERE c.categoryName NOT LIKE '%Recycle%'
WHERE c.categoryName NOT LIKE ?1
WHERE c.parentCategory IS NOT NULL
WHERE c.items IS EMPTY (only path expression collection)
```

**Gráfico 68. Ejemplos de la cláusula LIKE**

## MEMBER OF

Se puede utilizar el operador MEMBER OF para probar si una variable de identificación, una expresión de ruta de un solo valor o parámetro de entrada existe en una expresión de ruta de colección de valor.

A continuación se muestra la sintaxis para el operador MEMBER OF:

```
entity_expression [NOT] MEMBER [OF] collection_value_path_expression
```

**Gráfico 69. Sintaxis de la cláusula MEMBER OF**

Las palabras clave OF y NOT son opcionales y pueden omitirse. He aquí un ejemplo del uso de un parámetro de entrada con MEMBER OF:

```
WHERE :item MEMBER OF c.items
```

**Gráfico 70. Ejemplo MEMBER OF**

# FUNCIONES JPQL

Estas funciones se pueden utilizar ya sea en la cláusula WHERE o HAVING de una instrucción JPQL.

- **Funciones String**

| Función | Descripción |
|---|---|
| CONCAT(string1, string2) | Devuelve el valor de concatenar dos literales |
| SUBSTRING(string, position, length) | Devuelve la subcadena comenzando en la posición y según la longitud especificada |
| TRIM([LEADING \| TRAILING \| BOTH] [trim_character] FROM] string_to_trimmed) | Recorta el carácter especificado a una nueva longitud. El recorte puede ser iniciales, finales o de ambos extremos. |
| LOWER(string) | Devuelve la cadena convertida a minúsculas |
| UPPER(string) | Devuelve la cadena convertida a mayúsculas |
| LENGTH(string) | Devuelve la longitud de la cadena |
| LOCATE(searchString, stringToBeSearched[initialPosition]) | Devuelve la posición de una determinada cadena dentro de otra cadena. La búsqueda comienza en la posición 1 si initialPosition no se especifica. |

```
WHERE CONCAT(u.firstName, u.lastName) = 'ViperAdmin'
WHERE SUBSTRING(u.lastName, 1, 3) = 'VIP'
```

**Gráfico 71. Ejemplos de funciones String**

- **Funciones aritméticas**

| Funciones aritméticas | Descripción |
|---|---|
| ABS(simple_arithmetic_expression) | Devuelve el valor absoluto de simple_arithmetic_expression |
| SQRT(simple_arithmetic_expression) | Devuelve el valor de la raíz cuadrada de simple_arithmetic_expression como un doble |
| MOD(num, div) | Devuelve el resultado de ejecutar la operación de módulo para num, div |
| SIZE(collection_value_path_expression) | Devuelve el número de elementos de una colección |

```
WHERE SIZE(c.items) = 5
```

**Gráfico 72. Ejemplo de función aritmética**

- **Funciones temporales**

| Funciones temporales | Descripción |
|---|---|
| CURRENT_DATE | Devuelve la fecha actual |
| CURRENT_TIME | Devuelve la hora actual |
| CURRENT_TIMESTAMP | Devuelve fecha y hora actuales |

## USANDO AGREGACIONES

Las agregaciones son útiles al escribir consultas de informes que tienen que ver con un conjunto de entidades.

| Funciones de agregación | Descripción | Tipo devuelto |
|---|---|---|
| AVG | Devuelve el valor promedio de todos los valores del campo sobre el que se aplica | Double |

| | | |
|---|---|---|
| COUNT | Devuelve el número de resultados devueltos por la consulta | Long |
| MAX | Devuelve el valor máximo del campo sobre el que se aplica | Depende del tipo del campo persistido |
| MIN | Devuelve el valor mínimo del campo sobre el que se aplica | Depende del tipo del campo persistido |
| SUM | Devuelve la suma de todos los valores del campo sobre el que se aplica | Puede devolver Long o Double |

- **GROUP**

Esta consulta generará un informe que muestra el número de entidades creadas por cada categoría c.user:

```
SELECT c.user, COUNT(c.categoryId)
FROM Category c
GROUP BY c.user
```

**Gráfico 73. Ejemplo con la cláusula GROUP**

Hemos agrupado por una entidad asociada. Puede agrupar por una expresión de ruta de un solo valor que es o bien una persistencia o un campo de asociación. Sólo se permiten funciones agregadas al realizar la agregación utilizando GROUP BY.

- **HAVING**

Supongamos que desea recuperar sólo los usuarios que han creado más de cinco entidades categoría. Basta con modificar la consulta anterior de la siguiente manera:

```
SELECT c.user, COUNT(c.categoryId)
FROM Category c
GROUP BY c.user
HAVING COUNT(c.categoryId) > 5

SELECT c.user, COUNT(c.categoryId)
FROM Category c
WHERE c.createDate is BETWEEN :date1 and :date2
GROUP BY c.user
HAVING COUNT(c.categoryId) > 5
```

**Gráfico 74. Ejemplos cláusula HAVING**

En primer lugar, la cláusula WHERE se aplica para filtrar los resultados. A continuación, los resultados se agregan en función de la cláusula GROUP BY. Por último, la cláusula HAVING se aplica para filtrar el resultado agregado.

## ORDENAR LOS RESULTADOS DE LA QUERY

Puede controlar el orden de los valores y los objetos recuperados por una consulta mediante la cláusula ORDER BY:

```
ORDER BY path_expression1 [ASC | DESC], ... path_expressionN
[ASC | DESC]

SELECT c
FROM Category c
ORDER BY c.categoryName ASC

SELECT c.categoryName, c.createDate
FROM Category c
ORDER BY c.categoryName ASC, c.createDate DESC
```

**Gráfico 75. Ejemplos de la cláusula ORDER BY**

Si se utiliza un solo valor expresiones de ruta en lugar de una variable de identificación, la cláusula SELECT debe contener la expresión de ruta que se utiliza en la cláusula ORDER BY.

## UTILIZAR SUBCONSULTAS

Una subconsulta es una consulta dentro de una consulta. Se utiliza una subconsulta ya sea en una cláusula WHERE o HAVING para filtrar el conjunto de resultados.

```
[NOT] IN / [NOT] EXISTS / ALL / ANY / SOME (subquery)
```

**Gráfico 76. Cláusulas a utilizar en subconsultas**

La subconsulta se evalúa en primer lugar, y luego la consulta principal se recupera basándose en el resultado de la subconsulta.

