FORMACIÓN EN NUEVAS TECNOLOGÍAS

HIBERNATE AVANZADO

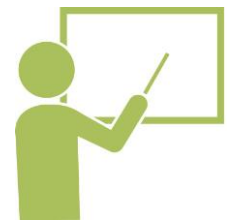
1

ICONO TRAINING

Formación en Nuevas Tecnologías

- www.iconotc.com
- training@iconotc.com
- linkedin.com/company/icono-training-consulting

FORMADOR



Javier Martín

- Consultor y Formador en las tecnologías de la información.

¡Síguenos en las Redes Sociales!

2

HIBERNATE AVANZADO

DURACIÓN

- 14 horas

LUGAR/FECHAS /HORARIO:

- Madrid. 11 y 12 de Febrero de 2019. De 9:00 hs a 14:00 hs y de 15:00 hs a 17:00 hs.

CONTENIDOS:

- **1.** Arquitectura (Impartición según nivel de los asistentes)
- **2.**Principales características ((Impartición según nivel de los asistentes)
- **3.**Clases simples
- **4.**Clases con componentes
- **5.**Clases con relaciones unidireccionales
- **6.**Clases con relaciones bidireccionales
- **7.**Clases con jerarquías de herencia
- **8.**El sistema de eventos e interceptadores

HIBERNATE AVANZADO

CONTENIDOS (continuación):

- **9.** Lenguaje de consulta
- **10.**Transacciones y concurrencia
- **11**.Uso de anotaciones para definir y manipular la persistencia
- **12**.Uso de Proxies en Hibernate e inicialización perezosa de colecciones y proxies.
- **13**. Optimización de navegación entre entidades (Estrategias para minimizar el número de consultas)
- **14.** Recuperación de lotes.
- **15**. Caches en Hibernate, primer y segundo nivel, Diferentes proveedores de caché de segundo nivel y ventajas de uso de cada uno, y ejemplos de uso de objetos relacionales (Configurados por regiones en la cache) .
- **16.** Diferencias entre el uso de StandardQueryCache y UpdateTimestampsCache (Y ejemplo práctico de uso) .

Hibernate

http://www.hibernate.org

5

INSTALACIÓN

9

Con Eclipse

- Descargar Hibernate:
  - http://hibernate.org/orm/downloads/
- Descargar e instalar JDK:
  - http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html
- Descargar y descomprimir Eclipse:
  - https://www.eclipse.org/downloads/
- Añadir a Eclipse las Hibernate Tools
  - Help > Eclipse Marketplace: JBoss Tools
- Crear una User Librarie para Hibernate
  - Window > Preferences > Java > Build Path > User Libraries > New
  - Add External JARs: \lib\required
- Descargar y registrar la definición del driver JDBC
  - √ Window > Preferences > Data Management > Connectivity > Driver Definition > Add

10

Oracle Driver con Maven

- http://www.oracle.com/technetwork/database/features/jdbc/index-091264.html
- Instalación de Maven:
  - Descargar y descomprimir (https://maven.apache.org)
  - Añadir al PATH: C:\Program Files\apache-maven\bin
  - Comprobar en la consola de comandos: mvn –v
- Descargar el JDBC Driver de Oracle (ojdbc6.jar):
  - https://www.oracle.com/technetwork/apps-tech/jdbc-112010-090769.html
- Instalar el artefacto ojdbc en el repositorio local de Maven
  - mvn install:install-file -Dfile=**Path/to/your**/ojdbc6.jar -DgroupId=com.oracle -DartifactId=ojdbc6 -Dversion=11.2.0 -Dpackaging=jar
- En el fichero pom.xml:

```
<dependency>
    <groupId>com.oracle</groupId>
    <artifactId>ojdbc6</artifactId>
    <version>11.2.0</version>
</dependency>
```


11

Creación del Proyecto

- New → Maven Proyect
- Dependencias en pom.xml

```
<properties>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
</properties>
<dependencies>
    <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-core</artifactId>
            <version>5.4.1.Final</version>
    </dependency>
    <dependency>
            <groupId>com.oracle</groupId>
            <artifactId>ojdbc6</artifactId>
            <version>11.2.0</version>
    </dependency>
</dependencies>
```


INTRODUCCIÓN A HIBERNATE

Introducción a Hibernate

- Cuando nos ponemos a desarrollar, en el código final lo que tenemos que hacer a veces es mucho.
- Deberemos desarrollar código para:
  - Lógica de la aplicación
  - Acceso a Base de Datos
  - EJB para modularizar las aplicaciones.
- Deberemos también programar en diferentes Lenguajes:
  - Java (mayoritariamente)
  - SQL
  - HTML, etc

"La vida es corta, dedique menos tiempo a escribir código para la unión BBDD-Aplicación JAVA y más tiempo añadiendo nuevas características"

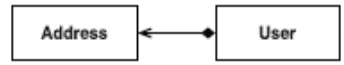
Introducción a Hibernate

- Las bases de datos relacionales son indiscutiblemente el centro de la empresa moderna.
- Los datos de la empresa se basan en entidades que están almacenadas en ubicaciones de naturaleza relacional. (Base de Datos)
- Los actuales lenguajes de programación, como Java, ofrecen una visión intuitiva, orientada a objetos de las entidades de negocios a nivel de aplicación.
- Se han realizado mucho intentos para poder combinar ambas tecnologías (relacionales y orientados a objetos), o para reemplazar uno con el otro, pero la diferencia entre ambos es muy grande.

Discrepancia del Paradigma

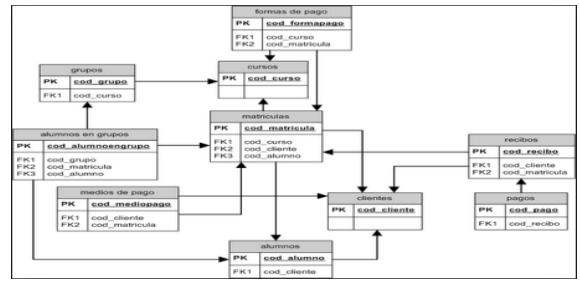
- Cuando trabajamos con diferentes paradigmas (Objetos vs Relacional) la discrepancia a la hora de unirlos surge a medida que complicamos el diseño.

- Cuando disponemos de clases sencillas y relaciones sencillas, no hay problemas.



Lo anterior no es lo habitual.

Pero esto SI:



17

Discrepancia del Paradigma

- Bauer & King (Bauer, C. & King, G. 2007) presentan una lista de los problemas de discrepancia en los paradigmas objeto/relacional

  - Problemas de granularidad
    - √ Java pueden definir clases con diferentes niveles de granularidad.
      - – Cuanto más fino las clases de grano (Direcciones), éstas pueden ser embebida en las clases de grano grueso (Usuario)
    - √ En cambio, el sistema de tipo de la base de datos SQL son limitados y la granularidad se puede aplicar sólo en dos niveles
      - – en la tabla (tabla de usuario) y el nivel de la columna (columna de dirección)
  - Problemas de subtipos
    - √ La herencia y el polimorfismo son las características básicas y principales del lenguaje de programación Java.
    - √ Los productos de base de datos SQL en general, no son compatibles con subtipos y herencia de tablas.

18

Discrepancia del Paradigma

- Problemas de identidad
  - √ Objetos Java definen dos nociones diferentes de identidad:
    - Identidad de objeto (equivalente a la posición de memoria, comprobar con un == b).
    - La igualdad como determinado por la aplicación de los métodos ***equals ().***
  - √ La identidad de una fila de base de datos se expresa como la clave primaria.
  - √ Ni ***equals()*** ni **==** es equivalente a la clave principal.

- Problemas de Asociaciones
  - √ El lenguaje Java representa a las asociaciones mediante utilizan referencias a objetos
    - Las asociaciones entre objetos son punteros **unidireccionales**.
  - √ Las asociaciones en BBDD están representados mediante el uso de claves externas.
    - Todas las asociaciones en una base de datos relacional son **bidireccional**

19

ORM

- Las siglas ORM significan “Object-Relational mapping” (Mapeo Objeto-Relacional)
- El ORM es un framework de persistencia de nuestros datos (objetos) a una base de datos relacional, es decir, código que escribimos para guardar el valor de nuestras clases en una base de datos relacional.
- ORM es el nombre dado a las soluciones automatizadas para solucionar el problema de falta de coincidencia (Objetos-Relacional).
- Un buen número de sistemas de mapeo objeto-relacional se han desarrollado a lo largo de los años, pero su efectividad en el mercado ha sido diversa.

√	Hibernate	TopLink
√	CocoBase	EclipseLink
√	OpenJPA	Kodo
√	DataNucleus	Amber

20

ORM

AÑO	Descripción	HIBERNATE
1999	Java Community Process empezó a estandarizar una tecnología de ORM llamada EBJ 1.0 en el JSR-19.	
2001	EJB 2.0. Ampliaron la funcionalidad existentes hasta ese momento	Gavin King creó Hibernate como alternativa a EJB x
2003	EJB 2.1 en el JSR-15. - Ampliaron la funcionalidad existentes hasta ese momento	Aparece Hibernate2 - Ofrece muchas mejoras significativas con respecto a la primera versión
2004	Java Data Objects (JDO) es una especificación de Java object persistence, que amplia la funcionalidades de EJB2 (Apenas usada)	Nace Hibernate 3.x - Nuevas características - arquitectura Interceptor/Callback - filtros y Anotaciones por el usuario
2006	EJB3 en JSR-220 y JPA en JSR-317	
2009-2010	EJB 3.1 en JSR-318	Hibernate 3.5 incluido en JPA
2011		Hibernate 4.0
2012-2013	EJB 3.2 en JSR-345	Hibernate 5.0

- **JPA** (Java Persistence API) es considerado el estándar de persistencia en Java desarrollada para la plataforma Java EE.
- **Hibernate** implementa la especificación de JPA

21

ORM

- Todo ORM deben de cumplir 4 especificaciones:

1. Un API para realizar operaciones básicas CRUD sobre los objetos de las clases persistentes. (JDBC)
2. Un lenguaje o API para la especificación de las consultas que hacen referencia a las clases y propiedades de clases (SQL)
3. Un elemento de ORM para interactuar con objetos transaccionales para realizar operaciones diversas (Conexión, validación, optimización, etc) SessionFactory
4. Un elemento (fichero) para especificar los metadatos de mapeo (XML)

22

Introducción a Hibernate

- La finalidad de Hibernate es:
  - Proporcionar un puente entre los datos relacionales (BBDD) y la Orientación a Objetos (Java) mediante su modelo de Mapeo Relacional/Objeto (ORM) .

- Para poder desarrollar en Hibernate, deberemos tener conocimientos de:
  - √ Conocer la lógica de la Aplicación, estructura de clases y objetos
  - √ JDBC
  - √ Conocer Lenguaje de consulta SQL y estructuras de almacenamiento relacional

- La meta de Hibernate es reducir drásticamente el tiempo y la energía que gasta el mantenimiento de este entorno, sin perder la potencia y la flexibilidad asociada a los dos mundos.

- Hibernate es una solución que satisface el 90% de todos las operaciones de una aplicación Java enfrentada a su base de datos relacional.

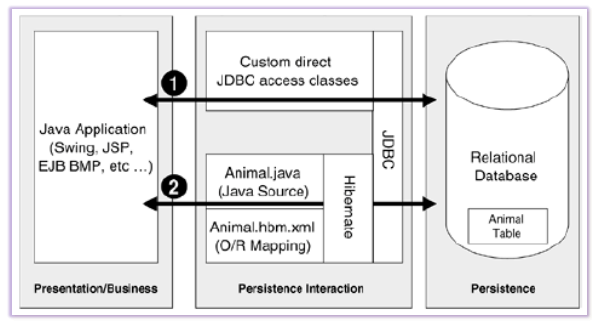
23

ORM: Hibernate

- Objetivo de Hibernate:

"Aliviar al desarrollador del 95 % de las tareas de programación relacionados con la persistencia de datos común."

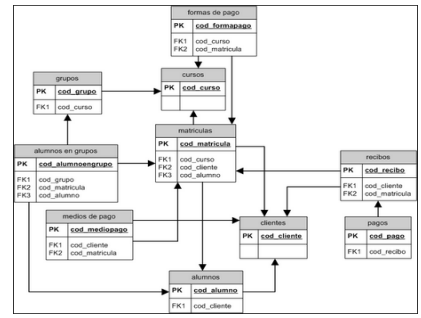
- Hibernate lo resuelve mediante la combinación de clases de Java y ficheros descriptores XML.



24

ORM: Hibernate

- Mediante el uso de Hibernate, un desarrollador se libera de escribir código JDBC y puede centrarse en la presentación y la lógica empresarial.
- La utilización de Hibernate y JDBC pueden coexistir sin ningún problema.
- Hibernate no obliga a seguir unas reglas estrictas ni patrones de diseño.
- Mediante Hibernate podemos:
  - √ Mapear objetos a tablas.
  - √ Expresar relaciones de 1 a 1
  - √ Expresar relaciones de 1 a N
  - √ Expresar relaciones de N a N

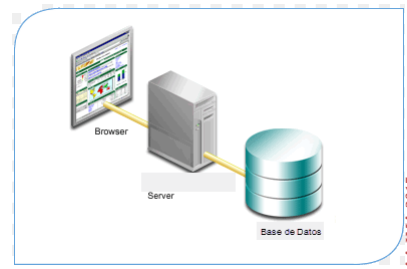


Hibernate: Módulos

- Hibernate implementa JPA, por lo que dispone de diferentes módulos para ampliar funcionalidades.
  - Hibernate Core
    - Es el servicio BASE para la persistencia.
    - Dispone de API nativa y ficheros XML para el mapeo de Entidades.
    - Dispone de un lenguaje de consulta llamado HQL.
    - Dispone de interfaces de consultas «criteria».
  - Hibernate Annotations
    - Estas anotaciones permiten especificar de una forma más compacta y sencilla la información de mapeo de las clases Java.
  - Hibernate EntityManager
    - (JPA) Es una API se utiliza para acceder a una base de datos en una unidad de trabajo.
    - Esta interfaz es similar a la *Session en Hib*ernate.
    - Se utiliza para crear y eliminar instancias de entidad persistentes.

Revisión de JDBC. Introducción

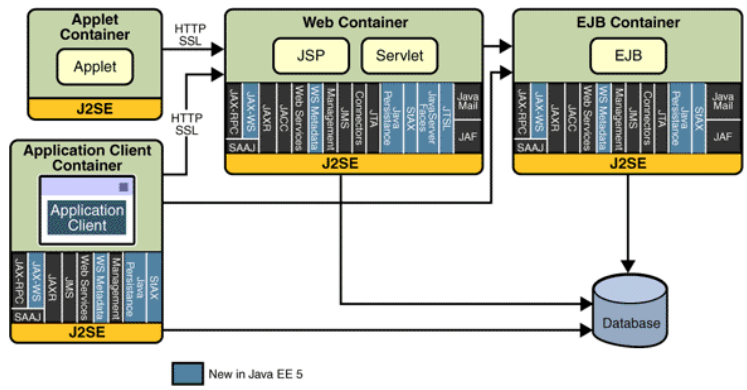
- Casi todas las aplicaciones web son una interfaz para mostrar datos obtenidos de almacenamientos internos.
- El almacenamiento mas utilizado son las bases de datos (DB).
- De hecho, lo que hace que las páginas web sean dinámica es el hecho de que hay una cantidad significativa de datos detrás de ellos.
- Una base de datos se compone de:
  - Unas estructuras de datos
  - Unas formas de acceso (esquemas)
  - Y Datos.
- Hoy en día, la mayoría de los SGBD son relacionales, es decir, sus datos están organizados en tablas.



30

Revisión de JDBC. Introducción

- Una estructura habitual de una aplicación Empresarial / J2EE es la siguiente:



31

Revisión de JDBC. Introducción

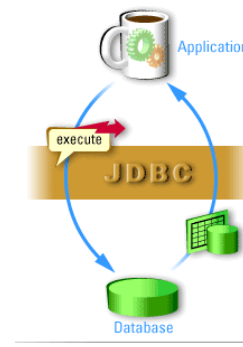
- JDBC proporciona una biblioteca estándar para acceder a bases de datos relacionales.
- Mediante el uso de la API de JDBC, puede acceder a una amplia variedad de bases de datos SQL con exactamente la misma sintaxis de Java.
- La API JDBC estandariza:
  - √ La conexión a bases de datos
  - √ La sintaxis para el envío de consultas
  - √ La confirmación de las transacciones
  - √ La estructura de datos que representa el resultado,
- La mayoría de las consultas siguen la sintaxis SQL estándar, y disponemos de múltiples posibilidades: *Cambio de BBDD, Hosts, Puertos, etc*

32

Pasos Generales JDBC

- Para realizar una conexión JDBC debemos realizar los siguiente pasos:

1. Cargar el controlador JDBC.
   - √ Especificamos el nombre de clase del controlador de base de datos en el método *Class.forName*.
2. Definir el URL de conexión con BBDD
   - √ Una URL de conexión especifica el host del servidor , el puerto y el nombre de base de datos con la que establecer una conexión.
3. Establecer la conexión
   - √ Necesitaremos un usuario y password correctos.
4. Preparar y Ejecutar consulta
   - √ Crear, modificar y enviar el objeto *Statemente*, con un consulta específica.
5. Procesas los resultados
   - √ Mediante el objeto *ResultSet*, se analizan y tratan los resultados obtenidos de la consulta.
6. Cierre de la conexión



33

Pasos Generales JDBC

Controladores JDBC

- Los controladores JDBC son el software que sabe cómo hablar con el servidor de base de datos real.
- Para cargar un determinado controlador, deberemos cargar la Clase de Java apropiada para ese Gestor.

√ Driver JDBC-ODBC:	sun.jdbc.odbc.JdbcOdbcDriver (incluido en la API de JDBC de la plataforma J2SE)	
√ Driver MySQL:	com.mysql.jdbc.Driver	(no incluido)
√ Driver DB2:	COM.ibm.db2.jdbc.app.DB2Driver	(no incluido)
√ Driver Oracle:	oracle.jdbc.driver.OracleDriver	(no incluido)

- Sino están incluidos, deberemos descargar el correspondiente ***.jar*** e insertarlo en el proyecto.

En uno de estos lugares:
- Proyect / propiedades / Java Build Path / Libraries / Add external JAR
- Directorio : WEB-INF/lib

Si necesitamos los JAR para múltiples aplicaciones →Directorio_de_instalación / common / lib

34

Pasos Generales JDBC

Controladores JDBC

- Sintaxis

```
try {
        Class.forName ("connect.microsoft.MicrosoftDriver");
        Class.forName ("oracle.jdbc.driver.OracleDriver");
        Class.forName ("com.sybase.jdbc.SybDriver");
        Class.forName ("com.mysql.jdbc.Driver");

    } catch(ClassNotFoundException cnfe) {
        System.err.println("Error loading driver: " + cnfe);
}
```

WEB.XML
```
<init-param>
   <param-name>Driver</param-name>
   <param-value>com.mysql.jdbc.Driver</param-value>
</init-param>
```

```
java.lang.Class.forName(config.getInitParameter("Driver"));
```


35

Pasos Generales JDBC

URL de Conexión

- Una vez que haya cargado el controlador JDBC, debe especificar la ubicación del servidor de base de datos.
- La URL de referencia utiliza:
  - √ Protocolo : Host : Puerto : Nombre_de_la_BBDD.
- El formato exacto se define en la documentación que viene con el controlador en particular

```
String host = "dbhost.yourcompany.com";
String dbName = "someName";
int port = 1234;

String oracleURL = "jdbc:oracle:thin:@" + host + ":" + port + ":" + dbName;
String sybaseURL = "jdbc:sybase:Tds:" + host + ":" + port + ":" + "?SERVICENAME=" + dbName;
String msAccessURL = "jdbc:odbc:" + dbName;
String MySQLURL = "jdbc:mysql://" + host + ":" + port + ":" + dbName;
```


36

Pasos Generales JDBC

Creando la Conexión

- Para hacer la conexión, deberemos pasar:
  - √ La dirección URL
  - √ Nombre de usuario de la Base de Datos
  - √ Contraseña del usuario anterior.
- Utilizaremos el método *getConnection* de la clase *DriverManager*,

```
String username = "Usuario01";
String password = "1234";
Connection connection = DriverManager.getConnection(oracleURL, username, password);
```

```
WEB.XML
:
<init-param>
   <param-name>Usuario</param-name>
   <param-value>Usuario01</param-value>
   :
</init-param>
```

```
String username= config.getInitParameter("Usuario")
String password= config.getInitParameter("Clave")
Connection connection =
DriverManager.getConnection(oracleURL, username, password);
```


37

Pasos Generales JDBC

TIPO	Clase JDBC
Implementación	java.sql.Driver
Conexión a base de datos	java.sql.Connection
Sentencias SQL	java.sql.Statement java.sql.PreparedStatement java.sql.CallableStatement
Datos	java.sql.ResulSet

- Ejemplo

```
public static void main(String args[]) {
      String usuario = "system";
      String password = "oracle";

      String host = "localhost"; // también puede ser una ip como "192.168.1.14"
      String puerto = "1521";
      String sid = "xe";

      String driver = "oracle.jdbc.driver.OracleDriver";
      String ulrjdbc = "jdbc:oracle:thin:" + usuario + "/" + password + "@" + host + ":" +
puerto + ":" + sid;
      try {
          Class.forName(driver).newInstance();
          Connection conexion= DriverManager.getConnection(ulrjdbc);
                                                  // Ya tenemos el objeto connection creado
          System.out.println("Nos hemos conectado correctamente a la Base de Datos");
          } catch (Exception e) { e.printStackTrace();  }
}
```


Pasos Generales JDBC

Preparar y Ejecutar Consulta

- Para preparar y ejecutar una consulta en la Base de Datos necesitamos crear un objeto Statement.
- Sintaxis

  Statement ***st*** *= connection.createStatement();*
- La mayoría, pero no todos, los controladores de base de datos permiten tener múltiples objetos *Statement* abiertos la misma conexión.
- Ahora utilizaremos este objeto para insertarle una Consulta SQL y ejecutarla

```
String query = "SELECT col1, col2, col3 FROM Tabla";

ResultSet resultSet = st.executeQuery (query);      SELECT
Int    filas = st.executeUpdate (query);            DML
Array[]  valores = st.executeBatch(query);          SQL Multiples
```

Pasos Generales JDBC

Procesar los resultados

- Para procesar los resultamos utilizamos el objeto ResultSet, para recorrer todos los resultados devueltos (tipo cursor)

- ResultSet proporciona diversos métodos getXXX que devuelven el resultado en una variedad de diferentes tipos de Java.
  - √ getInt → número entero
  - √ getString → Cadena de texto
  - √ getDate → Fechas

- Ahora utilizaremos este objeto para insertarle una Consulta SQL y ejecutarla

```
while(resultSet.next()) {
        System.out.println ( resultSet.getString(1) + " " +  resultSet.getString(2) + " " +
                resultSet.getString("nombre") + " " + resultSet.getString("apellido") );
}
```

Valor_c1 – 1	Valor_c2 – 1	Valor_nombre – 1	Valor_apellido – 1
Valor_c1 – 2	Valor_c2 – 2	Valor_nombre – 2	Valor_apellido – 2
Valor_c1 – 3	Valor_c2 – 3	Valor_nombre – 3	Valor_apellido – 3

Pasos Generales JDBC

Cerrar la conexión

- Cuando cerramos la conexión, todos los objetos del tipo *Statement* y *ResultSet* son eliminados.

- S queremos volver a comunicarnos con la BBDD después de haber realizado el cierre de una conexión, deberemos realizar todo el proceso completo de nuevo.

- Sintaxis

```
connection.close();
```

OPERACIONES BÁSICAS CON HIBERNATE

Elementos Básicos de Hibernate

- La forma más sencilla de comenzar con Hibernate es trabajar con una simple clase de Java y una única tabla en Base de Datos.
- Elementos básicos:
  - Estructura de datos disponible para su utilización (Base de Datos, Tablas, etc).
  - (opcional) Java Bean para la realización de peticiones o consultas.
  - Fichero de mapeado XML para el acceso a la BBDD (tabla.hbm.xml)
  - Fichero XML de configuración de Hibernate (hibernate.cfg.xml)
  - Clase Principal de Java para realizar las peticiones (a través de Java Beans)

*Para la explicación utilizaremos la TABLA PROFESOR creada por el **script_profesor.sql***

Elementos Básicos de Hibernate

- Elementos básicos:
  1. Estructura de datos disponible para su utilización (Base de Datos, Tablas, etc).

