Manual de Teoría de JPA e Hibernate

Contenido

1. PERSISTENCIA 4
2. JPA 5
INTRODUCCION 5
MAPEOS DE ENTIDADES 5
@Id 5
@IdClass 6
@Embeddable 6
@Embedded 7
@EmbeddedId 7
@Table 8
@Column 8
@SecondaryTable 9
@Enumerated 10
@Lob 11
@Basic 11
@Temporal 12
MODELO DE DOMINIO 12
RELACION ENTRE ENTIDADES 13
@OneToOne 13
@OneToMany y @ManyToOne 14
@ManyToMany 15
OPERACIONES EN CASCADA 15
3. HIBERNATE 17
HIBERNATE.CFG.XML 17
CREAR ENTIDADES 18
MAPEAR ENTIDADES 19
ENTIDAD 20
CLAVES PRIMARIAS 20
PROPIEDADES SIMPLES 22
TIPOS ENUMERADOS 22
CAMPOS EMBEBIDOS 23
MAPEOS DE CAMPOS A OTRA TABLA 24
SESSION Y SESSION FACTORY 24
RELACION ENTRE ENTIDADES 26
RELACIÓN ONETOONE 26
RELACIÓN ONETOMANY Y MANYTOONE 27
RELACIÓN MANYTOMANY 28
ESTRATEGIAS DEL MAPEO DE HERENCIA 29
- SingleTable 29
- Joined 29
TABLEPERCLASS 32
INTERCEPTORES 32
HQL 33
LA CLÁUSULA FROM 34
LA CLÁUSULA SELECT 34
FUNCIONES DE AGREGACIÓN 35
CONSULTAS POLIMÓRFICAS 35
LA CLÁUSULA WHERE 35
EXPRESIONES 36
LA CLÁUSULA ORDER BY 36

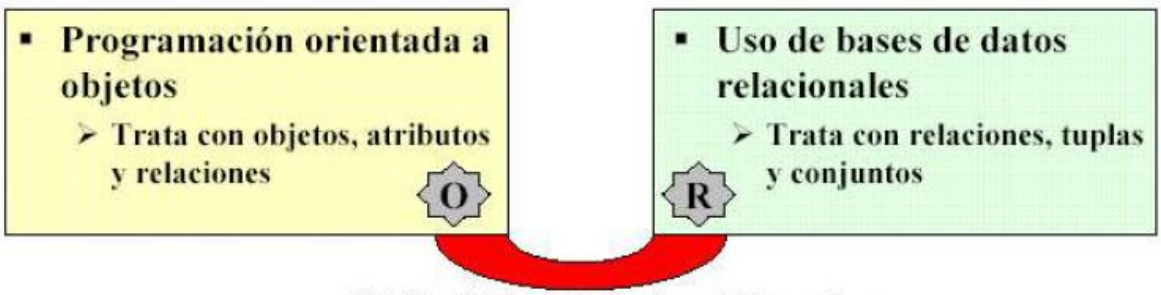
LA CLÁUSULA GROUP BY 37
SUBCONSULTAS.....37
INTEGRACION CON SPRING 38

1. PERSISTENCIA

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 (Object Relational Mapping). 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.



ORM: Object-Relational Mapping

- **Problema: un 35% del código de una aplicación para realizar la correspondencia O ↔ R**
- **Solución: utilizar una ORM, por ejemplo Hibernate**

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.

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

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 campo o si forma parte de una PK compuesta, de varios campos.

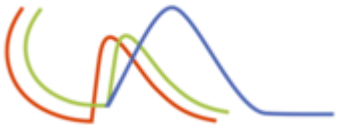
```
@Entity
public class Persona implements Serializable {

	@Id
	private String nif;
```

Gráfico 2. Marcamos la propiedad nif como clavePK

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().

```
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

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

```
@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

@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.

```
@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

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.

```
@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.

```java
@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

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

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

    @EmbeddedId
    PersonaPK personapk;
```

Gráfico 9. Declaración de clave primaria

@Table

Mediante esta anotación indicamos el nombre de la tabla donde se persistirán o recuperan 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.

```java
@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.