```
SELECT i
FROM Item i
WHERE i.user IN (   SELECT c.user
                    FROM Category c
                    WHERE c.categoryName LIKE :name)
```

**Gráfico 77. Ejemplo subconsulta**

La misma consulta es:

```
SELECT i
FROM Item i
WHERE EXISTS (      SELECT c
                    FROM Category c
                    WHERE c.user = i.user)

SELECT c
FROM Category c
WHERE c.createDate >= ALL
             (SELECT i.createDate
             FROM Item i
             WHERE i.user = c.user)
```

**Gráfico 78. Ejemplo subconsulta**

La subconsulta devuelve true si todos los resultados obtenidos por la subconsulta cumplen la condición, de lo contrario, la expresión devuelve falso.

Si usamos ANY o SOME, la expresión devuelve verdadero si alguno de los resultados obtenidos cumple con la condición de consulta. Podemos usar ANY en una consulta de la siguiente manera:

```
SELECT c
FROM Category c
WHERE c.createDate >= ANY
             (SELECT i.createDate
             FROM Item i
             WHERE i.seller = c.user)
```

**Gráfico 79. Ejemplo subconsulta**

SOME es sólo un alias (o un sinónimo) para ANY, y se puede utilizar en cualquier lugar donde se puede utilizar.

## UNIR ENTIDADES

Puede usar JOIN para crear un producto cartesiano entre dos entidades.

Normalmente se proporciona una cláusula WHERE para especificar la condición de unión entre las entidades en lugar de sólo la creación de un producto cartesiano.

Tiene que especificar las entidades en la cláusula FROM para crear una unión entre dos o más entidades. Las dos entidades son unidas basadas ya sea en sus relaciones o cualquier campo persistencia arbitrarias.

Cuando dos entidades se unen, se puede decidir para recuperar los resultados que coinciden con las condiciones de combinación.

Por ejemplo, supongamos que nos unimos categoría y el elemento utilizando las relaciones entre ellos y recuperar las únicas entidades que responden a la condición de unión. Dicha suma se conocen como combinaciones internas.

A la inversa, supongamos que tenemos que recuperar los resultados que satisfagan las condiciones de combinación, sino también incluir entidades de un lado del dominio que no tienen las entidades correspondientes en el otro lado.

Por ejemplo, es posible que desee recuperar todas las instancias de la categoría, incluso si no hay ninguna instancia correspondiente del artículo.

Este tipo de unión se denomina una combinación externa. Tenga en cuenta que una unión externa puede ser de izquierda, derecha, o ambas cosas.

- **Inner joins**

Una situación más común en las aplicaciones es la necesidad de unir dos o más entidades sobre la base de sus relaciones. A continuación se muestra la sintaxis INNER JOIN para:

```
[INNER] JOIN join_association_path_expression [AS] identification_variable
```

**Gráfico 80. Sintaxis INNER JOIN**

```
SELECT u
FROM User u INNER JOIN u.Category c
WHERE u.userId LIKE ?1
```

**Gráfico 81. Ejemplo INNER JOIN**

- **Outer joins**

Las combinaciones externas le permiten recuperar entidades adicionales que no coinciden con las condiciones de unión cuando las asociaciones entre entidades son opcionales. Las combinaciones externas son particularmente útiles en la elaboración de informes.

Vamos a suponer que existe una relación entre el Usuario y opcional de categoría y queremos generar un informe que imprime todos los nombres de categoría para el usuario. Si el usuario no tiene ninguna categoría, entonces queremos imprimir NULL. Si se especifica el usuario en el lado izquierdo de la unión, se Puede utilizar el LEFT JOIN o LEFT OUTER JOIN frases de palabras clave con una consulta JPQL de la siguiente manera:

```
SELECT u
FROM User u LEFT OUTER JOIN u.Category c
WHERE u.userId like ?1
```

**Gráfico 82. Ejemplo OUTER JOIN**

- **Fetch joins**

En una aplicación comercial normal, es posible que desee consultar para una entidad en particular, sino también recuperar sus entidades asociadas al mismo tiempo. Podemos utilizar una fetch unirse en JPQL para recuperar una entidad asociada como un efecto secundario de la recuperación de una entidad:

```
SELECT b
FROM Bid b FETCH JOIN b.bidder
WHERE b.bidDate >= :bidDate
```

**Gráfico 83. . Ejemplo FETCH JOIN**

# INTEGRACION JPA CON SPRING

Los pasos para poder integrar JPA con Spring son los siguientes:

- Crear un EntityManagerFactory:

LocalEntityManagerFactoryBean (no J2EE)

  - LocalContainerEntityManagerFactoryBean (J2EE)
- Crear el adaptador si utilizamos un LocalContainerEntityManagerFactoryBean
- Crear el dao
- Crear el bean del dao o la detección automática.

### Paso 1. Crear un EntityManagerFactory

Este puede ser de dos tipos:

- LocalEntityManagerFactoryBean (no J2EE)
- LocalContainerEntityManagerFactoryBean (J2EE)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

       <bean id="emf"
             class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
           <property name="persistenceUnitName" value="PU" />
       </bean>
</beans>
```

**Gráfico 84. Creación del bean EntityManagerFactory con LocalEntityManagerFactoryBean**

Mostramos un ejemplo de cómo se crearía de la segunda forma.

```xml
<bean id="emf" class=
    "org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
  <property name="dataSource" ref="dataSource" />
  <property name="jpaVendorAdapter" ref="jpaVendorAdapter" />
</bean>
```

**Gráfico 85. Creación del bean EntityManagerFactory con LocalContainerEntityManagerFactoryBean**

## Paso 2. Crear el adaptador

En nuestro caso no necesitamos crear el adaptador puesto que utilizamos un LocalEntityManagerFactoryBean.