  Utilizaremos la Tabla PROFESOR creada por el ***script_profesor.sql***, con la siguiente estructura.

```
create table Profesor (
        Id integer not null,
        nombre varchar(255),
        ape1 varchar(255),
        ape2 varchar(255),
        primary key (Id)
    );
```

```
mysql> desc profesor
    -> ;
+--------+--------------+------+-----+---------+-------+
| Field  | Type         | Null | Key | Default | Extra |
+--------+--------------+------+-----+---------+-------+
| Id     | int(11)      | NO   | PRI | NULL    |       |
| nombre | varchar(255) | YES  |     | NULL    |       |
| ape1   | varchar(255) | YES  |     | NULL    |       |
| ape2   | varchar(255) | YES  |     | NULL    |       |
+--------+--------------+------+-----+---------+-------+
```


48

Elementos Básicos de Hibernate

- Elementos básicos:

  2.- (opcional) Java Bean para la realización de peticiones o consultas.

  - La clase Java que vamos a utilizar como entidad o documento para las operaciones DML/DQL, deberá seguir los estándar de creación de Java Beans.
    - √ Debe ser publica (no puede ser estar anidada ni final o tener miembros finales)
    - √ Deben tener un constructor público sin ningún tipo de argumentos.
    - √ Para cada propiedad que queramos persistir debe haber un método ***get/set*** asociado.
    - √ Debe tener una clave primaria
    - √ Debería sobrescribir los métodos equals y hashCode
    - √ Opcionalmente (para utilizar de forma remota), implementar el interfaz ***Serializable***
  - Habitualmente, el nombre de la clase de Java es igual a la tabla que vamos hacer referencia.

50

Elementos Básicos de Hibernate

- Elementos básicos:

  2.- (opcional) Java Bean para la realización de peticiones o consultas.

Fichero Profesor.java

→ new Package
→ new Class → Profesor

```
public class Profesor implements Serializable  {
 private int id;
 private String nombre;
 private String ape1;
 private String ape2;

 public Profesor(){ }

public Profesor(int id, String nombre, String ape1, String ape2) {
this.id = id;
this.nombre = nombre;
this.ape1 = ape1;
this.ape2 = ape2;
}

public int getId() { return id; }

public void setId(int id) { this.id = id; }
```

```
public String getNombre() { return nombre; }

public void setNombre(String nombre) { this.nombre = nombre; }

public String getApe1() { return ape1; }

public void setApe1(String ape1) { this.ape1 = ape1; }

public String getApe2() { return ape2; }

public void setApe2(String ape2) { this.ape2 = ape2; }
}
```

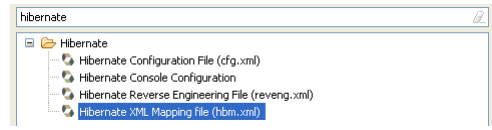

Elementos Básicos de Hibernate

- Elementos básicos:

  3.- Fichero de mapeado XML para el acceso a la BBDD (*Profesor.hbm.xml*)

  - Para cada clase que queremos persistir se creará un ***fichero .hbm.xml*** con la información que permitirá mapear la clase a una base de datos relacional.
  - Este fichero estará en el mismo paquete que la clase a persistir.
  - En nuestro caso, si queremos persistir la clase Profesor deberemos crear el fichero Profesor.hbm.xml en el mismo paquete que la clase Java.
  - El fichero tiene una configuración XML

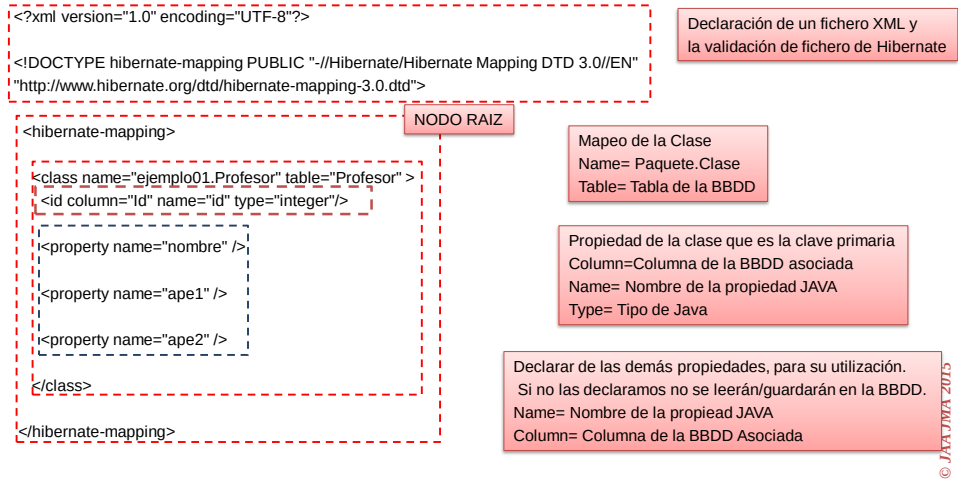
NEW→ Others → Find Hibernate →



Elementos Básicos de Hibernate

- Elementos básicos:

  3.- Fichero de mapeado XML para el acceso a la BBDD (*Profesor.hbm.xml*)



Elementos Básicos de Hibernate

- Elementos básicos:

  4.- Fichero XML de configuración de Hibernate (hibernate.cfg.xml)

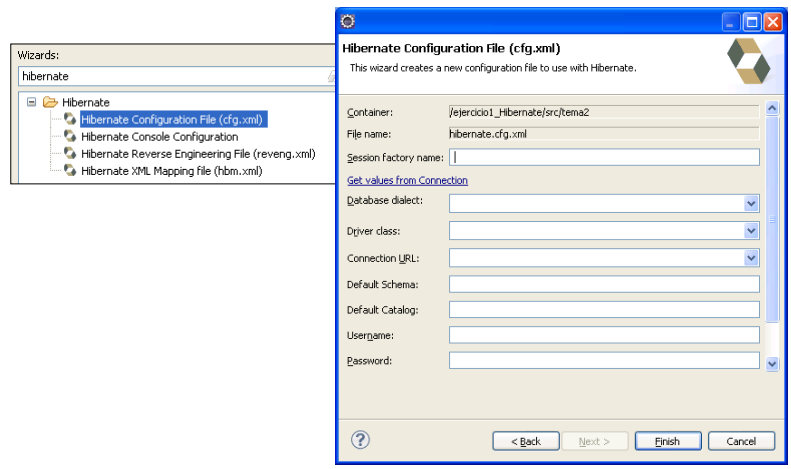
  - Para poder realizar la conexión con la BBDD, necesitamos un fichero donde aparezca dicha información ***fichero hibernate.cfg.xml*** .
  - Este fichero deberemos guardarlo en el paquete raíz de nuestras clases Java, es decir fuera de cualquier paquete.
  - La información que contiene es la siguiente:
    - √ Propiedades de configuración.
      - Driver, usuario, password, Lenguaje de comunicación, etc
    - √ Las clases que se quieren mapear.
  - Su ubicación debe de ser en los directorios ***web-inf*** o ***src***

Elementos Básicos de Hibernate

- Elementos básicos:

  4.- Fichero XML de configuración de Hibernate (hibernate.cfg.xml)

NEW→ Others → Find Hibernate →



55

Elementos Básicos de Hibernate

- Elementos básicos:

  4.- Fichero XML de configuración de Hibernate (hibernate.cfg.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">
```

Declaración de un fichero XML y la validación de fichero de Hibernate

RAIZ Documento

```
<hibernate-configuration>
  <session-factory>

    <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="connection.url">jdbc:mysql://localhost/hibernate1</property>
    <property name="connection.username">hibernate1</property>
    <property name="connection.password">hibernate1</property>
    <property name="dialect">org.hibernate.dialect.MySQL5Dialect</property>
    <property name="hibernate.show_sql">true</property>

    <mapping resource="ejemplo01/Profesor.hbm.xml"/>
    <mapping class="ejemplo01.Profesor"/>

  </session-factory>
</hibernate-configuration>
```

contienen propiedades de la configuración de conexión:
- driver, url, usuario, clave

Fichero .hbm.xml asociada a la clase que queremos persistir.

contiene el nombre de la clase que queremos persistir.

56

Elementos Básicos de Hibernate

```
<?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.bytecode.use_reflection_optimizer">false</property>
        <property name="hibernate.connection.driver_class">oracle.jdbc.driver.OracleDriver</property>
        <property name="hibernate.connection.url">jdbc:oracle:thin:@localhost:1521:xe</property>
        <property name="hibernate.connection.password">alumno</property>
        <property name="hibernate.connection.username">alumno</property>
        <property name="hibernate.default_schema">ALUMNO</property>
        <property name="hibernate.dialect">org.hibernate.dialect.Oracle10gDialect</property>
        <property name="hibernate.search.autoregister_listeners">false</property>
        <property name="hibernate.validator.apply_to_ddl">false</property>
    </session-factory>
</hibernate-configuration>
```


Elementos Básicos de Hibernate

Propiedad	Descripción
connection.driver_class	- Driver de conexión con la BBDD
connection.url	- La URL de conexión a la base de datos tal y como se usa en JDBC
connection.username	- Usuario de la BBDD
connection.password	- Clave del usuario en la BBDD
dialect	- Especificación del Lenguaje SQL que usará Hbernate contra la BBDD. - Parámetro opcional, Hibernate puede "Averiguarlo"
hibernate.show_sql	- Indica si se mostrará por la consola la orden SQL que Hibernate contra la base de datos. - Su posibles valores son true o false. Esta propiedad es muy útil mientras programamos ya que nos ayudará a entender cómo está funcionando Hibernate
connection.datasource	- Indica el nombre del DataSource con el que se conectará Hibernate a la base de datos. - Esta propiedad se usa en aplicaciones Web ya que los datos de la conexión se definen en el servidor de aplicaciones y se accede a la base de datos a través del DataSource

Elementos Básicos de Hibernate

- Elementos básicos:

  4.- Fichero XML de configuración de Hibernate (hibernate.cfg.xml)

  TAG <mapping>

  √ Para indicar que clase queremos mapear y contra qué tabla utilizamos el TAG <mapping>

  √ Lo habitual es utilizar el fichero de mapeo utilizado anteriormente (tabla.hbm.xml) utilizando el atributo resource:

  <mapping resource="ejemplo01/Profesor.hbm.xml"/>

  √ Para mapear la clase utilizamos el atributo class:

  <mapping class="ejemplo01.Profesor"/>

Elementos Básicos de Hibernate

- Elementos básicos:

  5.- Clase Principal de Java para realizar las peticiones

  - Para poder dar funcionalidad a todo esto, necesitaremos 3 elementos:
    - √ Objeto ***Session*** de la Clase ***org.hibernate.Session***.
    - √ Objeto ***SessionFactory*** de la Clase ***org.hibernate.SessionFactory***.
    - √ Objeto ***Configuration*** de la Clase ***org.hibernate.cfg.Confguration***.
  - La clase ***Session*** contiene métodos para leer, guardar o borrar entidades sobre la base de datos.
  - La Clase ***SessionFactory*** nos permite crear un objeto ***Session*** para toda la comunicación.
  - La Clase ***Configuration*** nos permite leer el fichero de configuración de Hibernate *hibernate.cfg.xml*

Elementos Básicos de Hibernate

- Elementos básicos:

  5.- Clase Principal de Java. **Creación del Objeto SessionFactory**

  - La primera operación a realizar es la creación del Objeto SessiónFactory.
  - Este objeto es de creación única por aplicación y lo podremos reutilizar en todas las llamadas que necesitemos.
  - Hay 2 posibilidades de creación (como ahora veremos), una < Hibernate 4 y otra post Hibernate 4 y superiores.

61

Elementos Básicos de Hibernate

- Elementos básicos:

  5.- Clase Principal de Java. **Creación del Objeto SessionFactory**

```
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;
import org.hibernate.service.ServiceRegistry;
import org.hibernate.service.ServiceRegistryBuilder;

SessionFactory sessionFactory;

Configuration configuration = new Configuration();
configuration.configure();
```

Deprecated

```
sessionFactory = configuration.buildSessionFactory();
```

```
ServiceRegistry serviceRegistry = new
                ServiceRegistryBuilder().applySettings(configuration.getProperties()).buildServiceRegistry();
sessionFactory = configuration.buildSessionFactory(serviceRegistry);
:
session.close();
```


62

HibernateUtil

```
public class HibernateUtil {
    private static final SessionFactory sessionFactory = buildSessionFactory();

    private static SessionFactory buildSessionFactory() {
        // A SessionFactory is set up once for an application!
        final StandardServiceRegistry registry = new StandardServiceRegistryBuilder()
                .configure() // configures settings from hibernate.cfg.xml
                .build();
        try {
         return new MetadataSources(registry ).buildMetadata().buildSessionFactory();
        }
        catch (Exception ex) {
         System.err.println("Initial SessionFactory creation failed." + ex);
         throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }
}
```


63

HibernateUtil (Ver. 5.x)

```
public class HibernateUtil {
    private static final SessionFactory sessionFactory = buildSessionFactory();

    private static SessionFactory buildSessionFactory() {
        final StandardServiceRegistry registry = new StandardServiceRegistryBuilder()
                .configure() // configures settings from hibernate.cfg.xml (JndiException: name="")
                .build();
        try {
         return new MetadataSources( registry ).buildMetadata().buildSessionFactory();
        }
        catch (Exception ex) {
         // The registry would be destroyed by the SessionFactory, but we had trouble building
         // the SessionFactory so destroy it manually.
         StandardServiceRegistryBuilder.destroy( registry );
         System.err.println("Initial SessionFactory creation failed." + ex);
         throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }
}
```


64

Elementos Básicos de Hibernate

- Elementos básicos:

  5.- Clase Principal de Java. **Creación del Objeto Session**

  - Una vez creado el Objeto SessionFactory podemos obtener la ***Session*** para trabajar con Hibernate.
  - Es tan sencillo como llamar al método ***openSession()*** de sessionFactory.

```
Session session = HibernateUtil.getSessionFactory().openSession();
```

  - Una vez obtenida la sesión trabajaremos con Hibernate persistiendo las clases y una vez finalizado se deberá cerrar la sesión con el método ***close():***

```
session.close();
```


Elementos Básicos de Hibernate

- Elementos básicos:

  5.- Clase Principal de Java. **Creación del Objeto SessionFactory**

```
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;
import org.hibernate.service.ServiceRegistry;
import org.hibernate.service.ServiceRegistryBuilder;

SessionFactory sessionFactory;

Configuration configuration = new Configuration();
configuration.configure();
 ServiceRegistry serviceRegistry = new
                ServiceRegistryBuilder().applySettings(configuration.getProperties()).buildServiceRegistry();
 sessionFactory = configuration.buildSessionFactory(serviceRegistry);

 Session session = sessionFactory.openSession();

:
session.close();
```

Elementos Básicos de Hibernate

- Elementos básicos:

  5.- Clase Principal de Java. **Transacciones**

  - Para poder trabajar con la Base de Datos, necesitamos utilizar transacciones.
  - En Hibernate disponemos de 4 operaciones básicas sobre transacciones:

    √ Creación de una transacción
    session.beginTransaction();

    √ Validación de la transacción actual.
    session.getTransaction().commit();

    √ Rollback de la transacción actual.
    session.getTransaction().rollback();

    √ Cierre de la transacción actual.
    session.closeTransaction();

Elementos Básicos de Hibernate

- Elementos básicos:

  5.- Clase Principal de Java. **Transacciones**

```
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;
import org.hibernate.service.ServiceRegistry;
import org.hibernate.service.ServiceRegistryBuilder;

SessionFactory sessionFactory;

Configuration configuration = new Configuration();
configuration.configure();
 ServiceRegistry serviceRegistry = new
            ServiceRegistryBuilder().applySettings(configuration.getProperties()).buildServiceRegistry();
 sessionFactory = configuration.buildSessionFactory(serviceRegistry);
 Session session = sessionFactory.openSession();
 Session.beginTransaction();
:
session.close();
```

Elementos Básicos de Hibernate

- Elementos básicos:

  5.- Clase Principal de Java. **Operaciones CRUD**

  - Entendemos por Operaciones CRUD las operaciones básicas que se realizan en toda tabla:
    - √ **C**reate: Insertar un nuevo objeto en la base de datos.
    - √ **R**ead: Leer los datos de un objeto de la base de datos.
    - √ **U**pdate: Actualizar los datos de un objeto de la base de datos.
    - √ **D**elete: Borrar los datos de un objeto de la base de datos.

  - Cada una de estas operaciones tiene asignado un método del objeto *Session* para su realización.
    - √ **C**reate: session.**save**(…..);
    - √ **R**ead: session.**get**(….);
    - √ **U**pdate: session.**update**(…..);
    - √ **D**elete: session.**delete**(….);
    - √ Actualizar o Insertar

69

Elementos Básicos de Hibernate

- Elementos básicos:

  5.- Clase Principal de Java. **Operaciones CRUD**

  - Lectura
    - √ Utilizaremos el método *get (Class, ID)* para obtener información del elemento guardado en la Tabla y que tenga ese número de Identificación.
    - √ Este método permite SOLO leer un UNICO Objeto
    - √ Necesitamos hacer un CAST para identificar el tipo de objeto.

```
:
Session session = sessionFactory.openSession();
session.beginTransaction();

Profesor profesor=(Profesor) session.get(Profesor.class, 101);
System.out.println(profesor.getNombre());
session.close()
```

SI error parsing JNDI hibernate name

Eliminar el elemento name del fichero hibernate.cfg.xml

<session-factory name=" "> → <session-factory >

70

Lenguajes de consultas

- En Hibernate existen 3 maneras para realizar consultas.
  - HQL (Hibernate Query Lenguage)
    session.createQuery("from Category c where c.name like 'Laptop%'");
    entityManager.createQuery("select c from Category c where c.name like 'Laptop%‘ ");
  - CRITERIA API
    session.createCriteria(Category.class).add(Restrictions.like("name", "Laptop%"))
  - Direct SQL
    session.createSQLQuery("select {c.*} from CATEGORY {c} where NAME like 'Laptop%‘ ").addEntity("c", Category.class);

Elementos Básicos de Hibernate

- Elementos básicos:
  5.- Clase Principal de Java. **Operaciones CRUD**
  - Lectura múltiple (consulta)
    - √ La consultas se crean con la sesión.
    - √ Se materializan con el método list()
    - √ La colección resultante estará vacía si la consulta no produce resultados.

```
:
Session session = sessionFactory.openSession();
Query<Profesor> consulta = session.createQuery("from Profesor");
List<Profesor> rslt = consulta.list();
for(Profesor profesor : rslt){
   System.out.println(profesor.getNombre());
}
session.close()
```

SI error parsing JNDI hibernate name

Eliminar el elemento name del fichero hibernate.cfg.xml

<session-factory name=" "> → <session-factory >

Elementos Básicos de Hibernate

- Elementos básicos:

  5.- Clase Principal de Java. **Operaciones CRUD**

  - Creación / Inserción
    - √ Utilizaremos el método ***save (objeto)*** con el elemento a guardar.
    - √ El objeto será/deberá de ser igual a un registro de la tabla que tengamos mapeado.
    - √ Siempre deberemos validar la transacción antes de su finalización.

```
:
Profesor profesor=new Profesor(101, "Adriana", "Sanchez", "Meneses");  //Creamos el objeto
 Session session = sessionFactory.openSession();
session.beginTransaction();

 session.save(profesor);

session.getTransaction().commit();
session.close()
```

SI error parsing JNDI hibernate name

Eliminar el elemento name del fichero hibernate.cfg.xml

<session-factory name=" "> → <session-factory >

Elementos Básicos de Hibernate

- Elementos básicos:

  5.- Clase Principal de Java. **Operaciones CRUD**

  - Actualización
    - √ Utilizaremos el método ***update(Object objeto)*** con el objeto a actualizar en la Base de Datos.
    - √ Este método permite SOLO actualizar un UNICO Objeto

```
:
Session session = sessionFactory.openSession();
session.beginTransaction();
Profesor profesor=(Profesor) session.get(Profesor.class, 101);
    profesor.setNombre("Pedro"));
    session.update(profesor);
    session.getTransaction().commit();
session.close()
```

SI error parsing JNDI hibernate name

Eliminar el elemento name del fichero hibernate.cfg.xml

<session-factory name=" "> → <session-factory >

Elementos Básicos de Hibernate

- Elementos básicos:

  5.- Clase Principal de Java. **Operaciones CRUD**

  - Borrado
    - √ Utilizaremos el método *delete (Object objeto)* con el objeto a borrar en la Base de Datos.
    - √ Este método permite SOLO actualizar un UNICO Objeto

```
:
Session session = sessionFactory.openSession();
session.beginTransaction();
Profesor profesor=(Profesor) session.get(Profesor.class, 101);
session.delete(profesor);
    session.getTransaction().commit();
session.close()
```

SI error parsing JNDI hibernate name

Eliminar el elemento name del fichero hibernate.cfg.xml

<session-factory name=" "> → <session-factory >

Elementos Básicos de Hibernate

- Elementos básicos:

  5.- Clase Principal de Java. **Operaciones CRUD**

  - Inserción o Actualizar
    - √ Hay veces que es útil realizar una Inserción o una Actualización en función de si el registro existe o no.
    - √ Disponemos del método *saveOrUpdate (objeto)* para realizar dichas acciones.
    - √ Siempre deberemos validar la transacción antes de su finalización.

```
:
Profesor profesor=new Profesor(112, “Adriana", “Sanchez", “Meneses");  //Creamos el objeto
 Session session = sessionFactory.openSession();
session.beginTransaction();

 session.saveOrUpdate(profesor);

session.getTransaction().commit();
session.close()
```

SI error parsing JNDI hibernate name

Eliminar el elemento name del fichero hibernate.cfg.xml

<session-factory name=" "> → <session-factory >

Anotaciones JPA

- Mediante los ficheros *.hbm.xml* podemos especificar cómo mapear la clases Java en tablas de base de datos.
- Pero….. cuando el desarrollo es grande, el número de ficheros *.hbm.xml* puede ser desmesurado.
- ¿Cómo puedo hacer el mapeo sin utilizar ficheros XML?
- Solución: Uso de Anotaciones Java
  - Estas anotaciones permiten especificar de una forma compacta y sencilla la información de mapeo de las clases Java

Anotaciones JPA

- Hibernate dispone de sus propias anotaciones en el paquete *org.hibernate.annotations* *(a partir de la versión 4, la mayoría de dichas anotaciones han sido marcadas como java.lang.Deprecated y ya no deben usarse)*
- Las anotaciones que deben usarse actualmente son las del estándar de JPA que se encuentran en el paquete *javax.persistence.*
- Sin embargo hay características específicas de Hibernate que no posee JPA lo que hace que aun sea necesario usar alguna anotación del paquete org.hibernate.annotations.

  (dichas anotaciones no han sido marcadas como Deprecated en Hibernate 4)

Anotaciones JPA

Anotación	Descripción
@Entity	- Se aplica a la clase. - Indica que esta clase Java es una entidad a persistir.
@Table(name="Tabla")	- Se aplica a la clase e indica el nombre de la tabla de la base de datos donde se persistirá la clase. - Es opcional si el nombre de la clase coincide con el de la tabla.
@Id	- Se aplica a una propiedad Java e indica que este atributo es la clave primaria.
@Column(name="Id")	- Se aplica a una propiedad Java e indica el nombre de la columna de la base de datos en la que se persistirá la propiedad. - Es opcional si el nombre de la propiedad Java coincide con el de la columna de la base de datos.
@Column(…)	- name: nombre - length: longitud - precision: número total de dígitos - scale: número de digitos decimales - unique: restriccion valor unico - nullable: restriccion valor obligatorio - insertable: es insertable - updatable: es modificable
@Transient	- Se aplica a una propiedad Java e indica que este atributo no es persistente

Anotaciones JPA

```
Import javax.persistence.*;

@Entity
@Table(name="Profesor")
public class Profesor implements Serializable  {

@Id
@Column(name="Id")
 private int id;

@Column(name="nombre")
 private String nombre;

@Column(name="ape1")
 private String ape1;

@Column(name="ape2")
 private String ape2;

 public Profesor(){ }

public Profesor(int id, String nombre, String ape1, String ape2) {
this.id = id;
this.nombre = nombre;
this.ape1 = ape1;
this.ape2 = ape2;
}
```

```
public int getId() { return id; }

public void setId(int id) { this.id = id; }

public String getNombre() { return nombre; }

public void setNombre(String nombre) { this.nombre = nombre; }

public String getApe1() { return ape1; }

public void setApe1(String ape1) { this.ape1 = ape1; }

public String getApe2() { return ape2; }

public void setApe2(String ape2) { this.ape2 = ape2; }
}
```

Anotaciones JPA

- Para migrar de un proyecto Sin Anotaciones a uno con Anotaciones JPA deberemos:
  1. Modificar el fichero de configuración de Hibernate (hibernate.cfg.xml)
     Todos los elementos de mapeos **resource** deben de ser eliminados

```
<mapping class="tema2.Profesor"/>
<mapping resource="tema2/Profesor.hbm.xml"/>
```

  2. Modificar el la clase Principal de Java utilizado para crear el objeto *sessionFactory*.
     Crear el objeto *sessionFactory* de la siguiente forma:

```
Configuration configuration = new Configuration(); configuration.configure();
ServiceRegistry serviceRegistry = new
        ServiceRegistryBuilder().applySettings(configuration.getProperties()).buildServiceRegistry();
sessionFactory = configuration.buildSessionFactory(serviceRegistry);

sessionFactory = new AnnotationConfiguration().configure().buildSessionFactory();
```


Anotaciones JPA

NOTAS

- El/los Fichero/s de mapeo *Tabla.hbm.xml* es obligatorio indicar todas las propiedades que queremos que se persistan en la base de datos.
- En las anotaciones no es necesario, persisten todas las propiedades que tengan los métodos get/set.

- Si colocamos las anotaciones sobre las propiedades, el acceso será a las propiedades y no serán necesarios los métodos get/set.

- Si colocamos las anotaciones sobre los métodos get() , el acceso será mediante los métodos get/set.

Java Persistence API (JPA)

- JPA define un proceso de arranque diferente que usa su propio archivo de configuración llamado persistence.xml distinto del archivo hibernate.cfg.xml.
- Este proceso de arranque está definido por la especificación JPA.
- En entornos JavaSE, el proveedor de persistencia (Hibernate en este caso) debe ubicar todos los archivos de configuración JPA en el classpath con el nombre del recurso de META-INF/persistence.xml (src/main/resources/META-INF)
- Los archivos persistence.xml deben proporcionar un nombre único para cada "unidad de persistencia".
- Las aplicaciones utilizan este nombre para hacer referencia a la configuración al obtener una referencia javax.persistence.EntityManagerFactory.

persistence.xml

- Crear directorio
  - src/main/resources/META-INF
- Crear fichero persistence.xml

```
<persistence version="2.1"
   xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
   xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
   <persistence-unit name="TestPersistence" transaction-type="RESOURCE_LOCAL">
      <class><!-- Entity Class Name --></class>
      <properties>
         <property name="javax.persistence.jdbc.driver" value="Database Driver Name" />
         <property name="javax.persistence.jdbc.url" value="Database Url" />
         <property name="javax.persistence.jdbc.user" value="Database Username" />
         <property name="javax.persistence.jdbc.password" value="Database Password" />
      </properties>
   </persistence-unit>
</persistence>
```

EntityManagerFactory

- La clase EntityManagerFactory nos permite crear un objeto EntityManager para toda la comunicación, recibe el nombre de la "unidad de persistencia".
  - EntityManagerFactory emf = Persistence.createEntityManagerFactory("dbPersistence");
- La clase EntityManager contiene métodos para leer, guardar o borrar entidades sobre la base de datos.
  - EntityManager em = emf.createEntityManager();
- En JEE con un contenedor EJB:
  - @PersistenceContext(unitName = "dbPersistence")
  - private EntityManager em;
- Para cerrar el gestor de entidades:
  - em.close();
- Para obtener el Session desde el EntityManager:
  - Session session = em.unwrap(Session.class);

Transacciones con EntityManager

- Creación de una transacción
  - em.getTransaction().begin();
- Validación de la transacción actual.
  - em.getTransaction().commit();
- Rollback de la transacción actual.
  - em.getTransaction().rollback();
- Consultar si la transacción esta activa.
  - if(em.getTransaction().isActive())

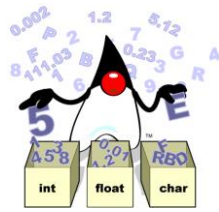
Operaciones CRUD con EntityManager

- Create:
  - em.persist(…..);
- Read:
  - em.find(….);
- Update (adjuntar):
  - em.merge(…..); // Entidad detach
- Delete:
  - em.remove(….);
- Persistencia sin transacciones (sincroniza con la base de datos):
  - em.flush();
- Gestión a través de “entidades administradas”:
  - Separar del contenedor: em.detach(…);
  - Buscar en el contenedor: em.contains(…);
  - Borrar contenedor: em.clear();

TIPOS DE DATOS Y CLAVES PRIMARIAS

Tipos Básicos

- Cuando trabajamos con un ORM, éste debe de ser capaz de trabajar con los diferentes tipos de datos existentes:
  - √ Tipos de datos en JAVA
  - √ Tipos de datos en SQL
- ORM debe de ser capaz de crear un puente entre ambos, totalmente transparente para el desarrollador.



93

Tipos Básicos

- En toda aplicación que utiliza Hibernate, podemos diferenciar 3 tipos de datos:
  - Tipos de Datos de Java → Ficheros .java
  - Tipos de Datos de SQL → Columnas de las tablas de la BBDD
  - Tipos de Datos de Hibernate → Ficheros de mapeo .hbm.xml
- Estos *tipos de datos Hibernate*, son utilizados como conversores para poder trasladar tipos de datos Java a SQL y viceversa.
- Hibernate intentara determinar la conversión correcta y el tipo de datos si el atributo *type* no se encuentra presente.

```
<property name="date" type="timestamp" column="EVENT_DATE"/>
 <property name="title"/>
```


94

Tipos Básicos

- Si no se especifica el tipo Hibernate, se produce una *detección automática*, que es determinada mediante reflexión cuando se procesan los ficheros de mapeo.

- La detección automática, puede escoger un tipo que no es el que se esperaba o necesitaba.
  - √ Propiedad del tipo java.util.Date → ???? date ?
    timestamp?
    time ?

- La detección automática, también conlleva un tiempo y recursos extras.

Tipos Básicos

Tipo Hibernate	Tipo Java	Tipo base de datos (MySQL)
integer	int , `java.lang.Integer`	INTEGER
long	long, `java.lang.Long`	BIGINT
short	short , `java.lang.Short`	SMALLINT
float	float , `java.lang.Float`	FLOAT
double	double , `java.lang.Double`	DOUBLE
character	char , `java.lang.Character`	CHAR
byte	byte , `java.lang.Byte`	TINYINT
boolean	boolean , `java.lang.Boolean`	TINYINT. Guarda el `true` como `"1"` y el `false` como `"0"`
yes_no	boolean , `java.lang.Boolean`	CHAR(1). Guarda el `true` como `"Y"` y el `false` como `"N"`
true_false	boolean , `java.lang.Boolean`	CHAR(1). Guarda el `true` como `"T"` y el `false` como `"F"`

Tipos Básicos

Tipo Hibernate	Tipo Java	Tipo base de datos (MySQL)
string	java.lang.String	VARCHAR
date	java.util.Date	DATE. Solo se guarda solo la información de la fecha (año,
time	java.util.Date	TIME. Solo se guarda solo la información de la hora (horas,
timestamp	java.util.Date	DATETIME. Se guarda la información de la fecha y la hora (minutos y segundos)
text	java.lang.String	LONGTEXT
binary	byte[]	TINYBLOB
big_decimal	java.math.BigDecimal	DECIMAL
big_integer	java.math.BigInteger	DECIMAL

97

Tipos Básicos

Fecha y Hora

- Cuando trabajamos con campos de Fecha y Hora deberemos tener cuidado con la información que queremos tratar.

- Hay veces que sólo nos interesa:
  - √ Fecha sin hora, minutos ni segundos
  - √ Hora, sin día, mes, año
  - √ Hora, sin milisegundos, microsegundos, etc…

- Deberemos seguir los siguientes criterios para el buen uso de estos tipos de datos:

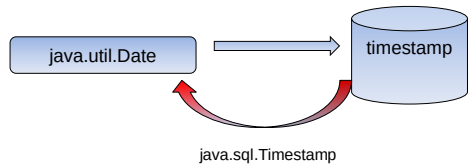
Mapping type	Java type	Standard SQL built-in type	
date	java.util.Date or java.sql.Date	DATE	Día, Mes, Año
time	java.util.Date or java.sql.Time	TIME	Hora, Minuto, Segundo
timestamp	java.util.Date or java.sql.Timestamp	TIMESTAMP	Timestamp
calendar	java.util.Calendar	TIMESTAMP	Hora, Minuto, Segundo
calendar_date	java.util.Calendar	DATE	Día, Mes, Año

98

Tipos Básicos

Fecha y Hora

- Deberemos tener cuidado con:
  - √ Si mapeamos una Variable Java (java.util.Date) con una columna Timestamp, Hibernate devuelve un dato java.sql.Timestamp



  - √ Debemos de tener cuidado al utilizar el método ***equal()*** pues se producirá una excepción *objeto_Date.equal(Objeto_java.sql.Timestamp)*

- Si podremos realizar comparativas del modo:

  aDate.getTime() > bDate.getTime()

Tipos Básicos

Boolean

- Hibernate permite 3 formas distintas de almacenar un booleano de Java en la base de datos.

- Para ello existen 3 tipos de datos en Hibernate.

  - √ boolean:
    - – Es la forma estándar de guardar un booleano en la base de datos.

  - √ yes_no:
    - – El valor se guardará como un CHAR(1) con los valores de Y y N para true y false respectivamente.

  - √ true_false:
    - – El valor se guardará como un CHAR(1) con los valores de T y F para true y false respectivamente.

@Temporal

- Mediante anotaciones

```
@Temporal(TemporalType.DATE)
private Date fecha;
@Temporal(TemporalType.TIME)
private Date hora;
@Temporal(TemporalType.TIMESTAMP)
private Date momento;
```

- Java 8 dispone de los nuevos tipos específicos soportados por Hibernate:
  - FECHA
    - √ java.time.LocalDate
  - HORA
    - √ java.time.LocalTime, java.time.OffsetTime
  - TIMESTAMP
    - √ java.time.Instant, java.time.LocalDateTime, java.time.OffsetDateTimeYjava.time.ZonedDateTime

Tipos Básicos

Texto

- Hibernate permite 2 formas distintas de almacenar un *java.lang.String* de Java en la base de datos.

- Para ello existen 2 tipos de datos en Hibernate:
  - √ string:
    Se guardará el java.lang.String como un VARCHAR en la base de datos.
  - √ text:
    Se guardará el java.lang.String como un CLOB o TEXT ,etc. en la base de datos.

Tipos Básicos

Datos Binarios y Valores Grandes

- Si una propiedad en su clase Java persistente es de tipo ***byte []***, Hibernate puede asignarlo a una columna VARBINARY dentro de la Base de Datos.
  - √ El tipo VARBINARY depende del SGBDR
    - PostgreSQL → BYTEA
    - Oracle → RAW

- Si una propiedad en la clase Java es del tipo ***java.lang.String***, y el tipo de Hibernate es TEXT, puede asignarlo a una columna CLOB SQL en la BBDD.

- Si desea asignar un *java.lang.String*, *char [], Character [],* a una columna CLOB, es necesario hacer una anotacion @Lob

```
@Lob
@Column(name = "ITEM_DESCRIPTION")
private String description;
```


Claves Primarias

- En el diseño de bases de datos relacionales, se llama clave primaria a un columna o combinación de columnas que identifica de forma única a cada fila de una tabla.

- Una clave primaria debe identificar unívocamente a todas las posibles filas de una tabla actuales y futuras.

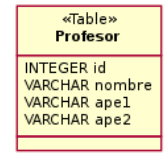
- Propiedades que debe de cumplir:
  - √ Debe ser única
  - √ No puede ser NULL
  - √ A ser posible, nunca debería cambiar
  - √ Debe ser corta
  - √ Debe ser rápida de generar

- En el modelo entidad-relación, la clave primaria permite las relaciones entre tablas.

Claves Primarias

- PK Naturales:
  - Una clave primaria natural es aquella columna/s de base de datos que actúa de clave primaria en el modelo de negocio en el que estamos trabajando.

    DNI, Número SS, Número de Cliente, etc
- PK Artificiales:
  - Son aquellas claves que sin pertenecer al modelo de negocio hay que crear para que todas las filas tengan un identificador, las "ideales" son las autonuméricas.
- Si es posible, para Hibernate, es mejor utilizar Claves Primarias Autonuméricas.
- Estas PK pueden ser creadas de forma automática en Hibernate mediante el TAG ***<generator>*** en el fichero .hbm.xml.
- Si Hibernate genera la clave primaria nos ahorramos tener que incluirla en el constructor.



Claves Primarias

- Para definir las Claves primarias en Hibernate lo hacemos en el fichero de mapeo de la clase Java donde esté la PK
  - √ Introduciremos el TAG <generator> junto con la forma de su generación
  - √ <generator class="assigned" />

Profesor.hbm.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD
3.0//EN" "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
   <class name="ejemplo02.Profesor" >
      <id column="Id" name="id" type="integer">
         <generator class="increment" />
      </id>
      <property name="nombre" />
      <property name="ape1" />
      <property name="ape2" />
   </class>

</hibernate-mapping>
```

TAG <generator>

Se utiliza para indicar que la clave primaria será generada por el propio Hibernate en vez de asignarla directamente el usuario.

class:

Indica el método que usará Hibernate para calcular la clave primaria.

Si es "assigned" es una clave natural que no se genera, se introduce manualmente.

Claves Primarias

Valor	Descripción
native	Hibernate usará alguno de los siguientes métodos dependiendo de la base de datos. De esta forma si cambiamos de base de datos se seguirá usando la mejor forma de generar la clave primaria
identity	Hibernate usará el valor de la columna de tipo autoincremento. Es decir, que al insertar la fila, la base de datos le asignará el valor. La columna de base de datos debe ser de tipo autonumérico
sequence	Se utiliza una secuencia como las que existen en Oracle o PostgreSQL , no es compatible con MySQL. La columna de base de datos debe ser de tipo numérico
increment	Se lanza una consulta `SELECT MAX()` contra la columna de la base de datos y se obtiene el valor de la última clave primaria, incrementando el nº en 1.La columna de base de datos debe ser de tipo numérico
uuid.hex	Hibernate genera un identificador único como un String. Se usa para generar claves primarias únicas entre distintas bases de datos.La columna de base de datos debe ser de tipo alfanumérico.
guid	Hibernate genera un identificador único como un String pero usando las funciones que provee SQL Server y MySQL. Se usa para generar claves primarias únicas entre distintas bases de datos.La columna de base de datos debe ser de tipo alfanumérico.
foreign	Se usará el valor de otro objeto como la clave primaria. Un uso de ello es en relaciones *uno a uno* donde el segundo objeto debe tener la misma clave primaria que el primer objeto a guardar.

```
<id column="Id" name="id" type="integer">
    <generator class="sequence" >
        <param name="sequence">secuencia_idProfesor</param>
    </generator>
</id>
```


107

Claves Primarias Compuestas

- Una tabla con clave compuesta se puede mapear con múltiples propiedades de la clase como propiedades identificadoras.
- El elemento <composite-id> acepta los mapeos de propiedad <key-property> y los mapeos <key-many-to-one> como elementos hijos.

```
<composite-id>
    <key-property name="idAsignatura"/>
    <key-property name="idAlumno"/>
</composite-id>
```

- La clase persistente tiene que sobrescribir equals() y hashCode() para implementar la igualdad del identificador compuesto. También tiene que implementar Serializable.

108

Claves Primarias. Anotaciones

- Para usar anotaciones deberemos modificar el código fuente de las clases Java y **no** los ficheros .hbm.xml.

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

    @Id
    @Column(name="Id")
    @GeneratedValue(strategy=GenerationType.AUTO)
    private int id;
:
```

@GeneratedValue:
Esta anotación indica que Hibernate deberá generar el valor de la clave primaria.

Strategy=GenerationType.XXXXXXX
IIndica el método en el que Hibernate debe generar la clave primaria
- AUTO
- IDENTITY
- SECUENCE

- JPA no soporta todos los tipos de generaciones de Hibernate.

109

Claves Primarias. Anotaciones

- Si queremos hacer uso de todos los métodos de generación de claves primarias de que dispone Hibernate mediante el uso de anotaciones, deberemos usar la anotación propietaria de Hibernate.
  - @GeneratedValue(generator = "*generador_propietario_hibernate_increment*")
  - @org.hibernate.annotations.GenericGenerator (…..)

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

   @Id
   @GeneratedValue( generator = "generador_propietario_hibernate_increment" )
   @org.hibernate.annotations.GenericGenerator(
     name = "generador_propietario_hibernate_increment",
     strategy = "increment"
     )
    private int id;
:
```


110

Claves Primarias Compuestas. Anotaciones

- Es necesario crear una clase auxiliar que reúna los elementos de la clave primaria y asociarla a la entidad.

```
@IdClass(ProfesorAsignaturaPK.class)
@Entity
@Table(name="ProfesorAsignatura")
public class ProfesorAsignatura implements Serializable {

    @Id
    @Column(name="IdProfesor")
    private int idProfesor;
    @Id
    @Column(name="IdAsignatura")
    private int idAsignatura;
```

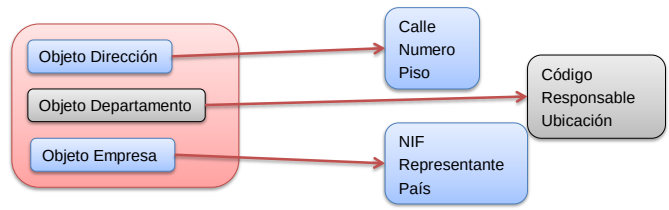
@IdClass:
Especifica la clase de clave principal compuesta que está asignado a múltiples campos o propiedades de la entidad.

Los nombres de los campos o propiedades de la clase de clave principal y los campos de clave principal o propiedades de la entidad deben corresponderse y sus tipos deben ser los mismos.

111

Componentes

- Lenguajes orientados a objetos como Java hacen más fácil definir objetos complejos como composición de otros ya definidos.
  - Nombre Cliente, Dirección, Departamento → Objeto cliente

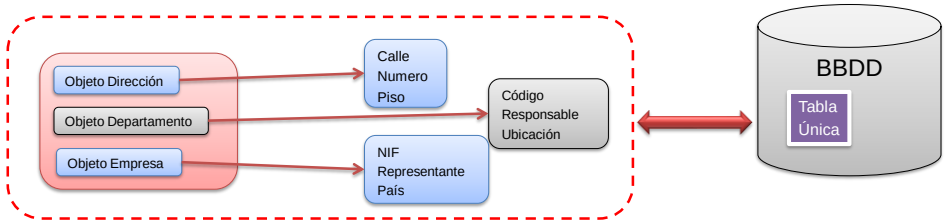


- Hibernate dispone de un elemento llamado Componente que utilizamos para expresar este tipo de datos complejos.

112

Componentes

- Los componentes son clases definidos por el usuario que son persistentes dentro de la propia tabla.
- Permiten que varias clases relacionadas se almacenen en una única tabla de la BBDD.



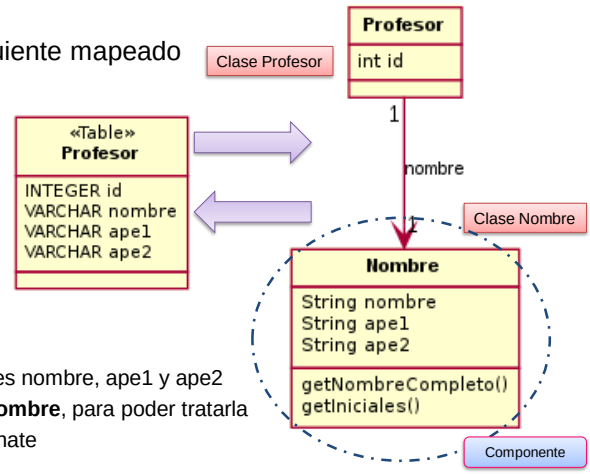
- Estos componentes son tratados como un «value type», como un tipo primitivo, String, etc

  Clase Dirección → String (Calle+Numero+Piso)
  - Los "value types" forman parte de una única clase

113

Componentes

- Podríamos realizar el siguiente mapeado



- Se han extraído las propiedades nombre, ape1 y ape2 en una nueva clase llamada **Nombre**, para poder tratarla como un componente en Hbernate
- Cuando hagamos Persistencia de Profesor y Nombre, se almacenarán dentro de Tabla Profesor de forma única.

114

Componentes

- Los componentes no requieren fichero de configuración *.hbm.xml*
- **El tag <component>** se utiliza para especificar que la propiedad Java de la clase se persistirá en la propia tabla de la clase principal.

Profesor.hbm.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping
DTD 3.0//EN" "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping>

 <class name="ejemplo04.Profesor" >
  <id column="Id" name="id" type="integer"/>

  <component name="nombre">
    <property name="nombre" />
    <property name="ape1" />
    <property name="ape2" />
  </component>

 </class>

</hibernate-mapping>
```


Componentes. Anotaciones

- Para usar anotaciones deberemos modificar el código fuente de las clases Java y no usar los ficheros .hbm.xml.
- Deberemos indicar la anotación *@Embedded* en la propiedad que sea el componente.

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

   @Id
   @Column(name="Id")
   private int id;

   @Embedded
   private Nombre nombre;
```

@Embedded:
Esta anotación se usa para indicar que la propiedad nombre se guardará en la misma tabla que Profesor.

Componentes. Anotaciones

- En la clase "Componente" deberemos introducir la anotación **@*Embeddable*** al comienzo de la clase, en vez de @Entity

```
@Embeddable
public class Nombre implements Serializable {

   @Column(name="nombre")
   private String nombre;

   @Column(name="ape1")
   private String ape1;

   @Column(name="ape2")
   private String ape2;
:
```

@Embeddable:
Se usa para indicar que esta clase se usará como un componente y que se guardará en la misma tabla que la clase que la posee..

Tipos Enumerados

- Los tipos enumerados en Java, permiten que una variable tenga un conjunto de valores restringidos.
- Son utilizados con valores fijos (meses del año, colores, etc)

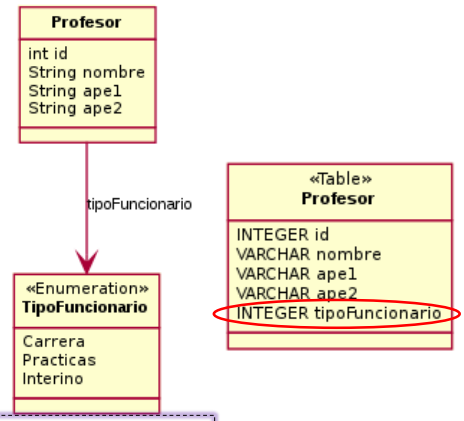
```
public enum TipoFuncionario { Carrera,    Practicas,    Interino };
public enum Colores { Blanco, Negro, Amarillo, Rojo, Azul };
```

- Hibernate no soporta directamente el persistir los enumerados.
- Pero dispone de mecanismos sencillos para persistir un enumerado.

Tipos Enumerados

- En el siguiente entorno:

```
public enum TipoFuncionario {
    Carrera,
    Practicas,
    Interino
}

public class Profesor implements Serializable  {

    private int id;
    private String nombre;
    private String ape1;
    private String ape2;
    private TipoFuncionario tipoFuncionario;
```



- Dentro de la Clase Profesor incluimos un dato Enumerado TipoFuncionario
- Dentro de la TABLA Profesor incluimos una columna INTEGER / VARCHAR

119

Tipos Enumerados

- Sólo será necesario un fichero de persistencia profesor.hbm.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD
3.0//EN" "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
   <class name="ejemplo06.Profesor" >
      <id column="Id" name="id" type="integer"/>
      <property name="nombre" />
      <property name="ape1" />
      <property name="ape2" />

      <property name="tipoFuncionario" >
         <type name="org.hibernate.type.EnumType">
            <param name="enumClass">ejemplo06.TipoFuncionario</param>
            <param name="type">4</param>
         </type>
      </property>
   </class>
</hibernate-mapping>
```

TAG <type>

Se utiliza para definir una forma personalizada de persistir una propiedad Java.

Type name:

Indica la clase Java que sabe cómo persistir una propiedad Java, en este caso los Enumerados.

Param name="enumClass">

Indica el Tipo Enumerado ha persistir

Param name ="type">

Debe de ser el tipo de persistencia a realizar:

- 1 → CHAR
- 2 → NUMERIC
- 4 → INTEGER
- 5 → SMALLINT
- 12 → VARCHAR

120

Tipos Enumerados

Valor numérico	Tipo SQL	Forma de persistencia
4	Types.INTEGER	Se almacena el **ordinal** del enumerado
2	Types.NUMERIC	Se almacena el **ordinal** del enumerado
5	Types.SMALLINT	Se almacena el **ordinal** del enumerado
-6	Types.TINYINT	Se almacena el **ordinal** del enumerado
-5	Types.BIGINT	Se almacena el **ordinal** del enumerado
3	Types.DECIMAL	Se almacena el **ordinal** del enumerado
8	Types.DOUBLE	Se almacena el **ordinal** del enumerado
6	Types.FLOAT	Se almacena el **ordinal** del enumerado
1	Types.CHAR	Se almacena el **nombre** del enumerado
-16	Types.LONGVARCHAR	Se almacena el **nombre** del enumerado
12	Types.VARCHAR	Se almacena el **nombre** del enumerado

```
<param name="type">4</param>
```

Valor del Enumerado	Valor guardado en la base de datos
Carrera	0
Practicas	1
Interino	2

```
<param name="type">12</param>
```

Valor del Enumerado	Valor guardado en la base de datos
Carrera	Carrera
Practicas	Practicas
Interino	Interino

121

Especiales

- Especificar que se debe recuperar perezosamente esta propiedad cuando se acceda por primera vez la variable de instancia.