```java
@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

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 EstadoCivil

```
@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

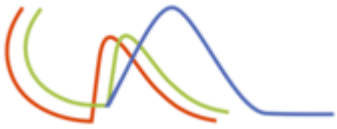
@Basic

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 implicita.

```
@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.

MODELO DE DOMINIO

Que ocurre cuando queremos trabajar con varias entidades a la vez que están relacionadas entre ellas. Por ejemplo, cómo 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:

literatura.

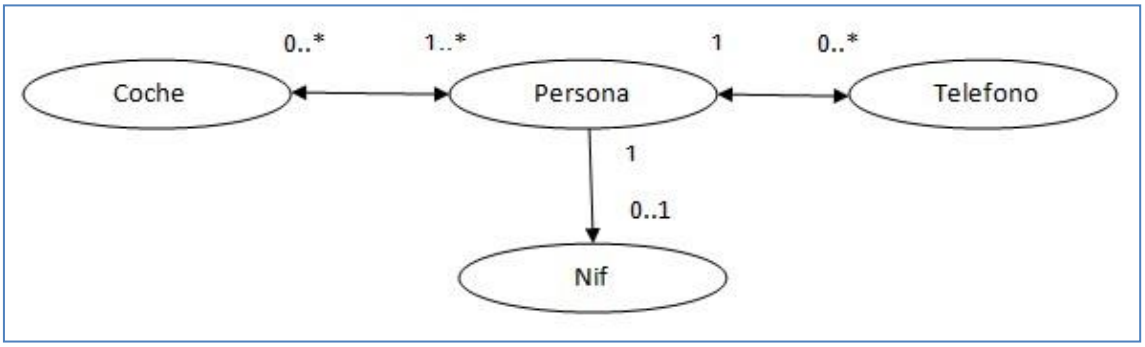


Gráfico 26. Ejemplo modelo de dominio

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

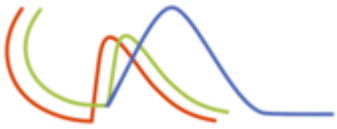
Para establecer las relaciones entre entidades utilizamos anotaciones como:

- @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 es necesario 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

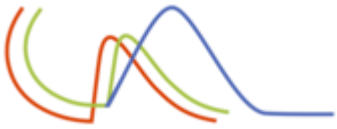
```
@ManyToMany(mappedBy="coches",
            cascade={CascadeType.MERGE,
                     CascadeType.PERSIST,
                     CascadeType.REFRESH})
private Set<Persona> propietarios = new HashSet<Persona>();

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

Gráfico 32. Anotación @ManyToMany

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.

3. HIBERNATE

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

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
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
        <property name="hibernate.connection.password">root</property>
        <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/ejemplos_hibernate?useSSL=false</pr
        <property name="hibernate.connection.username">root</property>
        <property name="hibernate.dialect">org.hibernate.dialect.MySQL5InnoDBDialect</property>
        <property name="hibernate.show_sql">true</property>
        <property name="hibernate.hbm2ddl.auto">create</property>
    </session-factory>
</hibernate-configuration>

```

Ilustración 1 -Archivo hibernate.cfg.xml

```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

CREAR ENTIDADES

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

```java
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) {...}
```

Ilustración 2-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
<?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>
```

Ilustración 3 -Archivo de

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">
```

Ilustración 4 -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>
```

Ilustración 5 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.

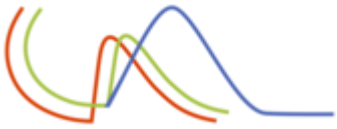
- Clave primaria compuesta

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>
```

Ilustración 6 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"/>
```

Ilustración 7 -Mapeo de la propiedad nombre

En el siguiente ejemplo mapeamos una propiedad de tipo Date.

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

Ilustración 8-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.

```java
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;
   }
```

Ilustración 9 -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.