Mostramos un ejemplo de cómo se haría si además utilizásemos Hibernate como ORM.

```xml
<bean id="jpaVendorAdapter"
    class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
  <property name="database" value="HSQL" />
  <property name="showSql" value="true"/>
  <property name="generateDdl" value="false"/>
  <property name="databasePlatform"
            value="org.hibernate.dialect.HSQLDialect" />
</bean>
```

**Gráfico 86. Creación del adaptador**

## Paso 3. Crear el dao

A continuación mostramos un ejemplo de cómo desarrollaríamos la clase DAO.

```
@Repository("spitterDao")
@Transactional
public class JpaSpitterDao implements SpitterDao {
  private static final String RECENT_SPITTLES =
      "SELECT s FROM Spittle s";
  private static final String ALL_SPITTERS =
      "SELECT s FROM Spitter s";
  private static final String SPITTER_FOR_USERNAME =
      "SELECT s FROM Spitter s WHERE s.username = :username";
  private static final String SPITTLES_BY_USERNAME =
      "SELECT s FROM Spittle s WHERE s.spitter.username = :username";

  @PersistenceContext
  private EntityManager em; //<co id="co_injectEM"/>

  public void addSpitter(Spitter spitter) {
    em.persist(spitter); //<co id="co_useEM"/>
  }
```

**Gráfico 87. Creación de la clase DAO**

## Paso 4. Crear el bean del dao o la detección automática

```
<context:component-scan base-package="com.habuma.spitter.persistence" />
```

**Gráfico 88. Anotación que permite la detección automática de beans.**

Todo el código de este ejemplo lo encontrareis en **7. Spring JPA Entidades**

El código de este ejemplo lo encontrareis en **Ejemplo7_Spring_JPA_Entidades**

# 3. HIBERNATE

La alternativa a JPA es Hibernate. Este framework se ha convertido durante mucho tiempo en el más utilizado.

Si JPA utiliza el sistema de anotación para realizar mapeos y configuración en Hibernate se sigue utilizando XML. Aunque es cierto que las últimas versiones incluyen la posibilidad de sustituir el código XML por anotaciones en la gran mayoría de proyectos se trabaja de la primera forma.

Una vez que hemos visto JPA los conceptos son los mismos para Hibernate, solo que los reconoceremos con otros nombres. A continuación mostramos una tabla donde podemos ver los recursos comunes a ambos frameworks:

| **JPA** | **Hibernate** |
|---|---|
| EntityManagerFactory | SessionFactory |
| EntityManager | Session |
| EntityTransaction | Transaction |
| JPQL | HQL |
| Persistence.xml | Hibernate.cfg.xml |
| Anotaciones | Archivos de mapeo .xml |

## CREAR ENTIDADES

Con Hibernate nuestras entidades se convierten en POJO´s, esto es, las clases vuelven a ser simplemente clases Java simples sin anotaciones.

Las entidades en Hibernate pueden ser clases reutilizables ya que no tienen una dependencia con el framework.

En el siguiente ejemplo vemos un fragmento de la entidad Persona donde apreciamos los requisitos mínimos que deben cumplir las entidades. Estos son:

- Deben implementar la interface Serializable.
- Deben tener un constructor sin argumentos y que este sea público.
- Todas las propiedades han de tener métodos de acceso get() y set().

```
public class Persona implements Serializable{

    private PersonaPK pk;

    private String nombre;
    private Date fechaNacimiento;
    private EstadoCivil estado;
    private Direccion direccion;
    private String curriculum;

    public Persona() {
    }

    public String getCurriculum() {...}

    public void setCurriculum(String curriculum) {...}

    public Direccion getDireccion() {...}

    public void setDireccion(Direccion direccion) {...}
```

**Gráfico 89. Fragmento de la entidad Persona**

## MAPEAR ENTIDADES

Para mapear entidades utilizamos archivos de mapeo con formato XML. Cada entidad ha de tener su propio archivo de mapeo.

En la siguiente figura vemos la estructura de este archivo.

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE hibernate-mapping PUBLIC
  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>

  <class>

</hibernate-mapping>
```

**Gráfico 90. Archivo de mapeo**

Vamos a detallar el mapeo de cada propiedad con la configuración XML necesaria.

## ENTIDAD

Con el elemento <class> definimos la entidad que estamos mapeando. Los atributos utilizados son el nombre completo de la clase, tabla en la cual se almacenan las entidades y el esquema utilizado en la base de datos.

```
<class name="app.modelo.Persona" table="ejemplo1_personas" schema="APP">
```

**Gráfico 91. Mapeo de la clase entidad**

## CLAVES PRIMARIAS

- Clave primaria simple

Una clave primaria simple es aquella que está formada por un único campo. Para realizar el mapeo en este caso utilizamos el elemento <id> con los siguientes atributos. El nombre de la propiedad, la columna o campo donde se almacenará en la tabla, el tipo de dato y la longitud de dicho campo.

```
<!-- clave primaria de un solo campo  -->
    <id name="id" column="ID_PERSONA" type="string" length="5">
            <generator class="assigned" />
    </id>
```

**Gráfico 92. Mapeo de clave primaria simple**

Se puede utilizar el elemento <generator> para especificar una clase Java que se encargará de generar identificadores únicos.

Los posibles generadores son:

1. **increment**; genera identificadores de tipo long, short o int que solamente son únicos cuando ningún otro proceso está insertando datos en la misma tabla. No lo utilice en un clúster.
2. **identity**; soporta columnas de identidad en DB2, MySQL, MS SQL Server, Sybase y HypersonicSQL. El identificador devuelto es de tipo long, short o int.
3. **sequence**; usa una secuencia en DB2, PostgreSQL, Oracle, SAP DB, McKoi o un generador en Interbase. El identificador devuelto es de tipo long, short o int.
4. **hilo**; utiliza un algoritmo alto/bajo para generar eficientemente identificadores de tipo long, short o int, dada una tabla y columna como fuente de valores altos (por defecto hibernate_unique_key y next_hi respectivamente). El algoritmo alto/bajo genera identificadores que son únicos solamente para una base de datos particular.
5. **seqhilo**; utiliza un algoritmo alto/bajo para generar eficientemente identificadores de tipo long, short o int, dada una secuencia de base de datos.

6. **uuid**; utiliza un algoritmo UUID de 128 bits para generar identificadores de tipo cadena, únicos dentro de una red (se utiliza la dirección IP). El UUID se codifica como una cadena hexadecimal de 32 dígitos de largo.
7. **guid**; utiliza una cadena GUID generada por base de datos en MS SQL Server y MySQL.
8. **native**; selecciona identity, sequence o hilo dependiendo de las capacidades de la base de datos subyacente.
9. **assigned**; deja a la aplicación asignar un identificador al objeto antes de que se llame a save(). Esta es la estrategia por defecto si no se especifica un elemento <generator>.
10. **select**; recupera una clave principal asignada por un disparador de base de datos seleccionando la fila por alguna clave única y recuperando el valor de la clave principal.
11. **foreign**; utiliza el identificador de otro objeto asociado. Generalmente se usa en conjunto con una asociación de clave principal <one-to-one>.
12. **sequence-identity**; una estrategia de generación de secuencias especializadas que utiliza una secuencia de base de datos para el valor real de la generación, pero combina esto junto con JDBC3 getGeneratedKeys para devolver el valor del identificador generado como parte de la ejecución de la declaración de inserción. Esta estrategia está soportada solamente en los controladores 10g de Oracle destinados para JDK1.4. Los comentarios en estas declaraciones de inserción están desactivados debido a un error en los controladores de Oracle.

- Clave primaria compuesta

Una clave primaria compuesta es aquella que utiliza varios campos de la tabla. Utilizamos el elemento <composite-id> con atributos como el nombre de la propiedad y la clase.