```
@Basic(fetch = FetchType.LAZY)
@Lob
 private Blob image;
```

- Se puede especificar una expresión SQL que define el valor para una propiedad calculada, no tienen una columna mapeada propia y se recupera ya calculada.

```
@Formula(value = "credit * rate")
private Double interest;
```

- Es necesario especificar las columnas calculadas, solo lectura, cuyas propiedades debe ignorar la persistir:
  - INSERT: el valor de propiedad se genera en la inserción pero no se regenera en actualizaciones posteriores.
  - ALWAYS: el valor de la propiedad se genera tanto en la inserción como en la actualización.

```
@Generated( value = GenerationTime.ALWAYS )
private Date timestamp;
```


122

Proxy

- El patrón Proxy es un patrón estructural que tiene como propósito proporcionar un subrogado o intermediario de un objeto para controlar su acceso.
- Un proxy es una clase que envuelve (hereda) a otra para poder retrasar el coste de crear e inicializar un objeto hasta que es realmente necesario.
- Hibernate utiliza proxys para implementar las cargas perezosas de forma transparente.
- Hibernate usa bibliotecas de manipulación de Bytecode como Javassist o Byte Buddy, que hacen uso de la reflexión para modificar la implementación de una clase en tiempo de ejecución .
- La colección org.hibernate.collection.internal.PersistentBag envuelve las colecciones con carga perezosa.
- Para evitar problemas (y seguir los buenos principios de diseño) es importante que las colecciones se definan utilizando la interfaz adecuada de Java Collections Framework en lugar de una implementación específica, Hibernate (como otros proveedores de persistencia) utilizará sus propias implementaciones de colección que se ajustan a las interfaces de Java Collections Framework.
- Las colecciones persistentes inyectadas por Hibernate se comportan como ArrayList, HashSet, TreeSet, HashMapo TreeMap, en función del tipo de interfaz.

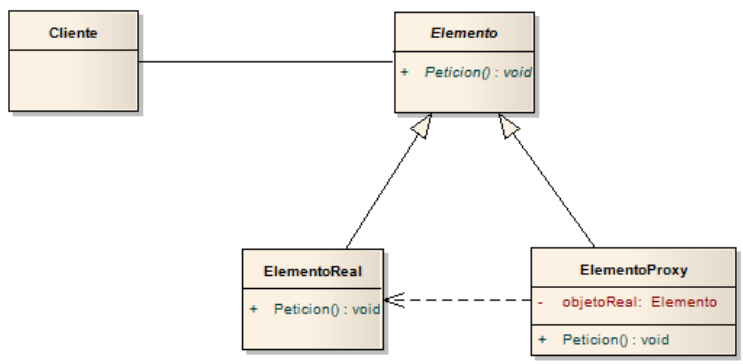
Tipos de Proxy

- Dependiendo de las responsabilidades y del comportamiento del proxy, tendremos varios tipos que realizarán unos tipos de tarea u otras. Los proxies más comunes son los siguientes:
  - Proxy remoto: un proxy remoto se comporta como un representante local de un objeto remoto. Se encarga principalmente de abstraer la comunicación entre nuestro cliente y el objeto remoto. Es el embajador de los proxies.
  - Proxy virtual: se encarga de instanciar objetos cuyo coste computacional es elevado. Es capaz de sustituir al objeto real durante el tiempo que el verdadero objeto está siendo construido y proporcionar funcionalidades como el lazy loading (realizar operaciones computacionalmente costosas únicamente cuando el acceso a el elemento es requerido).
  - Proxy de protección: establece controles de acceso a un objeto dependiendo de permisos o reglas de autorización.
- El patrón proxy modifica el comportamiento pero no amplia la funcionalidad, para ello se utilizaría patrones como Adapter o Decorator.

Estructura del patrón Proxy

- El Elemento es una clase abstracta (o interfaz) que define las operaciones que deberá cumplimentar tanto el objeto real (ElementoReal) como el proxy que actuará de intermediario (ElementoProxy). Ambos elementos, al heredar de Elemento, deberán ser, por tanto, intercambiables. De este modo, sustituir un objeto de la clase ElementoReal por un ElementoProxy debería de ser -idealmente- transparente.
- La clase ElementoReal es aquella que contiene la verdadera funcionalidad, es decir, la clase original que se quiere "proteger" a través del proxy.
- La clase ElementoProxy también hereda de Elemento, y como tal, posee todos sus métodos. La diferencia fundamental es que incorpora una referencia a otro ElementoReal y sus métodos actúan como pasarela a los métodos del ElementoReal.

Estructura del patrón Proxy



Implementación del patrón Proxy

```
public abstract class Elemento {
	public abstract void metodo();
}
public class ElementoReal extends Elemento {
	@Override
	public void metodo() { }
}
public class ElementoProxy extends Elemento {
	private ElementoReal objeto;
	private ElementoReal getObjeto() {
		if(objeto == null) objeto = new ElementoReal();
		return objeto;
	}
	@Override
	public void metodo() {
		getObjeto().metodo();
	}
}
```


127

Problemas con los Proxy Hibernate

- Cuando se difiere la carga, Hibernate sustituye la instancia real por un proxy heredero de la real:

```
Entidad e = session.getReference(Entidad.class, 1001L); // Tipo: Entidad$HibernateProxy$3UWSrq7i
```

- Lo cual presenta problemas con las clases entidad selladas, en cuyo caso no genera un proxy, materializa la consulta.
- Para evitar la materialización es necesario generar, implementar y asociar manualmente el interfaz que actuará de proxy:

```
public interface EntidadProxy {
    long getId();
    void setId(long id);
}

@Entity
@Proxy(proxyClass = EntidadProxy.class)
public final class Entidad implements Serializable, EntidadProxy { … }

Entidad e = session.getReference(Entidad.class, 1001L); // ERROR
EntidadProxy e = session.getReference(Entidad.class, 1001L); // OK
```


128

Proxys LOBs

- Los localizadores JDBC LOB existen para permitir un acceso eficiente a los datos LOB (Blob, Clob, NClod). Permiten que el controlador JDBC transmita partes de los datos LOB según sea necesario, lo que potencialmente libera espacio en la memoria. Sin embargo, pueden ser poco naturales de tratar y tienen ciertas limitaciones, para facilitar su manejo Hibernate suministra el correspondiente juego de proxys:

```
@Entity(name = "Product")
public static class Product {
      @Lob
      private Blob image;
}

byte[] image = new byte[] {1, 2, 3};

final Product product = new Product();
product.setImage( BlobProxy.generateProxy( image ) );
// …
try (InputStream inputStream = product.getImage().getBinaryStream()) {
   // …
}
```


Conversores

```
@Converter
public class PeriodStringConverter  implements AttributeConverter<Period, String> {

   @Override
   public String convertToDatabaseColumn(Period attribute) {
      return attribute.toString();
   }
   @Override
   public Period convertToEntityAttribute(String dbData) {
      return Period.parse( dbData );
   }
}

@Convert(converter = PeriodStringConverter.class)
@Column(columnDefinition = "")
private Period span;
```

Filtro de entidades

- A veces, se desea filtrar entidades o colecciones utilizando criterios SQL personalizados:

```
@Entity(name = "Account")
@Where( clause = "deleted = false" )
public static class Account {
```

- La anotación @Filter es otra forma de filtrar entidades o colecciones utilizando criterios SQL personalizados pero permite parametrizar la cláusula de filtro en tiempo de ejecución.

```
@Entity(name = "Account")
@FilterDef(name="activeAccount", parameters = @ParamDef(
      name="active", type="boolean"))
@Filter(name="activeAccount", condition="active_status = :active")
public static class Account {
```

- Los @Filter se activan al realizar las consultas.

Inmutabilidad

- Si una entidad específica es inmutable, no van a cambiar sus datos, es una buena práctica marcarla con la anotación @Immutable.
- Cuando una entidad es de solo lectura:
  - Hibernate no comprueba las propiedades simples de la entidad o las asociaciones de un solo extremo;
  - Hibernate no actualizará propiedades simples o asociaciones actualizables de un solo extremo;
  - Hibernate no actualizará la versión de la entidad de solo lectura si solo se modifican propiedades simples o asociaciones actualizables de un solo extremo;
  - Hibernate reducirá el uso de memoria, ya que no es necesario retener el estado para el mecanismo de gestión de cambios.
- También las colecciones pueden marcarse como @Immutable, el efecto será local a la colección.
- Las propiedades simples no se pueden marcar como @Immutable pero se pueden definir como:

```
@Column(insertable=false, updatable=false)
```

Método Equals

- Cuando en Java queremos comprobar si 2 objetos son idénticos, utilizamos el método *equals()*.
- En Hibernate esto difiere.
  - 2 objetos son iguales si hacen referencia a la misma fila de la base de datos aunque los objetos sean distintos.
- Este carácter especial hace que existan algunos "inconvenientes" a la hora de implementar esta comprobación en Hibernate.
- La igualdad de clave de principal significa que el método equals() solamente debe comparar las propiedades que forman la clave de principal.

Método Equals

```
public class Entidad {
   ...
   public boolean equals(Object other) {
      if (this == other) return true;
      if ( !(other instanceof Entidad) ) return false;

      return this.getId().Equals(((Entidad)other).getId());
   }
   public int hashCode() {
      return this.getId();
   }
}
```

ASOCIACIONES CON HIBERNATE

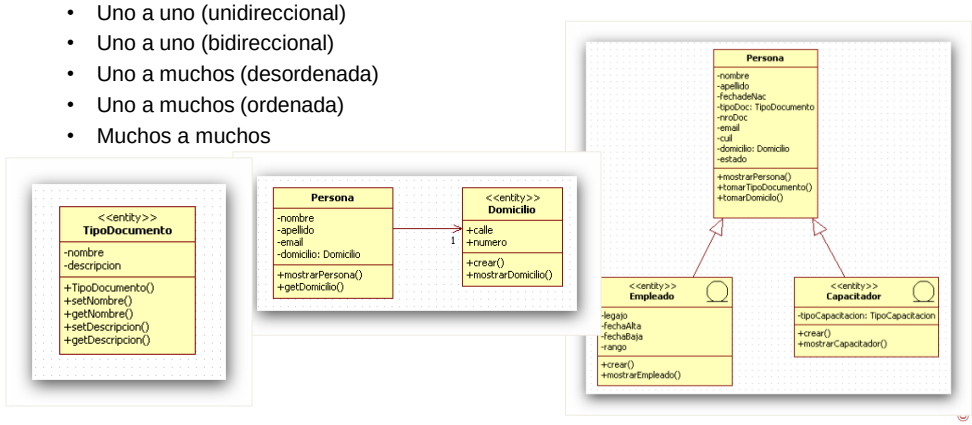
Asociaciones en Hibernate

- Existen 2 temas importantes cuando empezamos a desarrollar en Hibernate:
  - Mapeo de asociaciones entre las clases de entidad.
  - Mapeo de colecciones
- Hasta ahora hemos visto el mapeo simple entre una clase y una entidad, pero la base de datos no contienen solo tablas aisladas, la tablas tienen relaciones dentro de las BBDD.
- Todos los mapeos que vamos a realizar pueden hacer con sus correspondiente fichero *.hbm.xml* o con anotaciones JPA.

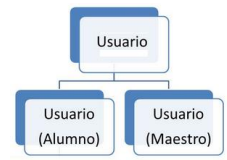
Asociaciones en Hibernate

- Cuando utilizamos la palabra asociación, nos referimos a la relación existente entre tablas de la Base de Datos (entidades).

- Básicamente se podrían definir las siguientes asociaciones:
  - Uno a uno (unidireccional)
  - Uno a uno (bidireccional)
  - Uno a muchos (desordenada)
  - Uno a muchos (ordenada)
  - Muchos a muchos



Asociaciones en Hibernate



Uno a uno

- La relación uno a uno en Hibernate consiste en que un objeto tenga una referencia a otro objeto de forma que al persistirse el primer objeto también se persista el segundo.

- Si en el Modelo Relacional encontramos PK-FK entre entidades, entonces deberemos crear una Asociación de ***1 a muchos***

- Este tipo de relación se puede crear cuando "necesitemos" establecer una relación entre entidades que no tengan columnas comunes ni relacionadas.

Asociaciones en Hibernate

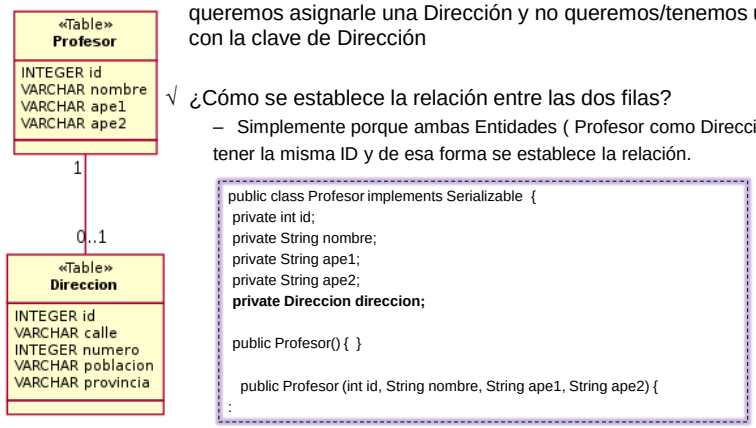
Uno a uno

- En Hibernate se puede diferenciar 2 tipos de relaciones uno-a-uno:
- Unidireccional
  - √ Cuando se hace la persistencia del primer Objeto, también se hace del objeto referenciado, pero no a la inversa.
- Bidireccional
  - √ Cuando se hace la persistencia del primer Objeto, también se hace del objeto referenciado, y a la inversa.
  - √ Cuando se hace la persistencia del objeto referenciado se puede hacer o no la del primer Objeto.

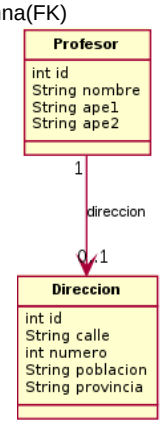
Asociaciones en Hibernate

Uno a uno (Unidireccional)

- √ Cuando 1 objeto tiene una referencia a otro objeto, cuando hacemos persistir el primero, también se hace la persistencia del segundo.
- √ Este tipo de relaciones pueden darse cuando en una entidad (Profesor) queremos asignarle una Dirección y no queremos/tenemos una columna(FK) con la clave de Dirección
- √ ¿Cómo se establece la relación entre las dos filas?
  - – Simplemente porque ambas Entidades (Profesor como Dirección) deben tener la misma ID y de esa forma se establece la relación.




```
public class Profesor implements Serializable  {
 private int id;
 private String nombre;
 private String ape1;
 private String ape2;
 private Direccion direccion;

 public Profesor() { }

   public Profesor (int id, String nombre, String ape1, String ape2) {
:
```



Asociaciones en Hibernate

Uno a uno (Unidireccional)

- √ Al estar utilizando 2 entidades necesitamos 2 archivos de persistencia .hbm.xml
  - – Profesor.hbm.xml
  - – Direccion.hbm.xml

- √ Similares a este: (profesor.hbm.xml)

Profesor.hbm.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD
3.0//EN" "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name="ejemplo01.Profesor" >
    <id column="Id" name="id" type="integer"/>
    <property name="nombre" />
    <property name="ape1" />
    <property name="ape2" />

    <one-to-one name="direccion" cascade="all" />

  </class>
</hibernate-mapping>
```

TAG <one-to-one>

Se utiliza para definir una relación uno a uno entre las dos clases Java.

name:

Nombre de la propiedad Java con la referencia al otro objeto con el que forma la relación uno a uno.

cascade:

Indica a Hibernate cómo debe actuar cuando realicemos las operaciones de persistencia CRUD.

ALL hacer lo mismo en el objeto referenciado

Asociaciones en Hibernate

Uno a uno (Unidireccional)

- √ Sin embargo en el fichero de persistencia de Direccion.hbm.xml, no se hará ninguna referencia al **tag <one-to-one>**

Direccion.hbm.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name="ejemplo01.Direccion" >
    <id column="Id" name="id" type="integer"/>
    <property name="calle"/>
    <property name="numero"/>
    <property name="poblacion"/>
    <property name="provincia"/>

  </class>
</hibernate-mapping>
```

- √ La relación uno a uno tiene una direccionalidad desde Profesor → Dirección por lo tanto Dirección no sabe nada sobre Profesor y por ello en su fichero de persistencia no hay nada relativo a dicha relación.

Asociaciones en Hibernate

Uno a uno (Unidireccional) con JPA

- √ Al igual que en los casos anteriores, cuando utilizamos anotaciones JPA deberemos modificar el código fuente de las clases Java **y no usar los ficheros .hbm.xml.**

- √ Sólo deberemos introducir las anotaciones relacionadas con *one-to-one* en la clase Profesor y en el elemento de unión de ambas.

@OneToOne(cascade=CascadeType.ALL):

Esta anotación indica la relación uno a uno de las 2 tablas.
También indicamos el valor de *cascade* al igual que en el fichero de Hibernate.

@PrimaryKeyJoinColumn:

Indicamos que la relación entre las dos tablas se realiza mediante la clave primaria.

Hibernate entiende por Clave primaria, el identificador ID de cada una de las tablas, no una PK de SGBDR

```
:
    @Column(name="ape2")
    private String ape2;

    @OneToOne(cascade=CascadeType.ALL)
    @PrimaryKeyJoinColumn
    private Direccion direccion;
}
:
```

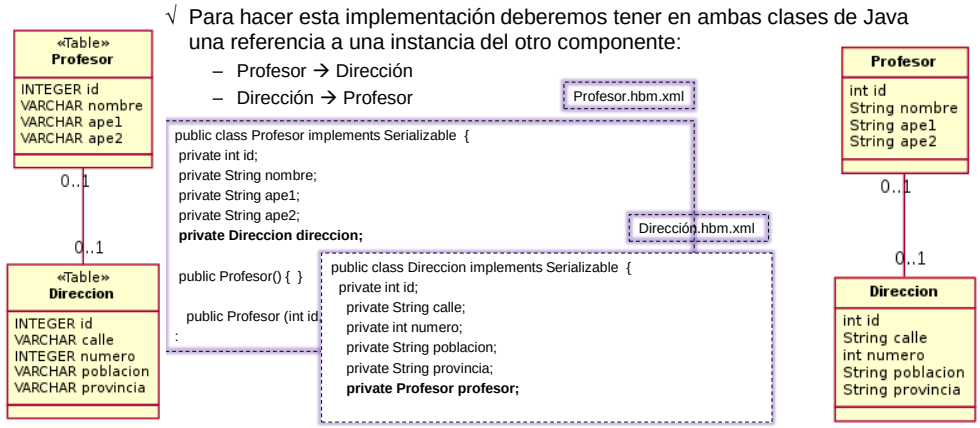

146

Asociaciones en Hibernate

Uno a uno (Bidireccional)

- √ Este tipo de relaciones aparecen cuando 1 objeto tiene una referencia a otro objeto, cuando hacemos persistir el primero, también se hace la persistencia del segundo y viceversa.

- √ Para hacer esta implementación deberemos tener en ambas clases de Java una referencia a una instancia del otro componente:
  - Profesor → Dirección
  - Dirección → Profesor



147

Asociaciones en Hibernate

Uno a uno (Bidireccional)

- √ Al estar utilizando 2 entidades necesitamos 2 archivos de persistencia .hbm.xml
  - – Profesor.hbm.xml
  - – Direccion.hbm.xml

- √ Deberemos poner el TAG de <one-to-one> en ambos ficheros .hbm.xml

Profesor.hbm.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD
3.0//EN" "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name="ejemplo01.Profesor" >
    <id column="Id" name="id" type="integer"/>
    <property name="nombre" />
    <property name="ape1" />
    <property name="ape2" />

    <one-to-one name="direccion" cascade="all" />

  </class>
</hibernate-mapping>
```

TAG <one-to-one>

Se utiliza para definir una relación uno a uno entre las dos clases Java.

name:
Nombre de la propiedad Java con la referencia al otro objeto con el que forma la relación uno a uno.

cascade:
Indica a Hibernate cómo debe actuar cuando realicemos las operaciones de persistencia CRUD.
ALL hacer lo mismo en el objeto referenciado

```
:
  <property name="ape2" />
  <one-to-one name="profesor" cascade="all" />
 </class>
</hibernate-mapping>
```

Dirección.hbm.xml

Asociaciones en Hibernate

Uno a uno (bidireccional) con JPA

- √ Al igual que en los casos anteriores, cuando utilizamos anotaciones JPA deberemos modificar el código fuente de las clases Java **y no usar los ficheros .hbm.xml.**

- √ Sólo deberemos introducir las anotaciones relacionadas con *one-to-one* en las clase Profesor y Direccion en el elemento de unión de ambas.

@OneToOne(cascade=CascadeType.ALL):

Esta anotación indica la relación uno a uno de las 2 tablas.
También indicamos el valor de *cascade* al igual que en el fichero de Hibernate.

@PrimaryKeyJoinColumn:

Indicamos que la relación entre las dos tablas se realiza mediante la clave primaria.

Hibernate entiende por Clave primaria, el identificador ID de cada una de las tablas, no una PK de SGBDR

```
:
    @OneToOne(cascade=CascadeType.ALL)
    @PrimaryKeyJoinColumn
    private Direccion direccion;
}
:
```

```
:
    @OneToOne(cascade=CascadeType.ALL)
    @PrimaryKeyJoinColumn
    private Profesor profesor;
}
:
```

Asociaciones en Hibernate

Uno a Muchos

- La relación uno a muchos consiste en que un objeto (padre) tenga una lista sin ordenar de otros objetos (hijo) de forma que al persistirse el objeto principal también se persista la lista de objetos hijo.

- Esta relación también suele llamarse maestro-detalle o padre-hijo.
  - Clientes → Facturas
  - Facturas → Líneas de Detalle
  - Provincias → Localidades
  - Departamentos → Empleados

- Podemos distinguir 2 tipos de Asociaciones:
  - √ Uno a Muchos Desordenados
  - √ Uno a Muchos Ordenados

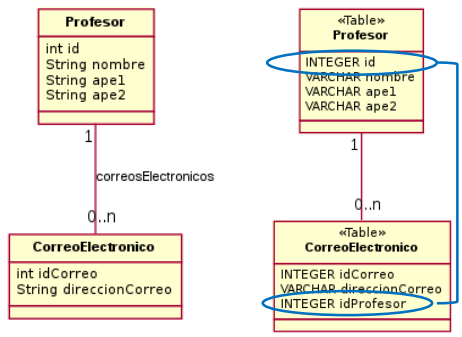
- Habitualmente la lista de objetos usadas son colecciones *(Set, List, etc)* y dependiendo de la colección utilizada se designan Ordenados o Desordenados.

Asociaciones en Hibernate

Uno a Muchos Desordenados

- Para poder realizar este tipo de asociaciones necesitamos una Bidireccionalidad entre ambas (padre-hijo).

- En Java almacenamos la serie de objetos hijos mediante el interface SET o cualquier otra colección que no implique orden de los objetos *(List , Array, Tree no)*.



- En este tipo de relaciones existe una PK y una FK en las tablas relacionales

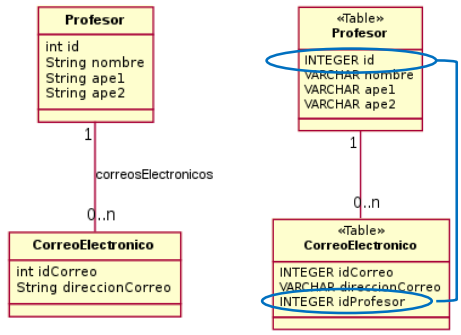
- Id(profesor) →PK
- idProfesor(correo) → FK

Asociaciones en Hibernate

Uno a Muchos Desordenados

- Para hacer esta implementación deberemos tener en ambas clases de Java una referencia a una instancia del otro componente:

- Profesor → **set <CorreoElectronico> correosElectronicos**
- CorreoElectronico → Profesor




```
public class Profesor implements Serializable  {
   private int id;
   private String nombre;
   private String ape1;
   private String ape2;
   private Set<CorreoElectronico> correosElectronicos;
:
```

```
public class CorreoElectronico implements Serializable {
  private int idCorreo;
  private String direccionCorreo;
  private Profesor profesor;
:
```


Asociaciones en Hibernate

Uno a Muchos Desordenados

√ Al estar utilizando 2 entidades necesitamos 2 archivos de persistencia .hbm.xml
  - **Profesor.hbm.xml**
  - CorreoElectronico.hbm.xml

√ Deberemos poner el TAG de **<set>** en el fichero de la clase PADRE

Profesor.hbm.xml

```
:
<hibernate-mapping>
  <class name="ejemplo01.Profesor" >
    <id column="Id" name="id" type="integer"/>
    <property name="nombre" />
    <property name="ape1" />
    <property name="ape2" />

   <set name="correosElectronicos"  cascade="all" inverse="true"  >
      <key>
         <column name="idProfesor"  />
      </key>
     <one-to-many class="ejemplo05.CorreoElectronico" />
   </set>
  </class>
</hibernate-mapping>
```

TAG <set>

Se utiliza para definir una relación uno a muchos desordenada entre 2 clases

name:
Nombre de la propiedad SET en la que se almacenan los objetos.

cascade:
Indica a Hibernate cómo debe actuar cuando realicemos las operaciones de persistencia CRUD.
ALL hacer lo mismo en el objeto referenciado

inverse:
Usado para minimizar las operaciones SQL que hace Hibernate contra la BBDD

Asociaciones en Hibernate

Uno a Muchos Desordenados

- √ Al estar utilizando 2 entidades necesitamos 2 archivos de persistencia .hbm.xml
  - – **Profesor.hbm.xml**
  - – CorreoElectronico.hbm.xml

- √ Deberemos poner el TAG de **<set>** en el fichero de la clase PADRE

Profesor.hbm.xml

```
:
<hibernate-mapping>
  <class name="ejemplo01.Profesor" >
    <id column="Id" name="id" type="integer"/>
    <property name="nombre" />
    <property name="ape1" />
    <property name="ape2" />

   <set name="correosElectronicos"  cascade="all" inverse="true"  >
      <key>
         <column name="idProfesor"  />
      </key>
      <one-to-many class="ejemplo05.CorreoElectronico" />
   </set>
  </class>
</hibernate-mapping>
```

TAG <set>

Se utiliza para definir una relación uno a muchos desordenada entre 2 clases

key:
Nombre de la columna de la BBDD que actúa como **clave Foranea FK** en la relación

<one-to-many>:
Contiene la clase de Java que actúa como HIJA en la relación.