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

Ilustración 10 -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>
```

Ilustración 11 -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>
```

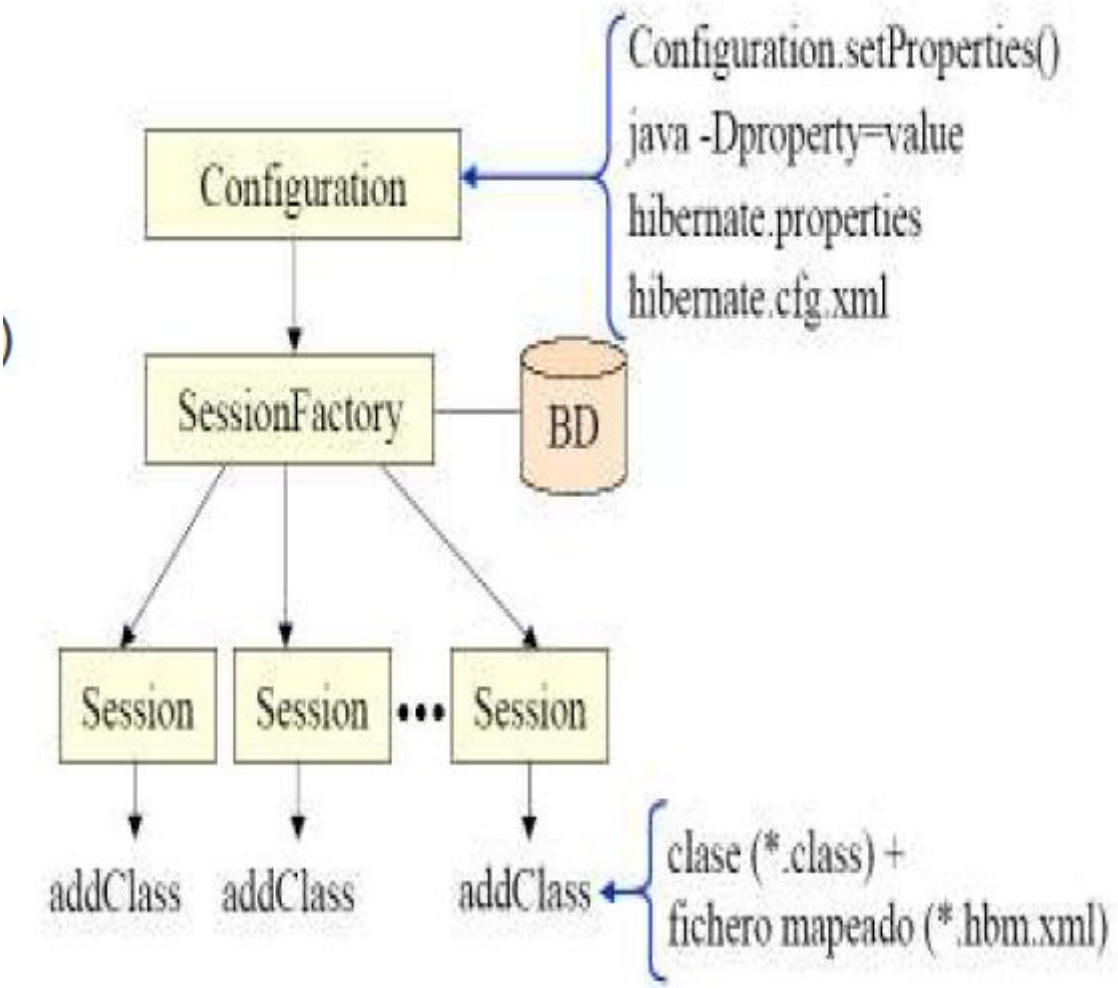
Ilustración 12-Ejemplo de uso de tablas secundarias

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.

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 factory = new Configuration()
        .configure("hibernate.cfg.xml")
        .addAnnotatedClass(Cliente.class)
        .addAnnotatedClass(Pedido.class)
        .addAnnotatedClass(InfoAdicional.class)
        .buildSessionFactory();
```

Ilustración 13 - 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();
```

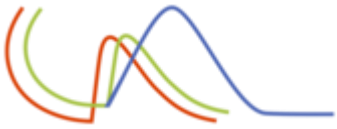


Ilustración 14 -Objeto Session

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

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

Ilustración 15 -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();
}
```

Ilustración 16 -Fragmento de persistencia de entidades

RELACION ENTRE ENTIDADES

Se puede 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 un ejemplo.

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"/>
```

Ilustración 17 -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" />
```

Ilustración 18 - 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>();
```

Ilustración 19 - Conjunto de teléfonos en la entidad Persona

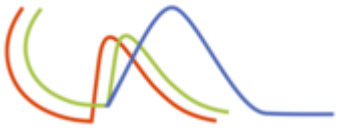
Recordamos que las propiedades de tipo colección necesitan de un método de sincronización.