```
<!--  Clave primaria compuesta. De varios campos  -->
<composite-id name="pk" class="app.modelo.PersonaPK">
    <key-property name="telefono" column="TELEFONO" type="long" />
    <key-property name="dni" column="DNI" type="string" />
</composite-id>
```

**Gráfico 93. Mapeo de clave primaria compuesta**

Los elementos <key-property> especifican los campos que utiliza la clave primaria compuesta con el nombre de la propiedad, el nombre del campo de la tabla y el tipo de datos.

## PROPIEDADES SIMPLES

En el siguiente ejemplo mapeamos una propiedad de tipo String, utilizando el elemento <property> con los siguientes atributos:

- name; nombre de la propiedad en la clase entidad

- column; columna o campo de la tabla donde se almacenará el valor
- type; tipo de datos
- not-null; el valor false indica que se pueden introducir valores nulos. Para no permitir valores nulos deberíamos establecer esta propiedad a true.
- unique; el valor false indica que los valores no tienen por qué ser únicos. Si queremos que sean valores únicos deberíamos establecer el valor true.
- length; longitud del campo.
- lazy; establecemos el modo de recuperación. Si ponemos false estamos indicando un modo de recuperación 'eager'.

```
<property name="nombre" column="NOMBRE" type="string"
          not-null="false" unique="false" length="20" lazy="false"/>
```

**Gráfico 94. Mapeo de la propiedad nombre**

En el siguiente ejemplo mapeamos una propiedad de tipo Date.

```
<property name="fechaNacimiento" column="FECHA_NACIMIENTO"
          type="date" />
```

**Gráfico 95. Mapeo de la propiedad de tipo Date**

## TIPOS ENUMERADOS

Para poder mapear un tipo de datos enumerado necesitamos la clase EnumUserType. Esta clase la podemos descargar de Internet.

```
public class EnumUserType implements EnhancedUserType, ParameterizedType {

   private Class<Enum> enumClass;

   public void setParameterValues(Properties parameters) {
      String enumClassName = parameters.getProperty("enumClassName");
      try {
         enumClass = (Class<Enum>) Class.forName(enumClassName);
      } catch (ClassNotFoundException cnfe) {
         throw new HibernateException("Enum class not found", cnfe);
      }
   }

   public Object assemble(Serializable cached, Object owner)
         throws HibernateException {
      return cached;
   }
```

**Gráfico 96. Fragmento de la clase EnumUserType**

En el siguiente ejemplo mapeamos un tipo enumerado. En el elemento <type> ponemos el nombre completo de la clase EnumUserType. Con el elemento <param> especificamos el nombre completo de nuestro tipo enumerado.

```
<property name="estado" column="ESTADO_CIVIL">
    <type name="utilidades.EnumUserType">
        <param name="enumClassName">
            app.modelo.EstadoCivil
        </param>
    </type>
</property>
```

**Gráfico 97. Mapeo de un tipo enumerado**

## CAMPOS EMBEBIDOS

En Hibernate las clases embebidas se conocen como componentes.

Según nuestro ejemplo si queremos que una propiedad de tipo Dirección se almacene con los campos: calle, localidad y código postal lo mapearemos con el elemento <component>.

Este elemento utiliza como atributos el nombre de la propiedad y la clase que queremos embeber.

Los elementos <property> detallan las propiedades de la clase Dirección y su mapeo contra la tabla.

```
<component name="direccion" class="app.modelo.Direccion">
    <property name="calle" column="CALLE" type="string"/>
    <property name="localidad" column="LOCALIDAD" type="string" />
    <property name="cp" column="CODIGO_POSTAL" type="integer" />
</component>
```

**Gráfico 98. Mapeo de un componente**

## MAPEOS DE CAMPOS A OTRA TABLA

Si queremos utilizar una tabla secundaria utilizamos el elemento <join> donde especificamos el nombre de la tabla y el esquema.

```
<join table="ejemplo1_curriculumns" schema="APP">
    <key>
        <column name="TELEFONO" />
        <column name="DNI" />
    </key>
    <property name="curriculum" column="CURRICULUM_VITAE"
              type="string"  length="1000" lazy="true"/>
</join>
```

**Gráfico 99. Ejemplo de uso de tablas secundarias**

Con el elemento anidado <key> pondremos los campos de la clave primaria que queremos replicar en la secundaria.

Con el elemento <property> pondremos la propiedad que queremos mapear a la tabla secundaria. En nuestro ejemplo enviamos el curriculum a otra tabla.

## HIBERNATE.CFG.XML

El archivo hibernate.cfg.xml es donde configuramos tanto las propiedades de conexión a la base de datos como las entidades que vamos a manejar.

En el elemento <session-factory> recogemos la configuración, según nuestro ejemplo contiene lo siguiente:

- Dialecto: Se precisa especificar el dialecto de la base de datos utilizada para que Hibernate pueda ajustar las queries a su formato adecuado.
- Propiedades de conexión: Se necesita el driver, url de la base de datos, usuario y password de la base de datos.
- Entidades; con el elemento <mapping> hacemos referencia a todas las entidades que estamos manejando.

También podemos incluir otras propiedades adicionales.

- show_sql; indicamos si queremos ver en la consola el código SQL generado.
- connection.pool_size; Número de conexiones abiertas a la vez.
- hbm2ddl.auto; Estrategia de generación de tablas. Admite como valores:
    - create; se crean las tablas

create-drop; se eliminan y crean las tablas

    - update; se actualiza la tabla con el nuevo formato
    - validate; no hace nada, es decir, ni crea, elimina ni modifica ninguna tabla.

- current_session_context_class; para permitir la conexión del campo de acción y el contexto de definición de las sesiones actuales
    - thread; las sesiones actuales son rastreadas por un hilo de ejecución
    - jta; una transacción JTA rastrea y asume las sesiones actuales
    - managed; usted es responsable de vincular y desvincular una instancia Session con métodos estáticos en esta clase: no abre, vacía o cierra una Session
- cache.provider_class; Especificamos sí que quieren cachear las queries o no.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <property name="hibernate.dialect">
            org.hibernate.dialect.DerbyDialect
        </property>

        <property name="hibernate.connection.driver_class">
            org.apache.derby.jdbc.ClientDriver
        </property>
        <property name="hibernate.connection.url">
            jdbc:derby://localhost:1527/sample
        </property>
        <property name="hibernate.connection.username">
            app
        </property>
        <property name="hibernate.connection.password">
            app
        </property>

        <mapping resource="app/modelo/Persona.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

**Gráfico 100. Archivo hibernate.cfg.xml**