155

Asociaciones en Hibernate

Uno a Muchos Desordenados

- √ Al estar utilizando 2 entidades necesitamos 2 archivos de persistencia .hbm.xml
  - – Profesor.hbm.xml
  - – **CorreoElectronico.hbm.xml**

- √ Deberemos poner el TAG de **<many-to-one>** en el fichero de la clase HIJA

CorreoElectronico.hbm.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD
3.0//EN" "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name="ejemplo05.CorreoElectronico" >
    <id column="IdCorreo" name="idCorreo" type="integer"/>
    <property name="direccionCorreo" />

   <many-to-one name="profesor">
        <column name="idProfesor"  />
   </many-to-one>

  </class>
</hibernate-mapping>
```

TAG <many-to-one>

Se utiliza para definir una relación de mucho a uno entre 2 clases

name:
Nombre de la PROPIEDAD JAVA que enlaza con el objeto PADRE, es decir, la variable declarada para referenciar al padre.

Column name:
Indica la columna de la Tabla Hija que actúa como FK

156

Asociaciones en Hibernate

Uno a Muchos Desordenados con JPA

√ Al igual que en los casos anteriores, cuando utilizamos anotaciones JPA deberemos modificar el código fuente de las clases Java **y no usar los ficheros .hbm.xml.**

√ Sólo deberemos introducir las anotaciones:

PADRE	(Profesor)	*@oneToMany*
HIJO	(correoElectronico)	*@ ManyToOne @JoinColumn*

Profesor.java

```
:
@OneToMany(mappedBy="profesor",cascade= CascadeType.ALL)
private Set<CorreoElectronico> correosElectronicos; }
:
```

@OneToMany**(mappedBy**="profesor",**cascade**= CascadeType.ALL)

Esta anotación indica la relación uno a Muchos

mappedBy:

√ Este atributo contendrá el Nombre de la **PROPIEDAD JAVA de la clase HIJA** que enlaza con el objeto PADRE, es decir, la variable declarada para referenciar al padre.

cascade

√ Este atributo tiene el mismo significado que el del fichero de mapeo de Hibernate.

Asociaciones en Hibernate

Uno a Muchos Desordenados con JPA

√ Al igual que en los casos anteriores, cuando utilizamos anotaciones JPA deberemos modificar el código fuente de las clases Java **y no usar los ficheros .hbm.xml.**

√ Sólo deberemos introducir las anotaciones:

PADRE	(Profesor)	*@oneToMany*
HIJO	(correoElectronico)	*@ManyToOne @JoinColumn*

correoElectronico.java

```
:
@ManyToOne
@JoinColumn(name="IdProfesor")
 private Profesor profesor;
:
```

@ManyToOne

Esta anotación indica la relación de Muchos a uno

@JoinColumn (name="idProfesor")

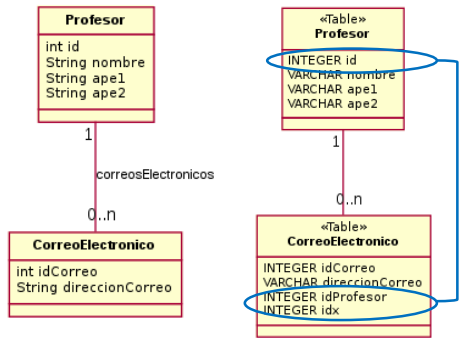
Indicaremos el nombre de la columna que en la tabla hija contiene la clave ajena a la tabla padre.

Asociaciones en Hibernate

Uno a Muchos Ordenados

- Para poder realizar este tipo de asociaciones necesitamos una Bidireccionalidad entre ambas (padre-hijo).

- En Java almacenamos la serie de objetos hijos mediante el interface LIST o cualquier otra colección que **implique** orden de los objetos *(List , Array, Tree)*.



- Id(profesor) →PK
- idProfesor(correo) → FK

- Idx → Columna adicional para indicar orden de los hijos

- Las clases ejemplo utilizadas serán idénticas a las anteriores pero utilizando una colección ordenada

Asociaciones en Hibernate

Uno a Muchos Ordenados

√ Al estar utilizando 2 entidades necesitamos 2 archivos de persistencia .hbm.xml
- **Profesor.hbm.xml**
- CorreoElectronico.hbm.xml

√ Deberemos poner el TAG de **<list>** en el fichero de la clase PADRE

Profesor.hbm.xml

```
<hibernate-mapping>
  <class name="ejemplo01.Profesor" >
    <id column="Id" name="id" type="integer"/>
    <property name="nombre" /> <property name="ape1" />   <property name="ape2" />

  <list name="correosElectronicos"  cascade="all" inverse="true"  >
      <key>
        <column name="idProfesor"  />
      </key>
      <list-index>
          <column name="Idx"  />
      </list-index>
      <one-to-many class="ejemplo05.CorreoElectronico" />
  </list>
  </class>
</hibernate-mapping>
```

TAG <list>

Se utiliza para definir una relación uno a muchos entre 2 clases en las cuales hay un orden

name:
Nombre de la propiedad SET en la que se almacenan los objetos.

cascade:
Indica a Hibernate cómo debe actuar cuando realicemos las operaciones de persistencia CRUD.
ALL hacer lo mismo en el objeto referenciado

inverse:
Usado para minimizar las operaciones SQL que hace Hibernate contra la BBDD

Asociaciones en Hibernate

Uno a Muchos Ordenados

- √ Al estar utilizando 2 entidades necesitamos 2 archivos de persistencia .hbm.xml
  - – **Profesor.hbm.xml**
  - – CorreoElectronico.hbm.xml

- √ Deberemos poner el TAG de **<list>** en el fichero de la clase PADRE

Profesor.hbm.xml

```
<hibernate-mapping>
  <class name="ejemplo01.Profesor" >
    <id column="Id" name="id" type="integer"/>
    <property name="nombre" /> <property name="ape1" />  <property name="ape2" />

  <list name="correosElectronicos"  cascade="all" inverse="true"  >
      <key>
         <column name="idProfesor"  />
      </key>
      <list-index>
          <column name="Idx"  />
      </list-index>
      <one-to-many class="ejemplo05.CorreoElectronico" />
    </list>
  </class>
</hibernate-mapping>
```

TAG <list>

Se utiliza para definir una relación uno a muchos desordenada entre 2 clases

key:
Nombre de la columna de la BBDD que actúa como **clave Foranea FK** en la relación

List-index: Column name
Nombre de la columna de la tabla HIJA donde se guarda el orden que ocupan dentro de la Lista.

<one-to-many>:
Contiene la clase de Java que actúa como HIJA en la relación.

Asociaciones en Hibernate

Uno a Muchos Ordenados

- √ Al estar utilizando 2 entidades necesitamos 2 archivos de persistencia .hbm.xml
  - – Profesor.hbm.xml
  - – **CorreoElectronico.hbm.xml**

- √ Deberemos poner el TAG de **<many-to-one>** en el fichero de la clase HIJA

CorreoElectronico.hbm.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD
3.0//EN" "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name="ejemplo05.CorreoElectronico" >
    <id column="IdCorreo" name="idCorreo" type="integer"/>
    <property name="direccionCorreo" />

    <many-to-one name="profesor">
         <column name="idProfesor"  />
    </many-to-one>

  </class>
</hibernate-mapping>
```

TAG <many-to-one>

Se utiliza para definir una relación de mucho a uno entre 2 clases

name:
Nombre de la PROPIEDAD JAVA que enlaza con el objeto PADRE, es decir, la variable declarada para referenciar al padre.

Column name:
Indica la columna de la Tabla Hija que actúa como FK

Asociaciones en Hibernate

Uno a Muchos Ordenados con JPA

√ Al igual que en los casos anteriores, cuando utilizamos anotaciones JPA deberemos modificar el código fuente de las clases Java **y no usar los ficheros .hbm.xml.**

√ Deberemos introducir las siguientes anotaciones:

PADRE	(Profesor)	*@oneToMany @JoinColum @IndexColumn*
HIJO	(correoElectronico)	*@ManyToOne @JoinColum*

Profesor.java

```
@OneToMany(cascade= CascadeType.ALL)
@JoinColumn(name="IdProfesor")
@IndexColumn(name="idx")
private List<CorreoElectronico> correosElectronicos;
```

@OneToMany**(mappedBy**="profesor",**cascade**= CascadeType.ALL)

Esta anotación indica la relación uno a Muchos y es idéntica a la anterior usada

@JoinColumn (name="idProfesor")

Indicaremos el nombre de la columna que en la tabla hija contiene la clave ajena a la tabla padre.

@IndexColumn (name="idx")

Indicaremos el nombre de la columna que en la tabla hija contiene el orden dentro de la Lista

Asociaciones en Hibernate

Uno a Muchos Ordenados con JPA

√ Al igual que en los casos anteriores, cuando utilizamos anotaciones JPA deberemos modificar el código fuente de las clases Java **y no usar los ficheros .hbm.xml.**

√ Deberemos introducir las siguientes anotaciones:

PADRE	(Profesor)	*@oneToMany @JoinColum @IndexColumn*
HIJO	(correoElectronico)	*@ManyToOne @JoinColum*

correoElectronico.java

```
:
@ManyToOne
@JoinColumn(name="IdProfesor")
 private Profesor profesor;:
```

@ManyToOne

Esta anotación indica la relación de Muchos a uno

@JoinColumn (name="idProfesor")

Indicaremos el nombre de la columna que en la tabla hija contiene la clave ajena a la tabla padre.

Asociaciones en Hibernate

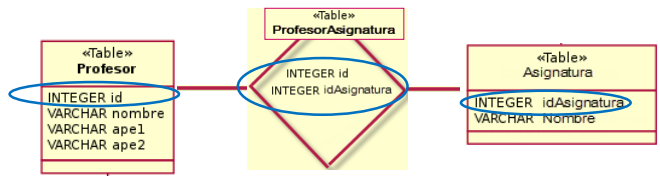
Muchos a Muchos

- La relación muchos a muchos consiste en que un objeto A tenga una lista de otros objetos B y también que el objeto B a su vez tenga la lista de objetos A.
- De forma que al persistirse cualquier objeto también se persista la lista de objetos que posee.
- En este tipo de relaciones, aparece una nueva Entidad (*Renacida*) que sirve de conexión entre ambas dos.
- Habitualmente las listas de objetos utilizadas son colecciones *(Set, HashSet, TreeSet, etc)* y dependiendo de la colección utilizada se designan Ordenados o Desordenados.
- Ejemplo:
  - √ Profesores → Asignaturas
  - √ Asignaturas → Profesores

Asociaciones en Hibernate

Muchos a Muchos

- En Relacional, en toda asociación muchos a muchos se descompone en una Entidad Renacida (ProfesorAsignatura) que contiene las PK de cada una de las Entidades.




```
Create Table ProfesorAsignatura(
id int(11) references Profesor(id),
IdAsignatura int(11) references Asignatura(idAsignatura));
```

```
Create Table Asignatura(
idAsignatura int(11),
Nombre varchar(255) )
```

Asociaciones en Hibernate

Muchos a Muchos

- Dentro de cada una de las clases, deberemos tener una referencia a un conjunto de objetos de la otra clase.

- Esta referencia e implementa mediante colecciones (Set)


```
public class Profesor implements Serializable  {
    private int id;
    private String nombre;
    private String ape1;      private String ape2;
    private Set<Asignatura> asignatura=new HashSet();

   public Profesor(){ }

  public Profesor(int id, String nombre, String ape1, String ape2)
        {   this.id = id;
            this.nombre = nombre;
            this.ape1 = ape1;
           this.ape2 = ape2;
   }
 }
```

```
public class Asignatura implements Serializable {
    private int idAsignatura;
    private String nombre;
    private Set<Profesor> profesores=new HashSet();

    public Asignatura() {  }

    public Asignatura (int id Asignatura, String nombre) {
       this.id Asignatura = idAsignatura;
       this.nombre = nombre;
    }
}
```


Asociaciones en Hibernate

Muchos a Muchos

√ Al estar utilizando 2 entidades necesitamos 2 archivos de persistencia .hbm.xml
- **Profesor.hbm.xml**
- Asignatura.hbm.xml

√ Deberemos poner el TAG de **<set>** en el fichero de la clase PADRE

Profesor.hbm.xml

```
<hibernate-mapping>
  <class name="ejemplo01.Profesor" >
    <id column="Id" name="id" type="integer"/>
    <property name="nombre" /> <property name="ape1" />   <property name="ape2" />

   <set name="asignatura"  table="ProfesorAsignatura" cascade="all"
inverse="true"  >
      <key>
         <column name="idProfesor"  />
      </key>
     <many-to-many column=idAsignatura class="ejemplo05.Asignatura" />
   </set>
  </class>
</hibernate-mapping>
```

TAG <set>

Se utiliza para definir una relación muchos a muchos entre 2 clases en las cuales hay un orden

name:
Nombre de la propiedad SET en la que se almacenan los objetos.

table:
Nombre de la tabla RENACIDA.

cascade:
Indica a Hibernate cómo debe actuar cuando realicemos las operaciones de persistencia CRUD.
ALL hacer lo mismo en el objeto referenciado

inverse:
Usado para minimizar las operaciones SQL que hace Hibernate contra la BBDD

Asociaciones en Hibernate

Muchos a Muchos Ordenados

- √ Al estar utilizando 2 entidades necesitamos 2 archivos de persistencia .hbm.xml
  - – **Profesor.hbm.xml**
  - – Asignatura.hbm.xml
- √ Deberemos poner el TAG de **<set>** en el fichero de la clase PADRE

Profesor.hbm.xml

```
<hibernate-mapping>
 <class name="ejemplo01.Profesor" >
  <id column="Id" name="id" type="integer"/>
  <property name="nombre" /> <property name="ape1" />  <property name="ape2" />

 <set name="asignatura"  table="ProfesorAsignatura" cascade="all"
inverse="true"  >
    <key>
      <column name="idProfesor"  />
    </key>
    <many-to-many column=idAsignatura class="ejemplo05.Asignatura" />
  </set>
 </class>
</hibernate-mapping>
```

TAG <set>

Se utiliza para definir una relación uno a muchos desordenada entre 2 clases

key:
Nombre de la columna de la RENACIDA que actúa como **clave Foranea FK** en la relación con esta tabla.

<many-to-many>:
Contiene la clase de Java que actúa como HIJA en la relación.

Asociaciones en Hibernate

Muchos a Muchos Ordenados

- √ Al estar utilizando 2 entidades necesitamos 2 archivos de persistencia .hbm.xml
  - – Profesor.hbm.xml
  - – **Asignatura.hbm.xml**
- √ Deberemos poner el TAG de **<set>** en el fichero de la clase PADRE

Asignatura.hbm.xml

```
<hibernate-mapping>
 <class name="ejemplo01.Asignatura" >
  <id column="IdAsignatura" name="idAsignatura" type="integer"/>
  <property name="nombre" />

 <set name="profesor"  table="ProfesorAsignatura" cascade="all"
inverse="true"  >
    <key>
      <column name="idAsignatura"  />
    </key>
    <many-to-many column=idProfesor. class="ejemplo05.Profesor" />
  </set>
 </class>
</hibernate-mapping>
```

TAG <set>

Se utiliza para definir una relación uno a muchos desordenada entre 2 clases

key:
Nombre de la columna de la RENACIDA que actúa como **clave Foranea FK** en la relación con esta tabla.

<many-to-many>:
Contiene la clase de Java que actúa como HIJA en la relación.

© JAA JMA 2015

Atributo: Cascade

- Como hemos visto anteriormente, el atributo *cascade* se utiliza en los mapeos de las Asociaciones..
- Este atributo indica qué debe hacer Hibernate con las clases relacionadas cuando realizamos alguna acción con la clase principal.
  - √ Si borramos en la Principal → ¿Tenemos que borrar en la clase Hija?
  - √ Si insertamos en la Principal → …….
- Este atributo tiene 11 valores dependiendo de nuestros modelos de clases.
- Es útil cuando queremos que las asociaciones se comporten de forma especial.

Atributo: Cascade

Valor	Descripción
none	- No se hace nada en la clase Hija
Save-update	- Si se inserta o actualiza el objeto principal también se realizará la inserción o actualización en los objetos relacionados.
delete	- Si se borra el objeto principal también se realizará el borrado en los objetos relacionados.
evict **lock** **merge** **refresh** **replicate**	- Si se llama al método Session.XXXXXXXX(Object objeto) para separar, bloquear, mezclar, refrescar o replicar el objeto principal también se llamará para los objetos relacionados
all	- Si se realiza cualquiera de las **anteriores** acciones sobre el objeto principal también se realizará sobre los objetos relacionados
delete-orphan	- Solo se usa con colecciones - Si en una colección del objeto principal eliminamos un elemento, al persistir el objeto principal deberemos borrar de la base de datos el elemento de la colección que habíamos eliminado.
all-delete-orphan	- Es la unión de los atributos **all** y **delete-orphan**

Cascada con anotaciones

- Parámetro cascade de la anotación:

  @OneToMany(mappedBy="profesor", cascade=CascadeType.ALL)

- Enumeración de tipo CascadeType:
  - ALL = {PERSIST, MERGE, REMOVE, REFRESH, DETACH}
  - DETACH
  - MERGE
  - PERSIST
  - REFRESH
  - REMOVE

- Acepta múltiples valores:

  @OneToMany(mappedBy="profesor", cascade={CascadeType.PERSIST, CascadeType.MERGE})

Lazy Loading (Carga Perezosa)

- Lazy Loading es la *estrategia de recuperación* del Hibernate asociada a Colecciones.

- Cuando intentemos recuperar los objetos de A se hará un consulta a la base de datos para recuperar los datos de la tabla TA, pero no se traerá los datos correspondientes de la tabla TB.

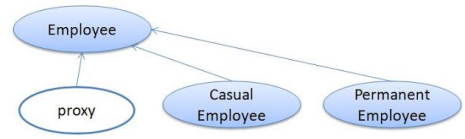
- Los datos de la TB se recuperarán cuando la aplicación invoca una operación sobre esa colección.
  - Si disponemos de un objeto que puede tener varios hijos, no recuperamos esos hijos hasta que los vayamos a utilizar
    - √ Profesores → Direcciones 1 → N
    - √ Profesores → CorreosElectronicos 1 → N

```
@OneToMany(fetch=FetchType.LAZY)
@OneToMany(fetch=FetchType.EAGER)
```

```
<many-to-one name="employee"
  class="com.myapp.Employee"
  lazy="false"
  fetch="select">
```

Lazy Loading (Carga Perezosa)

- En la primera petición, se proporcionará un proxy (gestor) que se encargará de hacer las consultas a la base de datos cuando realmente se consulta la colección de objetos de la clase B
  - PROXY: Es un mecanismo que permite a Hibernate romper la interconexión de los objetos en la base de datos en fragmentos más pequeños, que pueden caber fácilmente en la memoria.



- Visto desde Hibernate:
  - Esta aproximación es adecuada, ya que no se recuperan los datos de TB si no son realmente necesarios.
- Visto desde BBDD
  - Esta aproximación es ineficiente, ya que se hacen demasiadas consultas para recuperar una información que podríamos obtener con una sola consulta

Lazy Loading (Carga Perezosa)

- Configuración:
  - Lazy
    - √ Define el “cuando” se cargan datos.
    - √ Por defecto lazy es igual a true. Esto significa que la colección no se recupera de la base de datos hasta que se hace alguna operación sobre ella.
    - √ Si fijamos lazy a false, cuando se recupere la información de A también se traerá toda la información de los objetos relacionados de B.

```
<set name="asignatura" lazy="false">
@OneToMany(mappedBy="profesor", fetch=FetchType.LAZY) // Por defecto
```

  - Fetch
    - √ Define con que SQLs se van a lanzar para recuperar la información.
    - √ Por defecto fetch es igual a select. Esto implica que para recuperar cada objeto de B se lanzará una nueva consulta a la base de datos.
    - √ Si fijamos el valor de fetch a join, en la misma consulta que se recupera la información de A también se recuperará la información de todos los objetos relacionados de B. Esto se consigue con un left outer join en la sentencia SQL.

```
<set name="asignatura" fetch="join">
@OneToMany(mappedBy="profesor", fetch=FetchType.EAGER)
```

Mapeo de Herencia

- Hibernate soporta las tres estrategias básicas de mapeo de herencia:
  - tabla por jerarquía de clases
  - tabla por subclases
  - tabla por clase concreta
- Se modelizan a través de relaciones OR del modelo relacional.
- Es posible utilizar estrategias de mapeo diferentes para diferentes ramificaciones de la misma jerarquía de herencia.
- Es posible definir los mapeos subclass, union-subclass, y joined-subclass en documentos de mapeo separados, directamente debajo de hibernate-mapping.
- Esto permite extender una jerarquía de clases solamente añadiendo un nuevo archivo de mapeo.

Tabla por jerarquía de clases

- Requiere una sola tabla con:
  - Identificador común
  - Discriminador que establece a que subclase pertenece
  - Propiedades de la clase base, comunes a todas sus subclases
  - Propiedades particulares de cada subclase

- Hay una limitación en esta estrategia de mapeo:
  - La columna discriminador es obligatoria.
  - Las columnas declaradas por las subclases, no pueden tener restricciones NOT NULL.

Tabla por jerarquía de clases

```
<class name="Payment" table="PAYMENT">
   <id name="id" type="long" column="PAYMENT_ID">
      <generator class="native"/>
   </id>
   <discriminator column="PAYMENT_TYPE" type="string"/>
   <property name="amount" column="AMOUNT"/>
   ...
   <subclass name="CreditCardPayment" discriminator-value="CREDIT">
      <property name="creditCardType" column="CCTYPE"/>
      ...
   </subclass>
   <subclass name="CashPayment" discriminator-value="CASH">
      ...
   </subclass>
   <subclass name="ChequePayment" discriminator-value="CHEQUE">
      ...
   </subclass>
</class>
```


Anotaciones para Tabla por jerarquía de clases

```
@Entity(name = "Account")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="PAYMENT_TYPE")
public abstract class Account { … }

@Entity(name = "DebitAccount")
@DiscriminatorValue(value = "Debit")
public class DebitAccount extends Account { … }

@Entity(name = "CreditAccount")
@DiscriminatorValue( "Credit" )
public class CreditAccount extends Account { … }
```

Tabla por subclase

- Requiere una tabla para la superclase con:
  - Identificador común
  - Propiedades de la clase base, comunes a todas sus subclases
- Una tabla por cada subclase con:
  - Identificador común
  - Propiedades particulares de cada subclase
- Las tablas de subclase tienen asociaciones de clave principal a la tabla de la superclase de modo que en el modelo relacional es realmente una asociación uno-a-uno.
- La implementación de Hibernate de *tabla por subclase* no requiere ninguna columna discriminadora.