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

Ilustración 20 - 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>
```

Ilustración 21 - 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" />
```

Ilustración 22 - 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>();
```

Ilustración 23 - Conjunto de coches en la entidad Persona

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

Ilustración 24 - 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>
```

Ilustración 25 - 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>
```

Ilustración 26 - 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);
}
```

Ilustración 27 - Método de sincronización en la entidad Persona

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

Ilustración 28 - Método de sincronización en la entidad Coche

ESTRATEGIAS DEL MAPEO DE HERENCIA

Recordamos que existen tres estrategias del mapeo de herencia:

- 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>
```

Ilustración 29 - 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.

JOINED

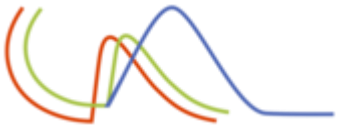
```
<class name="app.modelo.Persona" table="ejemplo5_personas" schema="APP">
    <id name="id" column="PERSONA_ID" type="long">
        <generator class="native" />
    </id>

    <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>
```

Ilustración 30 - 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.



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>
```

Ilustración 31 - 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.

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 +  "  ");
```

Ilustración 32 - 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.

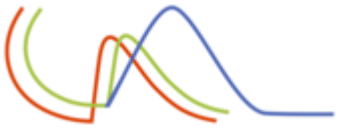
Para aplicar el interceptor creado anteriormente lo haremos generando una instancia de él en el momento de obtener el objeto Session.

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

Ilustración 33 - Aplicar el interceptor

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.



LA CLÁUSULA FROM

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

```
from Cat as cat
```

Ilustración 34 - 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
```

Ilustración 35 - 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
```

Ilustración 36 - 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
```

Ilustración 37 - 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
```

Ilustración 38 - 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
```

Ilustración 39 - 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:

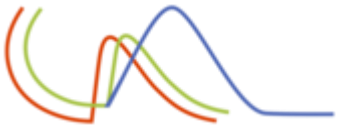
```
from Cat as cat
```

Ilustración 40 - 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'
```

Ilustración 41- 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
```

Ilustración 42 - 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
```

Ilustración 43 - Ejemplo de consulta agrupada

SUBCONSULTAS

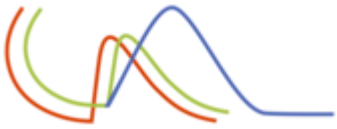
Para bases de datos que soportan subconsultas, Hibernate soporta subconsultas dentro de consultas. Una subconsulta se debe encerrar entre paréntesis (frecuentemente por una llamada a una función de agregación SQL). Incluso se permiten subconsultas correlacionadas (subconsultas que se refieren a un alias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

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

Ilustración 44 - Ejemplo de subconsulta

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

Ilustración 45 - 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>
```

Ilustración 46 - 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>
```

Ilustración 47 - 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.

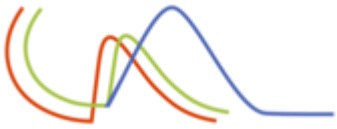
```
ApplicationContext contenedor =
        new ClassPathXmlApplicationContext("spring.xml");

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

Ilustración 48 - Recuperación del bean sessionFactory

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>
```

Ilustración 49 - 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>
```

Ilustración 50 - 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>
```

Ilustración 51 - Declaración del bean Template

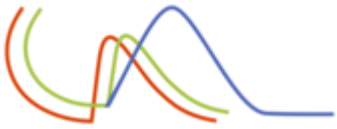
3. Crear el dao

```
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);
    }

}
```

Ilustración 52 - 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.

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

Ilustración 53 - Declaración del bean DAO