```
<property name="hibernate.show_sql">true</property>
<property name="hibernate.connection.pool_size">1</property>
<property name="hibernate.hbm2ddl.auto">create</property>
<property name="hibernate.current_session_context_class">thread</property>
<property name="hibernate.cache.provider_class">
    org.hibernate.cache.NoCacheProvider
</property>
```

**Gráfico 101. Propiedades adicionales**

## SESSION Y SESSION FACTORY

Una vez creadas las entidades y debidamente mapeadas procedemos a ver cómo podemos persistirlas en la base de datos.

Lo primero que necesitamos es obtener un objeto de tipo SessionFactory de la siguiente forma:

```
SessionFactory sf =
        new Configuration().configure().buildSessionFactory();
```

**Gráfico 102. Objeto SessionFactory**

Una vez obtenida el SessionFactory creamos un objeto de tipo Session que será el encargado de abrir la conexión a la base de datos.

```
Session session = sf.openSession();
```

**Gráfico 103. Objeto Session**

Cuando tenemos una session podremos obtener la transacción, necesaria para persistir los datos.

```
Transaction tx = session.getTransaction();
```

**Gráfico 104. Objeto Transaction**

En el siguiente fragmento vemos como se inicia la transacción (begin), como se persiste la entidad (save), se da por válida la transacción (commit), si existe una excepción se deshace la transacción (rollback) y finalmente se cierran los recursos abiertos la Session y el SessionFactory.

```
try{
    tx.begin();

    session.save(p);

    tx.commit();
}catch(Exception ex){
    tx.rollback();
    ex.printStackTrace();
}finally{
    session.close();
    sf.close();
}
```

**Gráfico 105. Fragmento de persistencia de entidades**

El código de este ejemplo lo encontrareis en **Ejemplo1_Hibernate_Entidades**

# RELACION ENTRE ENTIDADES

Al igual que en JPA podemos utilizar varias entidades relacionadas entre sí. Sería necesario crear el modelo de dominio estableciendo la cardinalidad y la direccionalidad de la relación.

Vamos a implementar con Hibernate el mismo ejemplo de JPA.

## RELACIÓN ONETOONE

Recordamos que según el ejemplo una Persona puede tener un Nif y un Nif pertenece únicamente a una Persona. Por lo cual es una relación bidireccional uno a uno.

En el archivo de mapeo de la entidad Persona: Persona.hbm.xml mapeamos la propiedad nif de la entidad.

```
<!-- unique="true" esto es lo que marca one-to-one -->
<many-to-one unique="true" name="nif" column="NIF_ID" not-null="true"
             cascade="all"/>
```

**Gráfico 106. Mapeo de la entidad Persona**

En el archivo de mapeo de la entidad Nif: Nif.hbm.xml mapeamos la propiedad p de la entidad Nif y referenciando la propiedad nif que aparece en la entidad Persona.

```
<one-to-one name="p" class="app.modelo.Persona" property-ref="nif" />
```

**Gráfico 107. Mapeo de la entidad Nif**

## RELACIÓN ONETOMANY Y MANYTOONE

Según el ejemplo una Persona puede tener varios teléfonos y varios teléfonos pertenecer a la misma persona. Lo que se representa con una relación 'uno a varios' desde la entidad Persona y una relación 'varios a uno' en la entidad Telefono.

```
private Set<Telefono> telefonos = new HashSet<Telefono>();
```

**Gráfico 108. Conjunto de teléfonos en la entidad Persona**

Recordamos que las propiedades de tipo colección necesitan de un método de sincronización.

```
public void addTelefono(Telefono t){
    telefonos.add(t);
    t.setP(this);
}
```

**Gráfico 109. Método de sincronización en la entidad Persona**

Comenzamos viendo el archivo de mapeo de Persona: Persona.hbm.xml

Mapeamos la propiedad telefonos de la entidad Persona con el elemento <set>. Con el elemento <key> indicamos la FK (Foreing Key) esto es el campo ID_PERSONA de la tabla de Telefonos.

El elemento <one-to-many> nos permite indicar la relación uno a varios desde la entidad Persona a la entidad Telefono.

```
<set name="telefonos" cascade="all">
    <key column="ID_PERSONA" />  <!-- FK de Telefono -->
    <one-to-many class="app.modelo.Telefono" />
</set>
```

**Gráfico 110. Mapeo de la propiedad teléfonos con relación uno a muchos**

En el archivo de mapeo de la entidad Telefono: Telefono.hbm.xml mapeamos la propiedad p de tipo Persona como relación 'varios a uno'.

```
<many-to-one name="p" class="app.modelo.Persona"
             column="ID_PERSONA" not-null="true" />
```

**Gráfico 111. Mapeo de la propiedad p con relación muchos a uno**

## RELACIÓN MANYTOMANY

Las relaciones ManyToMany necesitan de una tabla intermedia. En el siguiente ejemplo vemos como mapear la propiedad coches de la entidad Persona como una relación 'muchos a muchos' con la entidad Coche.

```
private Set<Coche> coches = new HashSet<Coche>();
```

**Gráfico 112. Conjunto de coches en la entidad Persona**

```
private Set<Persona> propietarios = new HashSet<Persona>();
```

**Gráfico 113. Conjunto de personas en la entidad Coche**

Ponemos como atributo el nombre de la tabla intermedia y además indicamos que la columna ID_PERSONA debe aparecer en dicha tabla.