Tabla por subclase

```
<class name="Payment" table="PAYMENT">
   <id name="id" type="long" column="PAYMENT_ID">
      <generator class="native"/>
   </id>
   <property name="amount" column="AMOUNT"/>
   ...
   <joined-subclass name="CreditCardPayment" table="CREDIT_PAYMENT">
      <key column="PAYMENT_ID"/>
      <property name="creditCardType" column="CCTYPE"/>
      ...
   </joined-subclass>
   <joined-subclass name="CashPayment" table="CASH_PAYMENT">
      <key column="PAYMENT_ID"/>
      ...
   </joined-subclass>
</class>
```

Anotaciones para Tabla por subclases

```
@Entity(name = "Account")
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class Account { … }

@Entity(name = "DebitAccount")
@PrimaryKeyJoinColumn(name = "account_id")
public class DebitAccount extends Account { … }

@Entity(name = "CreditAccount")
@PrimaryKeyJoinColumn(name = "account_id")
public class CreditAccount extends Account { … }
```


Tabla por clase concreta

- Requiere una tabla por cada clase.
- Si la superclase:
  - es abstracta, hay que mapearla con abstract="true".
  - no es abstracta, se necesita una tabla adicional para mantener las instancias de la superclase.
- Cada tabla define columnas para todas las propiedades de la clase, incluyendo las propiedades heredadas.
- La limitación de este enfoque es que si una propiedad se mapea en la superclase, el nombre de la columna debe ser el mismo en todas las tablas de subclase.
- La estrategia del generador de identidad no está permitida en la herencia de unión de subclase.
- La semilla de la clave principal (secuencia) tiene que compartirse a través de todas las subclases unidas de una jerarquía.

Tabla por clase concreta

```
<class name="Payment" abstract="true">
   <id name="id" type="long" column="PAYMENT_ID">
      <generator class="sequence"/>
   </id>
   <property name="amount" column="AMOUNT"/>
   ...
   <union-subclass name="CreditCardPayment" table="CREDIT_PAYMENT">
      <property name="creditCardType" column="CCTYPE"/>
      ...
   </union-subclass>
   <union-subclass name="CashPayment" table="CASH_PAYMENT">
      ...
   </union-subclass>
</class>
```


Anotaciones Tabla por clase concreta

```
@Entity(name = "Account")
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class Account { … }

@Entity(name = "DebitAccount")
public class DebitAccount extends Account { … }

@Entity(name = "CreditAccount")
public class CreditAccount extends Account { … }
```

@Any: Asociación polimórfica

```
public interface Property<T> {
   String getName();
   T getValue();
}

@Entity
@Table(name="integer_property")
public class IntegerProperty implements Property<Integer> { … }

@Entity
@Table(name="string_property")
public class StringProperty implements Property<String> { … }

 @Any(metaDef = "PropertyMetaDef", metaColumn = @Column( name = "property_type" )    )
 @JoinColumn( name = "property_id" )
 private Property property;
```


189

CONSULTAS CON HQL/JPQL

191

HQL (Hibernate Query Lenguage)

- HQL es una de las 3 maneras que existen en Hibernate para realizar consultas.
  - HQL (Hibernate Query Lenguage)

    ```
    session.createQuery("from Category c where c.name like 'Laptop%'");
    entityManager.createQuery("select c from Category c where c.name like 'Laptop%' ");
    ```

  - CRITERIA API

    ```
    session.createCriteria(Category.class).add( Restrictions.like("name", "Laptop%") )
    ```

  - Direct SQL

    ```
    session.createSQLQuery("select {c.*} from CATEGORY {c} where NAME like 'Laptop%' ").addEntity("c",
    Category.class);
    ```


HQL (Hibernate Query Lenguage)

- HQL es un lenguaje de consultas proporcionado por Hibernate "Hibernate Query Language" .
- HQL es muy parecido al SQL estándar, con la diferencia fundamental de estar completamente orientado a objetos
  - Usamos nombres de clases y sus atributos en lugar de nombres de tablas y columnas
- Al ser orientado a Objetos podemos usar cosas como herencia, polimorfismo y asociaciones.
  - Trabajamos realmente sobre las clase Java y es el Hibernate quien redacta las consultas sobre las tablas mapeadas.

HQL (Hibernate Query Lenguage)

- Al trabajar sobre Java, debemos respetar la diferencia sintáctica entre entre mayúsculas y minúsculas.
  - √ Las clausulas como SELECT, FROM, etc, no distingue entre mayúsculas y minúsculas.
  - √ Las Clases, Objetos y miembros si diferencian entre mayúsculas y minúsculas

- HQL está diseñado exclusivamente para realizar la recuperación y modificación de datos (DQL y DML).

- La redacción de las consultas HQL es muy similar a las cláusulas SELECT del lenguaje SQL.

```
SELECT u.username FROM hql.modelo.Usuario u

SELECT u.username FROM Usuario u
```


HQL (Hibernate Query Lenguage)

Características de HQL

- Soporte completo para operaciones relacionales:
  - √ HQL permite representar consultas SQL en forma de objetos.
  - √ HQL usa clases y atributos o propiedades en vez de tablas y columnas.
- Devolución de resultados en forma de objetos:
  - √ Las consultas realizadas usando HQL devuelven los resultados de las mismas en la forma de objetos o listas de objetos.
- Consultas Polimórficas:
  - √ Podemos declarar el resultado usando el tipo de la superclase y Hibernate se encargara de crear los objetos adecuados de las subclases correctas de forma automática.
- Fácil de Aprender:
  - √ Es muy similar a SQL estándar.
- Soporte para características avanzadas:
  - √ HQL contiene muchas características avanzadas que son muy útiles (fetch joins, inner y outer joins), funciones de agregación (max, avg), y agrupamientos, ordenación y subconsultas.
- Es independiente del gestor de base de datos.

Consultas con HQL

- Una consulta HQL debe ser preparada antes de ser ejecutada.
- La preparación implica varios pasos:
  1. Crear la consulta, con cualquier restricción arbitraria o proyección de los datos que se desean recuperar.
  2. Definición/Paso de variables BIND como parámetros de la Consulta, para poder reutilizar la consulta posteriormente.
  3. Ejecutar la consulta preparada contra la base de datos
  4. Recuperar los datos.
  - Podemos controlar como se ejecuta la consulta y como se deben de devolver los datos (de una vez o por partes, por ejemplo).

Consultas con HQL

1.- Crear la consulta

- Hibernate tiene el interfaz *Query* (org.hibernate.query) que nos da acceso a todas las funcionalidades para poder leer objetos desde la base de datos.
- Necesitamos crear una instancia de uno de estos objetos utilizando *Session*.
- Para crear una nueva instancia de Hibernate Query, se invoca a uno de los métodos CreateQuery () en *Session*.

```
Query query = session.createQuery("SELECT p.nombre FROM Profesor p");
Query query = session.createQuery("FROM Profesor");
```

- Hasta este momento no se ha enviado nada a la BBDD sólo se ha creado la consulta.

("FROM Profesor"); → *Select * en mysql*

Consultas con HQL

1.- Crear la consulta

- La clausula **FROM**
  - √ Es la clausula más simple que existe en Hibernate y la única obligatoria.
  - √ Esta clausula devuelve todos las filas de la tabla indicada que se encuentran en la base de datos.

    FROM hql.modelo.Usuario o FROM Usuario
  - √ Si ponemos mas de una Clase, haremos un producto cartesiano

    FROM Profesor, Asignatura

    FROM Profesor LEFT JOIN Asignatura
- Necesitamos crear ALIAS en las clases para poder hacer referencia a las propiedades de la clase, sino → Excepcion.

  FROM Profesor where Profesor.Id=12; → Excepction

  FROM Profesor p where p.Id=12; → OK

Consultas con HQL

1.- Crear la consulta

- La clausula **WHERE**
  - √ Es la clausula permite el filtrado de la consulta.
  - √ Los literales se encierran entre apostrofes (comillas simples)
- La expresión acepta operadores:
  - √ De comparación: = != <> > >= < <= BETWEEN LIKE IN IS NULL
  - √ Lógicos: AND OR NOT
  - √ Aritméticos: + - * /
  - √ Concatenacion: ||
  - √ Colecciones: SIZE IS EMPTY IS NOT EMPTY MINELEMENT MAXELEMENT
  - √ Subconsultas

    FROM Profesor p WHERE p.nombre='ANTONIO' AND p.ape1 LIKE 'LAR%'
- Dispone de funciones sobre escalares:
  - √ UPPER LOWER CONCAT TRIM SUBSTRING LENGTH ABS SQRT MOD …

Consultas con HQL

1.- Crear la consulta

- Agrupaciones
  - √ Al igual que en SQL se pueden realizar agrupaciones mediante las palabras claves **GROUP BY** y **HAVING**
  - √ Las funciones de agregación que soporta HQL son:
    - – COUNT SUM AVG MIN MAX

FROM Profesor p WHERE p.ape1 LIKE 'LAR%' GROUP BY p.nombre HAVING count(p.nombre)>2

- La clausula **ORDER BY**
  - √ También es posible ordenar los resultados como en SQL.

FROM Profesor p ORDER BY p.nombre, p.ape1 DESC

Consultas con HQL

1.- Crear la consulta

- Paginando resultados
  - √ Cuando realizamos una consulta podemos especificar el número máximo de filas a recuperar, mediante el método *query.setMaxResults(N)*;

```
Query query = session.createQuery("FROM Profesor");
Query.setMaxResults(10);
```

Sólo se recuperarán las 10 primeras filas

  - √ También podemos indicar que valor nos interesa de los recuperados mediante el método *query.setFirstResult(N)*;

```
query.setMaxResults(10);
query.setFirstResult(2);
```
→ *De los 10 resultados devueltos, solo se tratan a partir del 2*

Los elementos se enumeran desde 0

```
Query query = session.createQuery(".....");
query.setMaxResults(tamanyoPagina);
query.setFirstResult(paginaAMostrar * tamanyoPagina);
```

Consultas con HQL

2.- Paso/Definición de variables BIND

- Cuando necesitemos crea una sentencia DINAMICA, deberemos utilizar variables BIND para evitar SQL Inyection.
- Nunca deberemos escribir sentencias como adición de textos

```
String queryString =" from Item i where i.description like ‘  “ + search + “ ‘  “  ;

Search = ‘ uno or 1=1 ‘
```

- La forma correcta de reescritura de esta consulta sería mediante la utilización de **parámetros con nombre**

```
String queryString =" from Item i where i.description like :search“ ;

Query q = session.createQuery ( queryString ).setString ("search", ‘valor’);
                                              .setDate("search", ‘valor’);
                                              .setInteger("search", ‘valor’);
                                              .setParameter("search", ‘valor’);
```


Consultas con HQL

3y4.- Ejecución de la consulta y recuperación de datos

- Una vez que hayamos creado y preparado una consulta, ya estamos listos para para ejecutarlo y recuperar el resultado en la memoria.
- Para recuperar todo el resultado de una sola vez, utilizaremos el método *query.list()*.
- El método *query.list()* ejecuta la sentencia y devuelve el resultado como un objeto *java.util.List*.

```
Query<Profesor>  query = session.createQuery("FROM Profesor");
List <Profesor> profesores = query.list();
```

- Nos devolverá una Lista de instancias de Profesor donde después deberemos tratar las propiedades que deseemos.

Consultas con HQL

3y4.- Ejecución de la consulta y recuperación de datos

- HQL también permite que las consultas devuelvan datos escalares en vez de objetos completos.

  √ *SELECT p.id, p.nombre FROM Profesor p*

- La anterior consulta devolverá el código y el nombre del profesor en vez del Objeto profesor.

- En estos casos el método ***list()*** retorna una lista de Arrays de objetos, un array por fila de resultados con tantos elementos como propiedades hayamos definido en la SELECT.

```
Query query = session.createQuery("SELECT p.id, p.nombre FROM Profesor p");
List<Object[]> listDatos = query.list();

for (Object[] datos : listDatos) {
   System.out.println(datos[0] + "--" + datos[1]);
}
```


Consultas con HQL

3y4.- Ejecución de la consulta y recuperación de datos

- Si en vez de recuperar varias columnas, SOLO se recupera una

  √ *SELECT p.nombre FROM Profesor p*

- En estos casos el método ***list()*** retorna una Lista de objetos

```
Query query = session.createQuery("SELECT p.nombre FROM Profesor p");
List<Object> listDatos = query.list();

for (Object datos : listDatos) {
   System.out.println(datos);
}
```

Consultas con HQL

3y4.- Ejecución de la consulta y recuperación de datos

- Resultados Únicos
  - √ En muchas ocasiones una consulta únicamente devolverá cero o un resultado.
  - √ En ese caso es poco práctico que nos devolver un conjunto de elementos, cuando sólo vamos a recuperar un único elemento.
  - √ Para facilitarnos dicha tarea Hibernate dispone del método ***query.uniqueResult()***.
  - √ Este método sólo se debe de utilizar cuando estemos seguro que sólo nos devolverá 1 registro.

```
Query<Profesor> query = session.createQuery("FROM Profesor  where ……..");
Profesor profesor = query.uniqueResult();
```

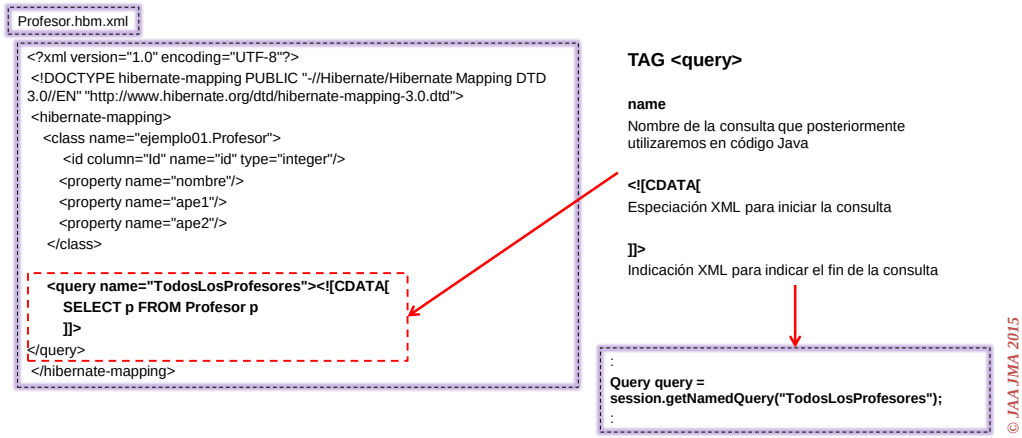

Consultas con Nombre

- Para facilitar la mantenibilidad del software, las consultas a base de datos no deberían escribirse directamente en el código sino que deberían estar en un fichero externo para que puedan modificarse fácilmente.
- Hibernate provee una funcionalidad para hacer esto mismo de una forma sencilla denominada «consultas con nombre».
- Esto lo podemos realizar, añadiendo el TAG **<query>** en el fichero de mapeo correspondiente a la Clase.
- En JPA se realizaría mediante la anotación @NamedQuery:

```
@Entity
@NamedQuery(name="Profesor.findAll", query="SELECT p FROM Profesor p")
public class Profesor { … }
```

Consultas con Nombre

- Este mapeo deberá contener:
  - NOMBRE que posteriormente utilizaremos en el código de Hibernate.
  - CONSULTAR a ejecutar posteriormente.



Recuperación

- Las diferentes estrategias de recuperación se dividen por:
  - Cuando:
    - √ EAGER (la segunda selección se emite inmediatamente)
    - √ LAZY (la segunda selección se retrasa hasta que se necesitan los datos)
  - Como:
    - √ SELECT: Realiza una selección SQL separada para cargar los datos. Esta es la estrategia generalmente denominada N + 1.
    - √ JOIN: Inherentemente un estilo de recuperación EAGER. Los datos que se van a recuperar se obtienen mediante el uso de una unión externa SQL.
    - √ BATCH: Realiza una selección de SQL separada para cargar una cantidad de elementos de datos relacionados utilizando una restricción de IN como parte de la cláusula WHERE de SQL basada en un tamaño de lote.
    - √ SUBSELECT: Realiza una selección de SQL por separado para cargar los datos asociados en función de la restricción de SQL utilizada para cargar el propietario.
  - Donde: (Ámbito de definición)
    - √ Estático: Definidas declarativamente en el mapeo.
    - √ Dinámico:
      - – HQL / JPQL
      - – Perfiles de búsqueda
      - – Grafos de entidades

Optimización de HQL

- Uno de los mayores problemas que nos podemos encontrar al usar un ORM es la lentitud que puede tener respecto a una aplicación realizada directamente con JDBC.
- Con JDBC determinamos cuantas instrucciones SQL lanzamos y como optimizarlas.
- Desde Hibernate, inicialmente, no podemos hacerlo ya que es Hibernate el que se encarga de realizar todas las SQL por nosotros.
- Hay 2 situaciones que debemos atender:
  - √ Lazy Load: "n+1" SELECTs
  - √ Consultas nativas

Optimización de HQL

"n+1" SELECTs

- Es uno de los mayores problemas del ORM.
- Este problema consiste en que al lanzar un consulta con HQL que retorna *n* filas, el ORM lanza n+1 consultas SQL de SELECT.
- Cuando tenemos un registro (profesor) que tiene varios registros asociados (correos), se lanza una consulta por cada correo devuelto.
- Suelen aparecer cuando estamos tratando Asociaciones:
  - √ 1 a N
  - √ N a M
- Ejecutando lo siguiente:

Optimización de HQL

"n+1" SELECTs

```
public class Profesor implements Serializable  {
    private int id;
    private String nombre;
    private String ape1;
    private String ape2;
    private Set<CorreoElectronico> correosElectronicos;
:
```

```
public class CorreoElectronico implements Serializable {
  private int idCorreo;
  private String direccionCorreo;
  private Profesor profesor;
:
```

```
:
Query query = session.createQuery("SELECT p FROM Profesor p");
List<Profesor> profesores = query.list();
 for (Profesor profesor : profesores) {
    System.out.println(profesor.toString());
    for (CorreoElectronico correoElectronico : profesor.getCorreosElectronicos()) {
       System.out.println("\t"+correoElectronico);    }
}
```

```
Hibernate: select profesor0_.Id as Id1_1_, profesor0_.nombre as nombre2_1_, profesor0_.a
Hibernate: select direccion0_.Id as Id1_0_0_, direccion0_.calle as calle2_0_0_, direccio
Hibernate: select direccion0_.Id as Id1_0_0_, direccion0_.calle as calle2_0_0_, direccio
Hibernate: select direccion0_.Id as Id1_0_0_, direccion0_.calle as calle2_0_0_, direccio
Hibernate: select direccion0_.Id as Id1_0_0_, direccion0_.calle as calle2_0_0_, direccio
Hibernate: select direccion0_.Id as Id1_0_0_, direccion0_.calle as calle2_0_0_, direccio
```


214

Optimización de HQL

"n+1" SELECTs

- El lenguaje SQL ESTANDAR no permite consultas jerárquicas.
- La consulta anterior devuelve todos los profesores con todos los correos, repitiendo el profesor tantas veces como correos tenga.
- Hibernate no elimina esa duplicidad así que debemos hacerlo explícitamente desde Java.
- Para evitar la duplicación desde la consulta se dispone de distinct:

```
Query<Profesor>  query = session.createQuery(
        "SELECT DISTINCT p FROM Profesor p LEFT JOIN FETCH p.correosElectronicos");
List <Profesor> profesores = query.list();
```


215

Optimización de HQL

Consultas Nativas

- Otra forma de optimizar las consultas es realizando Consultas Nativas directamente desde Hibernate.
- Pueden ser utilizadas para aprovechar las características propias de la Base de Datos (hints, órdenes propias (top, NVL, etc), procedimientos almacenados, etc)
- La ejecución de consultas SQL nativas se controla por medio de la interfaz *SQLQuery*, la cual se obtiene llamando a *Session.createSQLQuery()*.
- Este tipo de consulta puede devolver 2 tipos de elementos:
  - √ Valores CRUDOS/escalares básicos
  - √ Objetos de Entidades Mapeadas en Hibernate

Optimización de HQL

Consultas Nativas

- Valores CRUDOS/escalares básicos
  - √ Son valores independientes.
  - √ Se devolverá una lista de objetos arrays (Object[]) con valores escalares para cada columna en la tabla

```
Query sqlQuery = Session.createSQLQuery("SELECT *  FROM PROFESOR");
sqlQuery.list();
```

  - √ Podemos especificar las columnas de forma predeterminada

```
Query sqlQuery = Session.createSQLQuery("SELECT ID, nombre  FROM
PROFESOR");
```

O indicando el tipo de Hibernate a devolver

```
Query sqlQuery = Session.createSQLQuery("SELECT * FROM PROFESOR")
.addScalar("ID", Hibernate.LONG)
.addScalar("NOMBRE", Hibernate .STRING);
```

Optimización de HQL

Consultas Nativas

- Objetos de Entidades Mapeadas en Hibernate
  - √ La otra forma de trabajar con las consultas Nativas y es que devuelvan los datos como objetos de Entidades mapeadas en Hibernate.
  - √ Utilizaremos el método ***addEntity*** para obtenerlos
  - √ La siguiente orden devolverá una Lista en donde cada elemento es un objeto de la entidad mapeado

```
Query sqlQuery = Session.createSQLQuery("SELECT * FROM PROFESOR").addEntity(Profesor.class);
sqlQuery.list();

Query sqlQuery = Session.createSQLQuery("SELECT ID, Nombre FROM PROFESOR").addEntity(Profesor.class);
sqlQuery.list();
```


@Fetch

- Además de las anotaciones JPA FetchType.LAZY o FetchType.EAGER, también se puede usar la anotación @Fetch específica de Hibernate que acepta uno de los siguientes FetchMode(s):
  - SELECT (Es equivalente a la estrategia de obtención FetchType.LAZY)
    - √ La asociación se buscará perezosamente utilizando una selección secundaria para cada entidad individual, colección o carga de unión.
  - JOIN: (Es equivalente a la estrategia de obtención FetchType.EAGER)
    - √ Utiliza una combinación externa para cargar las entidades, colecciones o uniones relacionadas cuando se utiliza la obtención directa. de JPA.
  - SUBSELECT
    - √ Disponible solo para colecciones. Se utiliza un segundo SELECT para recuperar las colecciones asociadas de todas las entidades recuperadas en una consulta o recuperación previa. A menos de que deshabilite explícitamente la recuperación perezosa, esta segunda selección sólo se ejecutará cuando se acceda a la asociación. Una consulta por tabla.

```
@OneToMany(mappedBy = "region", fetch = FetchType.LAZY)
@Fetch(FetchMode.SUBSELECT)
private Set<Country> countries;
```

@BatchSize

- Usando la recuperación por lotes, Hibernate puede cargar varios proxies sin inicializar si se accede a un proxy (lectura anticipada). La recuperación en lotes es una optimización de la estrategia de recuperación por selección perezosa.
- A nivel de entidad:

```
@Entity
@BatchSize(size = 5)
public class Profesor implements Serializable { … }
```

- A nivel de colección:

```
@OneToMany(mappedBy = "region", fetch = FetchType.LAZY)
@BatchSize(size = 5)
private Set<Country> countries;
```


Dinámico: Perfiles de búsqueda

- Permiten sustituir dinámicamente las definiciones estáticas.

```
@FetchProfile(
        name = "region.all",
        fetchOverrides = {
            @FetchProfile.FetchOverride(
                entity = Region.class, association = "countries", mode = FetchMode.JOIN
            ),
            @FetchProfile.FetchOverride(
                entity = Country.class, association = "locations", mode = FetchMode.JOIN
            )
        }
    )
public class Region { … }
```

- Para utilizar el perfil se asocia a la sesión:

Dinámico: Grafos de entidades

- Se pueden configurar diferentes grafos de recuperación (grafo de componentes y atributos que cargar) y seleccionar dinámicamente cual utilizar en la consulta. Pueden anular una asociación de recuperación de LAZY.

```
@NamedEntityGraph(name = "region.countries+locations",
    attributeNodes = @NamedAttributeNode(
                        value = "countries", subgraph = "region.countries.locations"),
    subgraphs = @NamedSubgraph(
                        name = "region.countries.locations",
                        attributeNodes = @NamedAttributeNode("locations")))
public class Region { … }
```

- Para utilizar un grafo en una consulta:

```
Region region = em.find(Region.class, 1L, Collections.singletonMap("javax.persistence.fetchgraph",
            em.getEntityGraph("region.countries+locations");
    ));
List rslt = em.createQuery("from Region")
            .setHint("javax.persistence.fetchgraph", em.getEntityGraph("region.countries+locations"))
            .getResultList();
```


Consultas de solo lectura

Al igual que la inmutabilidad de entidades, la obtención de entidades en modo de solo lectura es mucho más eficiente que la obtención de entidades de lectura-escritura.

```
List rslt = entityManager.createQuery(" … ", Profesor.class )
    .setHint( "org.hibernate.readOnly", true )
    .getResultList();
```

También se puede pasar la sugerencia de solo lectura a las consultas con nombre usando la anotación @QueryHint de JPA.

```
@NamedQuery(
   name = "get_read_only_profesores",
   query = "select p from Profesor",
   hints = {  @QueryHint(name = "org.hibernate.readOnly", value = "true“) }
)
```

DML con HQL

- También podemos realizar operaciones DML con HQL
- Deberemos utilizar el método *executeUpdate* del objeto *session*.

```
session.beginTransaction();
session.createQuery("update Profesor p set p.nombre=UPPER(p.nombre) ").executeUpdate();
session.getTransaction().commit();
```

```
session.beginTransaction();
session.createQuery("delete from Profesor p where p.id=8").executeUpdate();
session.getTransaction().commit();
```


Procesamiento por lotes

- Inserciones masivas de datos son operaciones que NO deben de ser realizadas a través de Hibernate.
  - Por ejemplo, un usuario puede tener 100.000 registros que tienen que ser importados en una sola tabla.