```
<set name="coches" table="ejemplo2_personas_coches" cascade="all">
    <key column="ID_PERSONA" />  <!-- columna nueva en la tabla intermedia -->
    <many-to-many column="ID_COCHE" class="app.modelo.Coche" />
</set>
```

**Gráfico 114. Mapeo de la propiedad coches con relación muchos a muchos**

Igualmente en el archivo de mapeo Coche.hbm.xml mapeamos la propiedad propietarios haciendo uso de la tabla intermedia y creando en ella el campo ID_COCHE.

```
<set name="propietarios" inverse="true"
     table="ejemplo2_personas_coches" cascade="all">
    <key column="ID_COCHE" />  <!-- columna nueva en la tabla intermedia -->
    <many-to-many column="ID_PERSONA" class="app.modelo.Persona" />
</set>
```

**Gráfico 115. Mapeo de la propiedad propietarios con relación muchos a muchos**

En las entidades Persona y Coche siguen siendo necesarios los métodos de sincronización.

```
public void addCoche(Coche c){
    coches.add(c);
    c.getPropietarios().add(this);
}
```

**Gráfico 116. Método de sincronización en la entidad Persona**

```
public void addPropietario(Persona p){
    propietarios.add(p);
    p.getCoches().add(this);
}
```

**Gráfico 117. Método de sincronización en la entidad Coche**

El código de este ejemplo lo encontrareis en **Ejemplo2_Hibernate_Relaciones**

# ESTRATEGIAS DEL MAPEO DE HERENCIA

Recordamos que existen tres estrategias del mapeo de herencia al igual que en JPA:

- SingleTable
- Joined
- TablePerClass

## SINGLETABLE

```
<class name="app.modelo.Persona" table="ejemplo4_personas" schema="APP">
    <id name="id" column="PERSONA_ID" type="long">
        <generator class="native" />
    </id>

    <!--  discriminator siempre a continuacion del id  -->
    <discriminator column="DISCRIMINADOR" type="string" length="1" />

    <property name="nombre" column="NOMBRE" type="string" />

    <property name="apellido" column="APELLIDO" type="string" />


    <subclass name="app.modelo.Alumno" extends="app.modelo.Persona"
              discriminator-value="A">
        <property name="curso" column="CURSO" type="string" />
    </subclass>

    <subclass name="app.modelo.Profesor" extends="app.modelo.Persona"
              discriminator-value="P">
        <property name="titulacion" column="TITULACION" type="string" />
    </subclass>

</class>
```

**Gráfico 118. Archivo de mapeo de la entidad Persona**

Recordamos que esta estrategia necesita de un campo discriminador, el cual estamos declarando con el elemento <discriminator>.

Con el elemento <subclass> indicamos que las entidades de tipo Persona y Profesor son subclases de la entidad Persona. Además indicamos el valor para cada campo discriminador y las propiedades de cada subclase.

Resaltar que solo se necesita un solo archivo de mapeo, el de la superclase Persona.

El código de este ejemplo lo encontrareis en **Ejemplo4_Hibernate_SingleTable**

## JOINED

```
<class name="app.modelo.Persona" table="ejemplo5_personas" schema="APP">
    <id name="id" column="PERSONA_ID" type="long">
        <generator class="native" />
    </id>

    <property name="nombre" column="NOMBRE" type="string" />

    <property name="apellido" column="APELLIDO" type="string" />

    <joined-subclass name="app.modelo.Alumno" extends="app.modelo.Persona"
              table="ejemplo5_alumnos">
        <key column="ALUMNO_ID" />
        <property name="curso" column="CURSO" type="string" />
    </joined-subclass>

    <joined-subclass name="app.modelo.Profesor" extends="app.modelo.Persona"
              table="ejemplo5_profesores">
        <key column="PROFESOR_ID" />
        <property name="titulacion" column="TITULACION" type="string" />
    </joined-subclass>
</class>
```

**Gráfico 119. Archivo de mapeo de la entidad Persona**

En esta estrategia no necesitamos el campo discriminador. Utilizamos el elemento <joined-subclass> para indicar las entidades que son subclases.

Como atributos especificamos el nombre de cada tabla y el campo que actuará como clave primaria, además de las propiedades de cada subclase.

Únicamente utilizamos el archivo de mapeo de la superclase.

El código de este ejemplo lo encontrareis en **Ejemplo5_Hibernate_Joined**

## TABLEPERCLASS

```xml
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE hibernate-mapping PUBLIC
  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>

  <class name="app.modelo.Persona" table="ejemplo3_personas">
      <id name="id" column="PERSONA_ID" type="long">
          <generator class="assigned" />
      </id>

      <property name="nombre" column="NOMBRE" type="string" />

      <property name="apellido" column="APELLIDO" type="string" />


      <union-subclass name="app.modelo.Alumno" extends="app.modelo.Persona" table="ALUMNOS_3">
          <property name="curso" column="CURSO" type="string" />
      </union-subclass>

      <union-subclass name="app.modelo.Profesor" extends="app.modelo.Persona" table="PROFESORES_3">
          <property name="titulacion" column="TITULACION" type="string" />
      </union-subclass>

  </class>

</hibernate-mapping>
```

**Gráfico 120. Mapeo con estrategia TablePerClass**

En esta estrategia tampoco necesitamos el campo discriminador. Utilizamos el elemento <union-subclass> para indicar las entidades que son subclases.

Como atributos especificamos el nombre de cada tabla además de las propiedades de cada subclase.

Únicamente utilizamos el archivo de mapeo de la superclase.

Todo el código de este ejemplo lo encontrareis en **Ejemplo3_Hibernate_TablePerClass.zip**

## INTERCEPTORES

Son elementos de tipo Listener que se encargan de detectar determinados eventos en las entidades cuando se persisten, se eliminan, ...etc.

Para implementar un interceptor debemos crear una clase que herede de EmptyInterceptor.

En la clase se debe sobrescribir el método necesario según el evento que deseemos detectar. En nuestro ejemplo sería al persistir las entidades (método onSave).

```
public class MiInterceptor extends EmptyInterceptor implements Serializable{

    @Override
    public boolean onSave(Object entity, // entidad a persistir
                          Serializable id, // PK
                          Object[] state, // valores a persistir
                          String[] propertyNames, // nombres de propiedades
                          Type[] types)  { // tipos de las propiedades

        if (entity instanceof Persona){
            System.out.println("PK: " + id);

            System.out.println("Propiedades: ");
            for(String prop:propertyNames){
                System.out.print(prop +  "  ");
```

**Gráfico 121. Fragmento de la clase Interceptor**

Aparte de este método la clase EmptyInterceptor otros métodos:

- afterTransactionBegin; Se invoca al iniciar la transacción
- afterTransactionCompletion; se ejecuta al efectuar el commit o rollback de la transacción.
- beforeTransactionCompletion; se invoca antes de efectuar el commit, no el rollback.
- onDelete; se invoca antes de eliminar la instancia
- onSave; antes de persistir la entidad
- onLoad; antes de recuperar la entidad.

Para aplicar el interceptor creado anteriormente lo haremos generando una instancia de él en el momento de obtener el objeto Session.

```
Session session = sf.openSession(new MiInterceptor());
```

**Gráfico 122. Aplicar el interceptor**

Todo el código de este ejemplo lo encontrareis en **Ejemplo6_Hibernate_Interceptores.zip**

## HQL

Hibernate utiliza un lenguaje de consulta potente (HQL) que se parece a SQL. Sin embargo, comparado con SQL, HQL es completamente orientado a objetos y comprende nociones como herencia, polimorfismo y asociación.

Como podréis comprobar la sintaxis utilizada por HQL es prácticamente la misma que la utilizada por JPQL.

## LA CLÁUSULA FROM

La consulta posible más simple de Hibernate es de esta manera:

```
from Cat as cat
```

**Gráfico 123. Ejemplo cláusula FROM**

Esta consulta asigna el alias cat a las instancias Cat, de modo que puede utilizar ese alias luego en la consulta. La palabra clave as es opcional.

## LA CLÁUSULA SELECT

La cláusula SELECT escoge qué objetos y propiedades devolver en el conjunto de resultados de la consulta.

```
select cat.mate from Cat cat
```

**Gráfico 124. Ejemplo cláusula SELECT**

Las consultas pueden retornar propiedades de cualquier tipo de valor incluyendo propiedades del tipo componente:

```
select cust.name.firstName from Customer as cust
```

**Gráfico 125. Ejemplo de consulta retornando propiedades de componente**

Las consultas pueden retornar múltiples objetos y/o propiedades como un array de tipo Object[],

```
select mother, offspr, mate.name
from DomesticCat as mother
    inner join mother.mate as mate
    left outer join mother.kittens as offspr
```

**Gráfico 126. Consulta que devuelve un array de objetos**

O como una List:

```
select new list(mother, offspr, mate.name)
from DomesticCat as mother
    inner join mother.mate as mate
    left outer join mother.kittens as offspr
```

**Gráfico 127. Consulta que devuelve una lista de objetos**

## FUNCIONES DE AGREGACIÓN

Las consultas HQL pueden incluso retornar resultados de funciones de agregación sobre propiedades:

```
select avg(cat.weight), sum(cat.weight), max(cat.weight), count(cat)
from Cat cat
```

**Gráfico 128. Ejemplo de consulta con funciones de agregación**

Las funciones de agregación soportadas son:

- avg(...), sum(...), min(...), max(...)
- count(*)
- count(...), count(distinct ...), count(all...)

## CONSULTAS POLIMÓRFICAS

Una consulta como:

```
from Cat as cat
```

**Gráfico 129. Consulta que devuelve entidades de tipo Cat y sus subclases**

Devuelve instancias no solamente de Cat, sino también de subclases como DomesticCat. Las consultas de Hibernate pueden nombrar cualquier clase o interfaz Java en la cláusula from. La consulta retornará instancias de todas las clases persistentes que extiendan esa clase o implementen la interfaz.

## LA CLÁUSULA WHERE

La cláusula WHERE le permite refinar la lista de instancias retornadas. Si no existe ningún alias, puede referirse a las propiedades por nombre:

```
from Cat where name='Fritz'
```

**Gráfico 130. Consulta con cláusula WHERE**

## EXPRESIONES

Las expresiones utilizadas en la cláusula WHERE incluyen lo siguiente:

- operadores matemáticos: +, -, *, /
- operadores de comparación binarios: =, >=, <=, <>, !=, like
- operadores lógicos and, or, not
- Paréntesis ( ) que indican agrupación
- in, not in, between, is null, is not null, is empty, is not empty, member of y not member of
- Caso "simple", case ... when ... then ... else ... end, y caso "buscado", case when ... then ... else ... end
- concatenación de cadenas ...||... o concat(...,...)
- current_date(), current_time() y current_timestamp()
- second(...), minute(...), hour(...), day(...), month(...), and year(...)
- Cualquier función u operador definido por EJB-QL 3.0: substring(), trim(), lower(), upper(), length(), locate(), abs(), sqrt(), bit_length(), mod()
- coalesce() y nullif()
- str() para convertir valores numéricos o temporales a una cadena legible.
- cast(... as ...), donde el segundo argumento es el nombre de un tipo de Hibernate , y extract(... from ...) si cast() y extract() es soportado por la base de datos subyacente.
- la función index() de HQL, que se aplica a alias de una colección indexada unida.
- Las funciones de HQL que tomen expresiones de ruta valuadas en colecciones: size(), minelement(), maxelement(), minindex(), maxindex(), junto con las funciones especiales elements() e indices, las cuales se pueden cuantificar utilizando some, all, exists, any, in.
- Cualquier función escalar SQL soportada por la base de datos como sign(), trunc(), rtrim() y sin()
- parámetros posicionales JDBC ?
- parámetros con nombre :name, :start_date y :x1
- literales SQL 'foo', 69, 6.66E+2, '1970-01-01 10:00:01.0'
- constantes Java public static finaleg.Color.TABBY

## LA CLÁUSULA ORDER BY

La lista retornada por una consulta se puede ordenar por cualquier propiedad de una clase retornada o componentes:

```
from DomesticCat cat
order by cat.name asc, cat.weight desc, cat.birthdate
```

**Gráfico 131. Ejemplo de consulta con resultados ordenados**

Los asc o desc opcionales indican ordenamiento ascendente o descendente respectivamente.

## LA CLÁUSULA GROUP BY

Una consulta que retorna valores agregados se puede agrupar por cualquier propiedad de una clase retornada o componentes:

```
select cat.color, sum(cat.weight), count(cat)
from Cat cat
group by cat.color
```

**Gráfico 132. Ejemplo de consulta agrupada**

## SUBCONSULTAS

Para bases de datos que soportan subconsultas, Hibernate soporta subconsultas dentro de consultas. Una subconsulta se debe encerrar entre paréntesis (frecuentemente por una llamada a una función de agregación SQL). Incluso se permiten subconsultas correlacionadas (subconsultas que se refieren a un alias en la consulta exterior).