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
for ( int i=0; i<100000; i++ ) {
   Customer customer = new Customer(.....);
   session.save(customer);
}
tx.commit();
session.close();
```

  - Este código podría generar una excepcion *OutOfMemoryException*.
  - Hibernate mantiene en caché todas las instancias recién insertadas a nivel de sesión.

Procesamiento por lotes

- Si se puede es mejor realizarlo de forma externa o mediante JDBC con operaciones BULK.
- Si, por alguna razón, debemos de hacer la importación masiva a través de Hibernate, deberemos tener en cuenta unas recomendaciones:
  - Utilización de hibernate.jdbc.batch_size
  - Utilizar *Session.flush()* y *Session.clear()* regularmente
  - Considerar la utilización de *Session.commit;*
  - Utilizar una Session sin estado

Procesamiento por lotes

Utilización de hibernate.jdbc.batch_size

- La primera recomendación para un procesamiento masivo de filas (batch processing), es habilitar **el uso del lote JDBC.**
- Deberemos de introducirlo dentro del fichero de configuración de Hibernate (hibernate.cfg.xml)

```
<property name="hibernate.jdbc.batch_size">20</property>
```

- El valor recomendable es entre 10 y 50
- Este valor sólo se debe de indicar cuando estemos seguro que una transacción realiza Inserciones/Actualizaciones de filas en conjunto.
- El valor a definir sería el mayor número de filas que intervienen en una operación DML.

Procesamiento por lotes

Utilizar *Session.flush() y Session.close();*

- La utilización de *Session.flush() o Session.close ()* permite controlar la memoria cache de la Session.
- Hibernate "limpia" (Flush) la memoria de la Session, de forma automática, después de la realización de una transacción.
- Session.flush()
  - √ Fuerza la sincronización entre la Memoria de la Session y la Base de Datos, es decir, obliga a escribir los cambios en la BBDD
- Session.clear()
  - √ Fuerza la eliminación de las diferentes instancias creadas/modificadas que están en la Memoria de la Session

Procesamiento por lotes

Utilizar *Session.flush() y Session.close();*

- Al hacer persistentes los objetos nuevos, es necesario que realice ***flush()*** y luego ***clear()*** en la sesión regularmente para controlar el tamaño del caché de primer nivel.

```
for ( int i=0; i<100000; i++ ) {
   Customer customer = new Customer(.....);
   session.save(customer);
   if ( i % 20 == 0 ) {
      session.flush();
      session.clear();
   }
}
```

Procesamiento por lotes

Desplazamiento de sesión

- Además se deberia utilizar el método scroll() para aprovechar los cursores del lado del servidor para las consultas que devuelven muchas filas de datos.

```
ScrollableResults scrollableResults = null;
try {
        em = emf.createEntityManager();
        em.getTransaction().begin();
        scrollableResults = em.unwrap(Session.class).createQuery("from Person")
                .setCacheMode(CacheMode.IGNORE).scroll(ScrollMode.FORWARD_ONLY);
        int batchSize = 25; int count = 0;
        while (scrollableResults.next()) {
                processPerson((Person) scrollableResults.get(0));
                if (++count % batchSize == 0) { em.flush(); em.clear(); }
        }
        em.getTransaction().commit();
} catch (RuntimeException e) {
        if (em.getTransaction().isActive()) em.getTransaction().rollback();
        throw e;
} finally {
        if (scrollableResults != null) scrollableResults.close();
        if (em != null)  em.close();
}
```


Procesamiento por lotes

Considerar la utilización de *Session.commit;*

- Podemos utilizar *Session.commit* para validar la transacción por partes en vez de realizarla de forma completa.

- Ventajas
  - √ Grabación de datos ante posibles caídas de BBDD/Aplicación, etc.
  - √ Liberación de memoria Cache de la Session.

- Inconvenientes
  - √ Deberemos estar seguro que la transacción se puede validar por partes y no de forma completa.

```
for ( int i=0; i<100000; i++ ) {
   Customer customer = new Customer(.....);
   session.save(customer);
   if ( i % 20 == 0 ) { session.commit; }
}
```

Procesamiento por lotes

StatelessSession es una API orientada a comandos proporcionada por Hibernate para transmitir datos hacia y desde la base de datos en forma de objetos separados sin un contexto de persistencia asociado y no proporciona muchas de las semánticas de ciclo de vida de nivel superior.

```
try {
    statelessSession = emf.unwrap( SessionFactory.class).openStatelessSession();
    statelessSession.getTransaction().begin();
    scrollableResults = statelessSession
        .createQuery( "select p from Person p" )
        .scroll(ScrollMode.FORWARD_ONLY);
    while ( scrollableResults.next() ) {
        processPerson((Person) scrollableResults.get( 0 ));
        statelessSession.update( Person );
    }
     statelessSession.getTransaction().commit();
} catch (RuntimeException e) {
    if (statelessSession.getTransaction().getStatus() == TransactionStatus.ACTIVE)
        statelessSession.getTransaction(). rollback();
        throw e;
} finally {
    if (scrollableResults != null)  scrollableResults.close();
    if (statelessSession != null)  statelessSession.close();
}
```


Personalización de DML

- Hibernate nos da la posibilidad de indicarle en su fichero de configuración el modelo de orden DML que se ejecutará.
- Esto nos permite añadir elementos adicionales propios del gestor.
- El fichero de mapeo de Hibernate puede incluir los siguientes tags para especificarlo

  <sql-insert> : Contiene una sql de INSERT
  <sql-update> : Contiene una sql de UPDATE
  <sql-delete> : Contiene una sql de DELETE

Hibernate.cfg.xml

```
:
<sql-insert>  INSERT INTO Profesor (Nombre,Ape1,Ape2,Id) VALUES (?,?,?,?)</sql-insert>
<sql-update>UPDATE Profesor SET Nombre=?,Ape1=?,Ape2=? WHERE Id=? </sql-update>
<sql-delete>  DELETE FROM Profesor WHERE Id=?</sql-delete>

 </class>
```

Personalización de DML

- Al ser definiciones SQL Nativas se podría usar cualquier característica especifica que necesitemos de la base de datos que estemos usando.

Hibernate.cfg.xml

```
:
<sql-insert>  INSERT /*+ APPEND */ INTO Profesor (Nombre,Ape1,Ape2,Id) VALUES (?,?,?,?)</sql-insert>

<sql-update>UPDATE /*+ NO_INDEXES */ Profesor SET Nombre=?,Ape1=?,Ape2=? WHERE Id=? </sql-update>

<sql-delete>  DELETE FROM Profesor WHERE Id=?</sql-delete>

 </class>
```

- Cuando realicemos una orden DML, nos aparecerá algo similar a:

```
Hibernate: INSERT /*+ APPEND */ INTO Profesor (Nombre,Ape1,Ape2,Id) VALUES (?,?,?,?)
Hibernate: UPDATE /*+ NO_INDEXES */ Profesor SET Nombre=?,Ape1=?,Ape2=? WHERE Id=?
Hibernate: DELETE FROM Profesor WHERE Id=?
```

- JPA dispone de sus propias anotaciones @SQLInsert, @SQLUpdateY @SQLDelete para anular el INSERT, UPDATE, DELETE de una entidad determinada.

Uso de procedimientos almacenados

Hibernate proporciona soporte para consultas a través de procedimientos y funciones almacenados.

- Los argumentos de un procedimiento almacenado se declaran utilizando el modo de parámetro IN, el resultado puede marcarse con el modo OUT, un REF_CURSOR o simplemente podría devolver el resultado como una función.

```
StoredProcedureQuery query = entityManager.createStoredProcedureQuery( "sp_calc_value")
      .registerStoredProcedureParameter( "personId", Long.class, ParameterMode.IN);
      .registerStoredProcedureParameter( “rslt", Long.class, ParameterMode.OUT)
      .setParameter("personId", 1L)
      .execute();
Long value = (Long) query.getOutputParameterValue(“rslt");

List<Object[]> list = entityManager.createStoredProcedureQuery( "sp_phones")
      .registerStoredProcedureParameter( 1, Long.class, ParameterMode.IN)
      .setParameter(1, 1L)
      .getResultList();
```

API DE CRITERIA

Introducción

- Criteria API es una alternativa poderosa y elegante a HQL.
- Está especialmente diseñado para las funcionalidades de búsqueda WEB dinámica donde las consultas se hacen ‘online’.
  - √ Casillas de Verificación
  - √ Campos de entrada
  - √ Cambios de Usuarios.
- JPA no soporta Criteria API.
- Criteria permite crear consultas en programación mediante la creación y combinación de objetos en el orden correcto.

Introducción

- Criteria permite utilizar filtros y condiciones lógicas para la construcción de la consulta.

- La interfaz *Session* proporciona un método **createCriteria ()** para trabajar con objetos Criteria.

- Enfoque HQL:
  - la implementación de una consulta HQL multi-criterios implica la construcción de la misma en base a los criterios de búsqueda introducidos por el usuario.

```
queryBuf.append (firstClause "dónde": "y");
queryBuf . append ("s.date> =: startDate");
queryBuf.append (firstClause "dónde": "y ");
:
```

- Enfoque Criteria:

```
Criteria Criterios = session.createCriteria (Sale.class);
if (startDate! = null) { criteria.add (Expression.ge ("fecha", startDate); } i
f (endDate! = null) { criteria.add (Expression.le ("fecha", endDate); }
```


Creación de objetos Criteria

- La interfaz *Session* proporciona un método ***createCriteria ()*** que puede ser utilizado para crear un objeto Criteria.

```
Criteria criteria = session.createCriteria(Profesor.class);
```

- El objeto creado, podrá devolver todas las instancias de la clase especificada cuando ejecutemos la sentencia (list).

- Podemos crear objetos Criteria sin tener una Session; utilizaremos la clase *DetachedCriteria* para hacerlo y posteriormente añadirlo a una *Session*.

```
DetachedCriteria crit = DetachedCriteria.forClass(User.class)
:
List result = crit.getExecutableCriteria(session).list();
```

Creación de objetos Criteria

- Una vez creado el objeto Criteria, procederemos a filtrar/configurar nuestra consulta.

- Podemos
  - Añadir restricciones mediante el método .*add(Expresion)*
  - Utilizar métodos para devolución de un número máximo de valores o sólo las N primeras instancias.
  - Realizar ordenaciones en la obtención de datos

- Podemos añadir todas las restricciones/filtros que deseemos, siempre y cuando sean compatibles entre ellas.

Gestión de objetos Criteria

- Ordenaciones
  - El API de Criteria proporciona la clase *org.hibernate.criterion.Order* para ordenar su conjunto de resultados, de acuerdo con una de las propiedades de su objeto.

  - Esta ordenación se hace en el mismo momento de realizar la ejecución de la consulta.

  - Utilizaremos el método *criteria.addOrder(….)*

    √ criteria.addOrder (tipo_ordenacion (Columna))

    criteria.addOrder (Order.desc ("Fecha"));

    session.createCriteria(Profesor.class).addOrder(Order.asc("ape1")).addOrder(Order.asc("ape2"));

  - Ninguna instancia ha sido devuelta aún.

Gestión de objetos Criteria

- Paginación
  - El API de Criteria define 2 métodos para acotar la devolución de resultados:

  - *setFirstResult(int PrimeraFila)*
    - √ El número entero (PrimeraFila) define cual es la primera instancia a devolver.
    - √ El valor comienza desde 0.

  - *setMaxResults(int maxResults)*
    - √ maxResults indica a Criteria cual es el número máximo de instancias a devolver.

```
Criteria cr = session.createCriteria(Profesor.class);
cr.setFirstResult(1);
cr.setMaxResults(10);
:
List results = cr.list();          // Devuelve 10 resultados desde la instancia segunda (1)
```


Gestión de objetos Criteria

- Varios
  - *setCacheable (boolean)*
    - √ Habilitar el almacenamiento en caché de este resultado de la consulta, si la caché de consultas está activada.

  - *uniqueResult()*
    - √ Obliga a la consulta a devolver una sola instancias que coincida con la petición de la consulta.
    - √ Si la consulta no devuelve ningún resultado → NULL.

  - *setTimeout(int timeout)*
    - √ *Configura un tiempo máximo de espera para la consulta JDBC.*

Objetos Criteria. Expresiones

- Expresiones
  - El API de Criteria proporciona la clase *org.hibernate.criterion.Restrictions* la cual define métodos para incorporar Restricciones más específicas a nuestra consulta.

  - Disponemos restricciones referentes a:
    - √ Operadores estándar SQL **(=, <,<= >, >=) eq(), lt(), le(), gt(), ge()**
    - √ isNull, isNotNull
    - √ isEmpty, isNotEmpty
    - √ Or
    - √ Between
    - √ Like
    - √ In
    - √ eqProperty, ltProperty, gtProperty

  - Estas restricciones pueden ser añadidas al objeto CRITERIA en base a:
    - √ Mediante la creación de un objeto CRITERION
    - √ Directamente al objeto Criteria.

248

Objetos Criteria. Expresiones

- Expresiones. Objeto CRITERION

  - La creación de un Objeto CRITERION, permite la definición de una restricción que puede ser utilizar en 1 o en varios objetos Criteria.

  - Deberemos crearlo siguiendo la sintaxis:

    - √ *Criterion restriction = Restrictions.XXXXXX*

  - Una vez creado, podemos asignárselo al objeto Criteria que elijamos mediante el método *.add(objeto_criterion)*

```
Criterion restriction = Restrictions.between("amount",new BigDecimal(100), new BigDecimal(200) );

Criteria cri = session.createCriteria(Bid.class);
cri.add(restriction);
```


249

Objetos Criteria. Expresiones

- Expresiones. Directamente en Criteria
  - Si la restricción que vamos a crear sólo se aplica a un objeto Criteria, no es necesario la creación de un objeto CRITERION.
  - Podemos crear la restricción de forma online aplicando dicha restricción en el momento.

```
Criteria cri = session.createCriteria(Bid.class);

cri.add ( Restrictions.between("amount",new BigDecimal(100), new BigDecimal(200) );
```

  - Esta forma de aplicar restricciones crear un Objeto CRITERION sin referencia y sólo se puede utilizar en esta asignación.

Objetos Criteria. Expresiones

- Expresiones. Operadores estándar SQL
  - Podemos utilizar los operadores estándar de SQL**(=, <,<= >, >=)**, pero debemos de utilizarlos en formato texto y no el símbolo

Operación	Símbolo	Operador
Igual	=	**eq()**
Menor que	<	**lt()**
Menor o igual	<=	**le()**
Mayor que	>	**gt()**
Mayor o igual	>=	**ge()**

  - Estos operadores están disponibles como métodos del objeto *Restrictions (Expression deprecated)* y lo añadimos mediante el método .*add* al objeto Criteria

```
Criteria cr = session.createCriteria(Employee.class);
cr.add(Restrictions.XX ("COLUMNA", VALOR ) );
```

Objetos Criteria. Expresiones

- Expresiones. Operadores estándar SQL
  - EJEMPLOS

```
Criteria cr = session.createCriteria(Employee.class);

    cr.add(Restrictions.gt("salary", 2000));
    // Registros cuyo salario sea mayor que 2000

    cr.add(Restrictions.lt("salary", 2000));
    // Registros cuyo salario sea menor que 2000

    cr.add(Restrictions.eq("fecha", new  Date() ));
    // Registros cuya fecha sea  igual a la actual

    cr.add(Restrictions.eq("Nombre", "Jose"  ));
    // Registros cuya nombre sean Jose
```


Objetos Criteria. Expresiones

- Expresiones. isNull, iSNotNull, isEmpty, isNotEmpty
  - Para hacer el tratamiento de las columnas nulas o no Nulas de las Base de datos disponemos de las restricciones **isNull** y **isNotNull**
  - Para conocer si un elemento está vacío o no, disponemos de **isEmpty** y **isNotEmpty**

```
Criteria cr = session.createCriteria(Employee.class);

    cr.add(Restrictions.isNull("salary"));

    cr.add(Restrictions.isNotNull("salary"));

    cr.add(Restrictions.isEmpty("salary"));

    cr.add(Restrictions.isNotEmpty("salary"));
```

Objetos Criteria. Expresiones

- Expresiones. Or
  - Podemos crear expresiones lógicas OR y AND
  - Debemos realizar 2 pasos
    1. Crear los objetos CRITERION para realizar de forma correcta la comparativa.
    2. Crear un objeto LogicalExpresion para agrupar la Expresión Lógica

```
Criteria cr = session.createCriteria(Employee.class);

Criterion salary = Restrictions.gt ("salary", 2000);
Criterion dept1 = Restrictions.eq ("dept", 10);

LogicalExpression orExp = Restrictions.or(salary, dept1);

cr.add( orExp );
```

Creación de los objetos para luego hacer las comparaciones

Creación de *LogicalExpression* y aplicación de OR

Añadir la Restricción

Equivalente a => *where salary > 2000 or dept=10*

Objetos Criteria. Expresiones

- Expresiones. Or
  - Podemos crear expresiones lógicas OR y AND
  - También podemos definir las expresiones lógicas de la siguiente forma:

```
Criteria cr = session.createCriteria(Employee.class);

cr.add( Restrictions.or(
                Restrictions.and(
                        Restrictions.like("firstname", "G%"),
                        Restrictions.like("lastname", "K%")
                        ),
                Restrictions.in("email", emails)
                )
```

Objetos Criteria. Expresiones

- Expresiones. Between, Like e IN
  - También disponemos de la posibilidad de añadir restricciones de rangos (BETWEEN), patrones de búsqueda (LIKE) y valores concretos dentro de listas (IN)

```
Criteria cr = session.createCriteria(Employee.class);

        cr.add(Restrictions.like ("Nombre" , "J%" ));

        cr.add(Restrictions.between ("Salario" , 1000, 2000 ));
        cr.add(Restrictions.between ("Salario" , new BigDecimal(100), new BigDecimal(200) ));

        cr.add(Restrictions.between ("Fecha" , FechaInicial, new Date() ));

String[] emails = { "foo@hibernate.org", "bar@hibernate.org" };
        cr.add(Restrictions.in ("CorreoElectronico" , emails ));
```


Objetos Criteria. Expresiones

- Expresiones. eqProperty, ltProperty, gtProperty
  - También tenemos la posibilidad de comparar 2 propiedades determinadas, mediante los métodos:
    - √ eqProperty, ltProperty, gtProterty

```
Criteria cr = session.createCriteria(Employee.class);

        cr.add ( Restrictions.eqProperty("firstname", "username") )

        cr.add ( Restrictions.ltProperty("FechaExamen", "FechaAcceso") )
```

sqlRestriction

- La API de Criteria no tiene un equivalente a las funciones SQL nativas como LENGTH (), UPPER, LOWER, etc
- Criteria puede añadir este tipo de restricciones añadiendo Restricciones especiales tipo *Restrictions.sqlRestriction*.

```
Criteria cr = session.createCriteria(Employee.class);

cr..add( Restrictions.sqlRestriction (" length ( {alias}.PASSWORD ) < 5 ");

cr..add( Restrictions.sqlRestriction (
// v3.6-v4.1       " length ( {alias}.PASSWORD ) < ? ", 5, Hibernate.INTEGER );
// v4.2 sup        " length ( {alias}.PASSWORD ) < ? ", 5, org.hibernate.type.StandardBasicTypes.INTEGER);

                   {alias}              → Employee
                   ?                    → Variable Bind que es sustituido por 5
                   Hibernate.INTEGER → Especifica el tipo del parámetro utilizado
```


sqlRestriction

- Este tipo de configuración, es también utilizado para la creación de subconsultas dentro de una restricción

```
Criteria cr = session.createCriteria(Employee.class);

    cr..add( Restrictions.sqlRestriction (
            " 'salary' > all" +
            " ( select b.AMOUNT from BID b where b.ITEM_ID = {alias}.ITEM_ID )"
            )
```

Varios

- Además de todo lo anterior, la API de Criteria puede realizar operaciones de:
  - Especificar restricciones en entidades relacionadas (1aN)
    - √ Definir una Restricción en una tabla (1) y a la vez crear una Restricción en la Tabla Relacionada (N)
  - Recuperación dinámica de Asociaciones mediante ***setFetchMode()***.
    - √ Cuando recuperamos la principal, podemos a la vez recuperar todas las colecciones asociadas a la principal.
    - √ Seria como realizar un JOIN entre ambar tablas

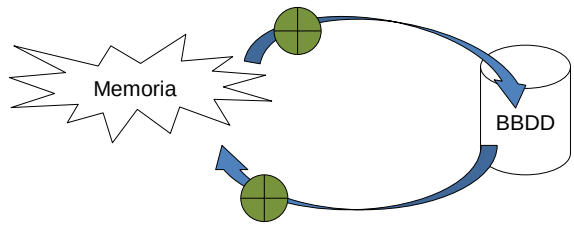
      .setFetchMode("profesor", FetchMode.EAGER)
  - Proyecciones, agregación y agrupamiento.
    - √ Nos permite obtener Funciones de Agrupamiento (SUM, AVG, MAX, ROWCOUNT) de la tabla principal

      .setProjection(Projections.rowCount())
      .setProjection(Projections.avg(" salary "))
      .setProjection(Projections.max(" salary "))
    - √ Sólo podemos definir una por operación

CICLO DE VIDA CON OBJETOS DE ENTIDAD

Introducción al ciclo de vida

- Hibernate utiliza un mecanismo transparente de persistencia y también una gestión optimizada de los objetos involucrados en ellos.
- Si entendemos el ciclo de vida de los objetos, podremos afinar más nuestras aplicaciones.
- Cualquier aplicación con persistente debe interactuar con el servicio de persistencia (Hibernate) cada vez que necesita propagar el estado que tiene en la memoria a la base de datos (o viceversa).



Introducción al ciclo de vida

- Nos referimos a ciclo de vida de la persistencia como los estados de un objeto atraviesa durante su vida.
- Todo objeto en Hibernate tiene un contexto de persistencia.
- Contexto de Persistencia
  - Es similar a una caché que recuerda todas las modificaciones y cambios de estado realizados en los objetos de una unidad de trabajo.
  - No es visible por la aplicación y tampoco puede ser llamado.
- El contexto de persistencia está definido por:
  - El objeto *Session* en una Aplicación Hibernate
  - El objeto *EntityManager* en una Aplicación Java Persistence

Introducción al ciclo de vida

- Cuando se utilizan "conversaciones" el contexto de persistencia es crítico.
  - Unidades de trabajo de ejecución larga que requieren *tiempo-para-pensar* por parte del usuario.

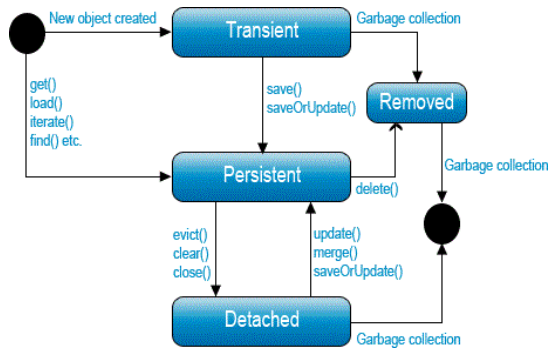
- ¿Cómo funciona el contexto?
  - √ Se crea con la Aplicación Hibernate / Aplicación Java Persistent. (Session)
  - √ Cuando una solicitud del usuario ha sido procesado, la aplicación se desconecta de la BBDD pero el contexto de persistencia no se cierra.
  - √ Esta desconexión dura el *tiempo-para-pensar* por parte del usuario.
  - √ Cuando el usuario continúa en la conversación, el contexto de persistencia se vuelve a conectar a la base de datos, y la siguiente petición pueda ser procesada.
  - √ Al final de la conversación, el contexto de persistencia está sincronizado con la base de datos y cerrado. (Session.close())

Introducción al ciclo de vida

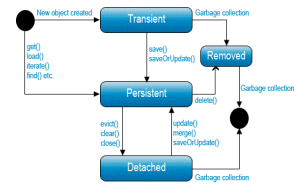
- ¿Por qué utilizar un Contexto de Persistencia?
  - Hibernate puede, de forma automática, chequear los datos "sucios" y realizar una escritura en background.
    - √ Por ejemplo, antes de realizar una consulta SQL, con el fin de sincronizar datos correctamente.
  - El contexto puede ser utilizado como una caché de primer nivel.
    - √ Podemos aprovechar las lectura de entidades anteriores y la ventaja de su reutilización.
  - Hibernate puede garantizar un ámbito de identidad del objeto Java.
    - √ Dentro de un contexto, sólo existe un objeto con la misma identidad.
  - Hibernate puede ampliar el contexto de persistencia en cualquier momento, añadiendo nuevas entidades o conversaciones.

Ciclo de vida. Estados

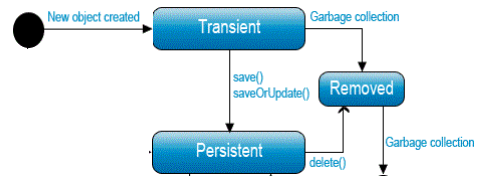
- Hibernate define y soporta los siguientes estados de objeto:
  - Transient (Transitorio)
  - Persistent (Persistente)
  - Detached (Separado)
  - Removed (Borrado)



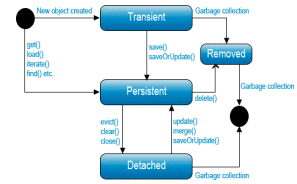
Ciclo de vida. Estados



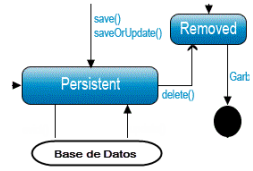
- Estado Transient (Transitorio)
  - un objeto es transitorio si ha sido instanciado utilizando el operador **new**, y no está asociado a una Session de Hibernate.
  - No tiene una representación persistente en la base de datos y no se le ha asignado un valor identificador.
  - Las instancias transitorias serán destruidas por el recolector de basura si la aplicación no mantiene más de una referencia.
  - Para convertir un objeto Transient → Persistent, utilizaremos Session.save() o Session.saveOrUpdate()



Ciclo de vida. Estados

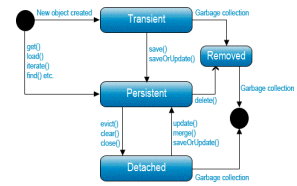


- Estado Persistent (Persistente)
  - Una instancia persistente tiene una representación en la base de datos y un valor identificador.
  - Los objetos se convierten en persistentes cuando se recuperan de la base de datos, se llaman a los métodos del gestor (save, saveOrUpdate) o cuando se crea una referencia de otro objeto persistente ya gestionado.
  - Todas las instancias Persistentes se encuentran dentro del ámbito de Session y disponen de un identificador único (ID).
  - Hibernate detectará cualquier cambio realizado a un objeto en estado persistente y sincronizará el estado con la base de datos cuando se complete la unidad de trabajo.

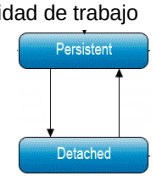


272

Ciclo de vida. Estados

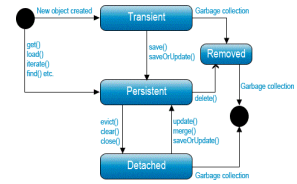


- Estado Detached (Separado)
  - Una instancia separada es un objeto que se ha hecho persistente, pero su Session ha sido cerrada.
  - La referencia al objeto todavía es válida, y podría ser modificada en este estado
  - Una instancia separada puede ser re-unida a una nueva Session más tarde, haciéndola persistente de nuevo (con todas las modificaciones).
  - Este aspecto habilita un modelo de programación para unidades de trabajo de ejecución larga que requieren *tiempo-para-pensar* por parte del usuario.
  - Las llamamos *transaccciones de aplicación*, por ejemplo, una unidad de trabajo desde el punto de vista del usuario.

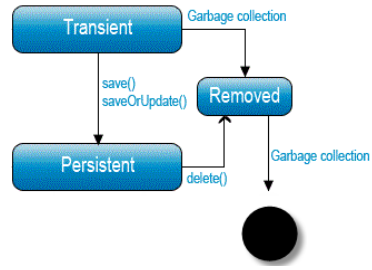


273

Ciclo de vida. Estados



- Estado Removed (Borrado)
  - Puede eliminar una instancia de entidad de varias maneras:
    - √ Llamando a una operación explícita del gestor de persistencia (delete).
    - √ De forma implícita, cuando se quitan cuando todas las referencias a la misma instancia, el recolector de basura las elimina directamente.
    - √ Los elementos borrados son marcados para su borrado; un objeto eliminado no se debe volver a utilizar, ya que se eliminará de la base de datos tan pronto como la unidad de trabajo se complete.



Trabajo con objetos persistentes

- En una aplicación de Hibernate, almacenar (store) y cargar (load) objetos se realiza básicamente mediante un cambio en su estado.
- Esto se hace en unidades de trabajo.
  - Una unidad de trabajo es un conjunto de operaciones que e ejecutan de modo atómico.
- Las unidades de trabajo son "similares" a las transacciones pero no idénticas.