```
from Cat as cat
where not exists (
    from Cat as mate where mate.mate = cat
)
```

**Gráfico 133. Ejemplo de subconsulta**

```
from DomesticCat as cat
where cat.name not in (
    select name.nickName from Name as name
)
```

**Gráfico 134. Ejemplo de subconsulta**

# INTEGRACION CON SPRING

Para integrar Hibernate con Spring tenemos dos opciones:

La primera de ellas es la más simple y consiste en declarar únicamente el bean SessionFactory en Spring y recuperarlo para manejar las entidades desde Hibernate.

Veámoslo con un ejemplo:

Declaramos el bean DataSource en el archivo de configuración de Spring con todos los datos para posibilitar la conexión a la base de datos.

```
<bean id="miDataSource"
         class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName"
                value="org.apache.derby.jdbc.ClientDriver" />
    <property name="url" value="jdbc:derby://localhost:1527/sample" />
    <property name="username" value="app" />
    <property name="password" value="app" />
</bean>
```

**Gráfico 135. Declaración del bean DataSource**

Declaramos el bean SessionFactory:

```
<bean id="sessionFactory"
         class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="dataSource" ref="miDataSource" />

    <property name="mappingResources">
        <list>
            <value>app/modelo/Persona.hbm.xml</value>
        </list>
    </property>

    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.dialect">org.hibernate.dialect.DerbyDialect</prop>
            <prop key="hibernate.show_sql">true</prop>
            <prop key="hibernate.connection.pool_size">1</prop>
            <prop key="hibernate.hbm2ddl.auto">create</prop>
            <prop key="hibernate.current_session_context_class">thread</prop>
            <prop key="hibernate.cache.provider_class">
                    org.hibernate.cache.NoCacheProvider
            </prop>
        </props>
    </property>
</bean>
```

**Gráfico 136. Declaración del bean SessionFactory**

Creamos el contenedor de beans y recuperamos el bean sessionFactory, a partir de aquí podemos trabajar con las entidades utilizando Hibernate.

```
ApplicationContext contenedor =
        new ClassPathXmlApplicationContext("spring.xml");

SessionFactory sf =
        (SessionFactory) contenedor.getBean("sessionFactory");
```

**Gráfico 137. Recuperación del bean sessionFactory**

Todo el código de este ejemplo lo encontrareis en **Ejemplo7_Spring_Hibernate_Entidad.zip**

La segunda forma de incorporar Spring e Hibernate consiste en seguir los siguientes pasos:

1. Crear un SessionFactory:
2. Crear la plantilla
3. Crear un bean del dao.
4. Crear el dao

### 1. Crear un SessionFactory

Igual que en el modelo anterior declaramos el bean DataSource con los datos de la conexión.

```
<bean id="miDataSource"
        class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName"
                value="org.apache.derby.jdbc.ClientDriver" />
    <property name="url" value="jdbc:derby://localhost:1527/sample" />
    <property name="username" value="app" />
    <property name="password" value="app" />
</bean>
```

**Gráfico 138. Declaración del bean DataSource**

Declaramos el bean SessionFactory para ello podemos utilizar dos clases

- LocalSessionFactoryBean; utilizamos esta clase cuando hacemos los mapeos con XML en Hibernate
- AnnotationSessionFactoryBean; utilizamos esta otra cuando los mapeos los hacemos con anotaciones

En nuestro ejemplo utilizamos la primera clase:

```
<bean id="sessionFactory"
         class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="dataSource" ref="miDataSource" />

    <property name="mappingResources">
        <list>
            <value>app/modelo/Persona.hbm.xml</value>
        </list>
    </property>

    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.dialect">org.hibernate.dialect.DerbyDialect</prop>
            <prop key="hibernate.show_sql">true</prop>
            <prop key="hibernate.connection.pool_size">1</prop>
            <prop key="hibernate.hbm2ddl.auto">create</prop>
            <prop key="hibernate.current_session_context_class">thread</prop>
            <prop key="hibernate.cache.provider_class">
                        org.hibernate.cache.NoCacheProvider
            </prop>
        </props>
    </property>
</bean>
```

**Gráfico 139. Declaración del bean SessionFactory**

## 2. Crear la plantilla

Del mismo modo que en Spring JDBC hacíamos uso de las plantillas (Templates) para no tener que escribir tanto código redundante. Spring proporciona otro tipo muy similar de plantilla para su integración con Hibernate.

```
<bean id="miTemplate"
      class="org.springframework.orm.hibernate3.HibernateTemplate">
    <property name="sessionFactory" ref="sessionFactory" />
</bean>
```

**Gráfico 140. Declaración del bean Template**

## 3. Crear el dao

```java
public class PersonaDAO {

    private HibernateTemplate template;

    public HibernateTemplate getTemplate() {
        return template;
    }

    public void setTemplate(HibernateTemplate template) {
        this.template = template;
    }

    public void insertar(Persona persona){
        template.save(persona);
    }

    public List<Persona> verTodas(){
        String query = "select p from Persona p";
        return template.find(query);
    }

    public Persona buscarPorDni(String dni){
        String query =
                "select p from Persona p where p.pk.dni = " + "'" + dni + "'";
        return (Persona) template.find(query).get(0);
    }

}
```

**Gráfico 141. Creación de la clase DAO**

En la clase PersonaDAO creamos una propiedad de tipo HibernateTemplate evidentemente con sus métodos get() y set().

Esta será la plantilla que utilizaremos para manejar las entidades en los métodos insertar, verTodas, buscarPorDni, ...etc.

## 4. Crear un bean del dao.

Declaramos el bean de la clase PersonaDAO donde inyectamos como propiedad la plantilla declarada en el paso 2.

```xml
<bean id="miDAO" class="app.persistencia.PersonaDAO">
    <property name="template" ref="miTemplate" />
</bean>
```

**Gráfico 142. Declaración del bean DAO**

Todo el código de este ejemplo lo encontrareis en **Ejemplo8_Spring_Hibernate_Template.zip**

# INDICE DE GRÁFICOS