```
Comienzo de la unidad de trabajo
Operaciones
:
Transacción1
Transacción2
:
Fin de la unidad de trabajo
```

Trabajo con objetos persistentes

Unidad de Trabajo

- Una unidad de trabajo comienza con la obtención de una instancia de la **sesión** de SessionFactory de la aplicación:

  - √ *SessionFactory sessionFactory = configuration.buildSessionFactory(serviceRegistry);*
  - √ ***Sesión session = sessionFactory.openSession ();***
  - √ *Transacción tx = sesion.beginTransaction ();*

- En este punto, un nuevo contexto de persistencia es creado y se encargará de todos los objetos con los que trabaja en dicha Sesión.

- La aplicación puede tener múltiples *SessionFactorys* si accede a varias bases de datos.

NOTAS:

- √ La creación de una ***sessionFactory*** es muy caro
- √ La creación de una ***sesión*** es barato.

Trabajo con objetos persistentes

Unidad de Trabajo

- Todas las operaciones que se producen dentro de una Unidad de Trabajo están dentro de una transacción.

- Las transacciones puede comenzar en cualquier punto de la unidad de trabajo.

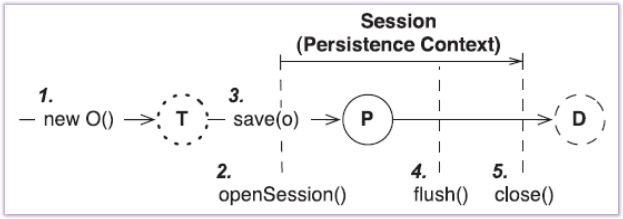
- Después de abrir un nuevo contexto de sesiones y persistencia, almacenaremos y leeremos objetos.

Trabajo con objetos persistentes

Creando objetos Persistentes

- Cuando creamos un nuevo objeto mediante new, el objeto es *Transient*.
- Para hacerlo persistente utilizamos el método *session.save(objeto)*.

```
Profesor profesor1=new Profesor(3, "Sergio", "Mateo", "Ramis");  1
Session session=sessionFactory.openSession();  2
session.beginTransaction();
session.save(profesor1);  3
session.getTransaction().commit();  4
session.close();  5
```



1.- Creación de elemento Transient

2.- Obtención de session y comienzo de una transacción

3.- Llamada a ***save***, para hacer persistencia. Ahora se asocia a la sesión actual y contexto de persistencia

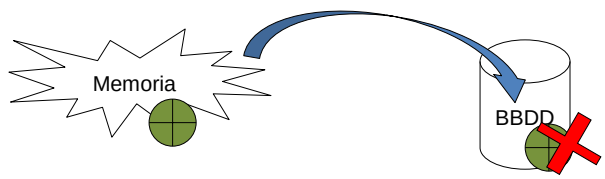
4.- Los cambios realizados en el objeto son sincronizados con la BBDD.
También se realiza un flush() de la memoria

5.- Sesión cerrada y contexto de persistencia cerrado.
El objeto ahora está en estado separado.

Trabajo con objetos persistentes

Objetos Persistentes

- Debemos de tener cuidado con las transacciones.
- Si una de las sentencias implicadas en una transacción falla:
  - √ Transacciones de BBDD
    - – Se deshace toda la transacción a nivel de Base de Datos
  - √ Transacciones Hibernate
    - – Hibernate no deshace los cambios en la memoria a los objetos persistentes.



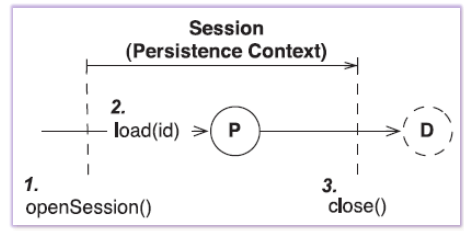
Trabajo con objetos persistentes

Devolviendo objetos Persistentes

- El objeto *session* también permite consultar la base de datos y recuperar objetos persistentes existentes.
- Hay 2 métodos para la recuperación de objetos persistentes:
  - √ *get(objeto….)*
  - √ *load(objeto….)*
- En ambos métodos el elemento de objeto recuperado está en estado persistente y cuando el contexto de persistencia se cierra, pasa a estado separado.

La única diferencia entre **get() y load()** es la forma en que indican que la instancia no se pudo encontrar

GET
  Devuelve NULL
LOAD
  Devuelve excepción
  ObjectNotFound-Exception

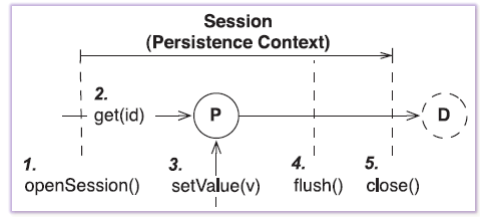


Trabajo con objetos persistentes

Modificando objetos Persistentes

- Cualquier objeto persistente devuelto por get(), load(), o cualquier entidad consultada ya asociada con la sesión actual y contexto de persistencia se puede modificar.
- Posteriormente deberemos sincronizarla con la con la Base de Datos. (*commit*)

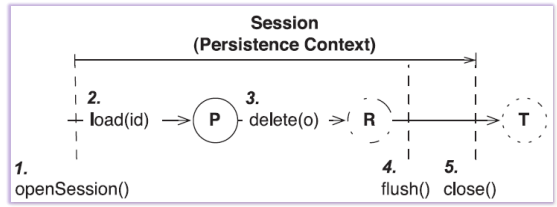
```
Session session = sessionFactory.openSession();
  Transaction tx = session.beginTransaction();
  Item item = (Item) session.get(Item.class, new Long(1234));
  item.setDescription("This Playstation is as good as new!");
  tx.commit();
session.close();
```



Trabajo con objetos persistentes

Eliminación de objetos Persistentes

- Mediante el método *Session.delete(objeto)* podemos hacer que el estado persistente de un objeto desaparezca.
- Después de la llamada al método ***delete***, el trabajo con este objeto es imposible.
- La orden SQL DELETE sólo es ejecutada cuando el contexto de persistencia es sincronizado con la Base de Datos.
- Después de la sesión se cierra, el objeto se considera una instancia transitoria ordinaria y es destruida por el recolector de basura cuando ya no está referenciada por cualquier otro objeto



Trabajo con objetos persistentes

Eliminación de objetos Persistentes

¿Deberemos cargar el objeto para ser borrado? Si y no

- SI
  - √ Un objeto debería que ser cargado en el contexto de persistencia para ser borrado, es decir, tiene que ser persistente.
  - √ ¿Porque?

    Todo objeto debe de estar dentro del ciclo de vida para su correcto tratamiento.

```
Session session = sessionFactory.openSession();
session.beginTransaction();
Profesor profesor=(Profesor) session.get(Profesor.class, 101);
session.delete(profesor);
    session.getTransaction().commit();
session.close()
```

- NO
  - √ Podemos ejecutar SQL Nativo desde HQL o Criteria sin pasar por el ciclo de vida del objeto

```
session.beginTransaction();
session.createQuery("delete from Profesor p where p.id=8").executeUpdate();
session.getTransaction().commit();
```

Trabajo con objetos Separados

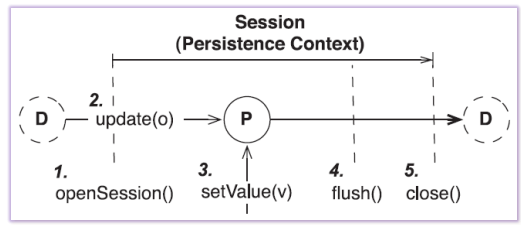
- Si modificamos un objeto separado (detached), esta modificación no persiste en la Base de Datos.
- Tan pronto como el contexto de persistencia está cerrado, el elemento se convierte en una instancia individual.
- Si desea guardar las modificaciones que haya realizado en un objeto individual, tenemos que volverlo a introducir en un contexto de persistencia de nuevo.
- Llamando al método *update* añadimos un objeto separado a una *session* existente.

Session.update(objeto_separado)

Trabajo con objetos Separados

- Hibernate siempre trata el objeto recién añadido como "sucio" y se forzará la actualización del estado de persistencia del objeto en la Base de Datos.

```
item.setDescription(...); // Leido en una sesion anterior

Session sessionTwo = sessionFactory.openSession();
Transaction tx = sessionTwo.beginTransaction();
sessionTwo.update(item);
item.setEndDate(...);
tx.commit();
sessionTwo.close();
```



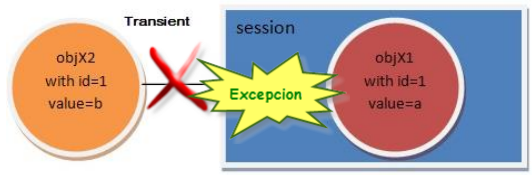
Métodos de Sesión

- El objeto session dispone de métodos adicionales para el control de los objetos en el contexto de persistencia.
- Estos métodos pueden ser utilizados de forma "manual" para nuestra gestión.
  - Session.evict(objeto)
    - √ Se utiliza para separar, manualmente, una instancia persistente del contexto de persistencia.
  - Session.clear()
    - √ Se utiliza para separar todas las instancias persistentes del contexto de persistencia.

```
Session session = sessionFactory.openSession();
Profesor profesor =(Profesor)session.get(Profesor.class,1001);
  session.evict(profesor);
session.close();
```


Métodos de Sesión

- Session.merge(objeto)
  - √ ¿Qué ocurre cuando intentamos añadir un Objeto *Transient* a una *session* que ya tiene un objeto persistente con el mismo ID?



Para solucionar este problema, utilizamos MERGE

- √ Fusiona un objeto separado (detached) con otro persistente con el mismo ID.
- √ Si no hay objeto persistente con ese ID, intentar cargarlo o crearlo nuevo.
- √ Merge copia el valor de un objeto separado en un objeto persistente con el mismo ID.

Métodos de Sesión

- Session.refresh(objeto)
  - √ Se utiliza para recargar el estado del objeto persistente que ha sido modificado en la base de datos por algún trigger o similar.
  - √ Si el objeto no existe en la Base de Datos, se producirá una Excepción. (org.hibernate.UnresolvableObjectException)

```
Session session = sessionFactory.openSession();
Profesor profesor = new Profesor(1001,"LLUIS", "GOMIS", "MARTINEZ");
 session.refresh(profesor);
session.close();
```

- Session.flush()
  - √ Los objetos no persisten en la Base de Datos hasta que se cierra la sesión
  - √ Con el método flush(), forzamos la persistencia de los objetos antes de cerrar la sesión

```
Session session = sessionFactory.openSession();
:
 session.flush();
:
session.close();
```


Excepciones

- Cuando Hibernate interactuar con la base de datos puede generar SQLExceptions.
- La gestión de excepciones en Hibernate se hace igual que en Java básico, mediante el bloque try y catch para su control.

```
try {
                student.setName("Johny");
                student.setCourse("Hibernate");
                 session.merge(student);
                 transaction.commit();
                System.out.println("Update Successfully");
                session.close();
  } catch (HibernateException e) {
                e.printStackTrace();
}
```

- Las excepciones mas usuales son:
  - √ LazyInitializationException
  - √ NonUniqueObjectException

Excepciones

LazyInitializationException

- Esta excepción se produce cuando se intenta cargar un objeto en memoria pero la sesión ya se ha cerrado.

- Se suele producir cuando trabajamos con Lazy Loading (o carga perezosa)

```
Session session = sessionFactory.openSession();
Profesor profesor = (Profesor) session.get(Profesor.class, 1065);
System.out.println("Leido el profesor");
System.out.println("Calle="+profesor.getDireccion().getMunicipio().getNombre());
session.close();
```

Todo OK

```
Session session = sessionFactory.openSession();
Profesor profesor = (Profesor) session.get(Profesor.class, 1065);
System.out.println("Leido el profesor");
session.close();
System.out.println("Calle="+profesor.getDireccion().getMunicipio().getNombre());
```

org.hibernate.LazyInitializationException

292

Excepciones

NonUniqueObjectException

- Esta excepción se produce cuando se intenta que haya más de un objeto en estado Persistido para la misma fila de la base de datos.

```
Session session = sessionFactory.openSession();
 session.beginTransaction();

 Profesor profesor = new Profesor(1001, "LLUIS", "GOMIS", "MARTINEZ");
 Profesor profesor2 = (Profesor) session.get(Profesor.class, 1001);

 profesor.setNombre("JUAN");
 session.update(profesor);

 session.getTransaction().commit();
session.close();
```

org.hibernate.NonUniqueObjectException

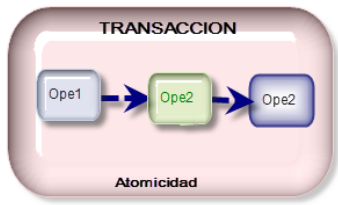
- Problema?
  - √ Hemos creado un objeto Transient con un códgo (1001) que ya existe en la BBDD
  - √ Hemos recuperado el objeto Persistente con código (1001)
  - √ Hemos modificado el objeto Transient .
  - √ Hemos intentado hacer Persistente el objeto Transient → ya existe en el contexto de Persistencia un objeto con ese código (1001) → ERROR

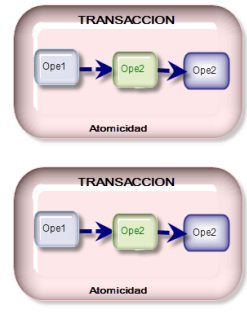
293

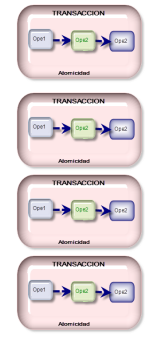
TRANSACCIONES Y CONCURRENCIA

Introducción

- Entendemos por Transacción: un grupo de operaciones que se ejecutan de forma atómica.
- Las transacciones permiten a varios usuarios trabajar simultáneamente con los mismos datos, sin comprometer la integridad y exactitud de los datos.







- Las transacciones son independientes entre si

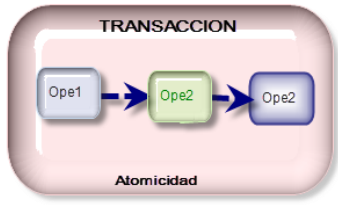
Introducción

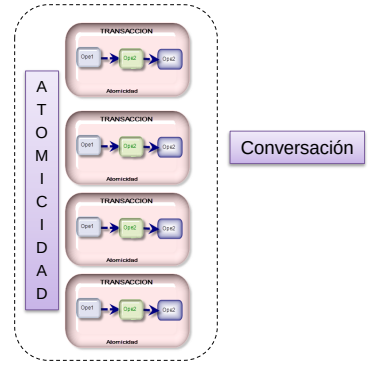
- Las transacciones, generalmente, disponen de varios atributos:
  - (**A**) Atomicidad
  - (**C**) Consistencia
  - (**I**) Aislamiento
  - (**D**) Durabilidad

  - Atomicidad:
    - √ En un conjunto de operaciones si una sola de ellas falla → Todas fallan.
  - Consistencia:
    - √ Cualquier transacción llevará a la base de datos desde un estado válido a otro también válido.
  - Aislamiento:
    - √ Es la propiedad que asegura que una operación no puede afectar a otras
  - Durabilidad:
    - √ Asegura que una vez realizada la operación, ésta persistirá y no se podrá deshacer aunque falle el sistema.

Introducción

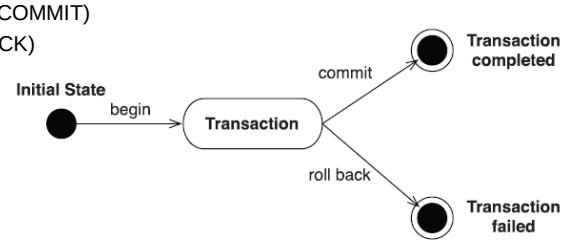
- Podemos encontrar transacciones CORTAS
  - Una sola transacción generalmente implica un solo lote de operaciones de bases de datos.
- En la actualidad, hablamos de CONVERSACIONES
  - Un grupo atómico de las operaciones de la base de datos se producen en no uno, sino varios lotes.





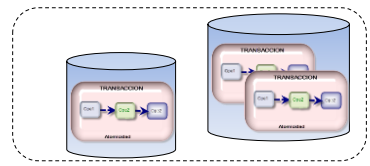
Transacciones y BBDD

- Bases de datos implementan el concepto de una unidad de trabajo como una transacción de base de datos.
- Todas las sentencias SQL se ejecutan dentro de una transacción, no hay manera de enviar una instrucción SQL fuera de una transacción de base de datos .
- Una transacción de BBDD termina de una de las 2 siguientes maneras:
  - Confirmada/Validada (COMMIT)
  - No Validada (ROLLBACK)



Transacciones y BBDD

- Dependiendo del SGBDR, debemos de Marcar el comienzo y Fin de una transacción.
  - BEGIN TRANSACTION / END TRANSACTION
  - COMMIT ------ operaciones ------ COMMIT
  - Usando JDBC –
    - √ setAutoCommit (false) – Operaciones JDBC – COMMIT/ROLLBACK
- Existen SGBDR donde las transacciones son órdenes SQL independientes (Auto-Commit).
- Cuando tenemos programas que trabajan con Unidades de Trabajo en varias BBDD's necesitamos un GESTOR de TRANSACCIONES para asegurar la atomicidad.

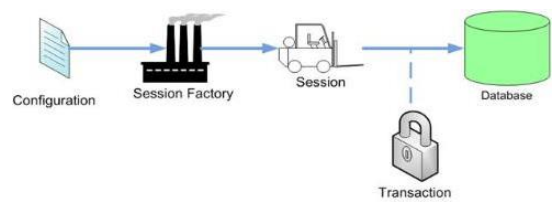


Transacciones e Hibernate

- En el mundo de Java, existen dos mecanismos bien definidos para tratar las transacciones en JDBC: JDBC en sí y JTA. Hibernate admite ambos mecanismos para integrarse con transacciones y permitir que las aplicaciones gestionen transacciones físicas.
- Hibernate se puede configurar asignando valor a la propiedad hibernate.transaction.coordinator_class:
  - jdbc (el valor predeterminado para aplicaciones no JPA)
    - √ Gestiona las transacciones a través de llamadas a java.sql.Connection, cubre solo las transacciones sobre base de datos.
  - jta
    - √ Gestiona las transacciones a través de JTA, cubre solo las transacciones sobre múltiples recursos, incluida la base de datos.
- Si una aplicación JPA no proporciona una configuración, Hibernate creará automáticamente el coordinador de transacciones adecuado en función del tipo de transacción para la unidad de persistencia.

Transacciones e Hibernate

- Hibernate trabaja directamente con conexiones JDBC sin agregar ningún comportamiento de bloqueo adicional.
- Hibernate no bloquea objetos, por lo que las transacciones producidas siguen el "aislamiento" definido en las bases de datos.
- Gracias al objeto Session, Hibernate proporciona la posibilidad de trabajar, eficientemente, con lecturas repetidas (cacheadas).
- Para poder tener transacciones en Hibernate necesitamos:
  - Objeto SessionFactory
  - Objeto Session



Transacciones e Hibernate

- Objeto SessionFactory
  - Una *SessionFactory* es un objeto seguro, con posibilidad de ser utilizado entre diferentes hilos.
  - Es MUY COSTOSO, por lo que es conveniente hacerlo las mínimas veces posible.
  - Se crea una sola vez, usualmente en el inicio de la aplicación, a partir de una instancia *Configuration*.

- Objeto Session
  - Una *Session* es un objeto de BAJO COSTO.
  - Su utilización entre hilos no es segura y debe de ser usado de una sola vez (una sola petición, una sola conversación o una sola unidad de trabajo)
  - Las *Sessions* pueden ser creadas y no consumirán recursos hasta que se utilice.

Transacciones e Hibernate

- Dependiendo del tipo de transacción tendremos:
  - Transacciones CORTAS:
    - √ Una transacción corta reduce la contención de bloqueos en la base de datos.
  - Transacciones LARGAS
    - √ Las transacciones largas son sinónimos de una alta carga de concurrencia.
    - √ No es recomendable mantener una transacción, de este tipo, ABIERTA durante mucho tiempo.
- Hay varios patrones de diseño para definir el funcionamiento correcto de las transacciones en Hibernate:
  - *sesión-por-operación (anti-patrón), sesión-por-aplicación (anti-patrón)*
    - √ *No utilizar, son pobres en Rendimiento.*
  - *sesión-por-petición*
    - √ *Es el patrón mas indicado en Hibernate*

Transacciones e Hibernate

- El patrón *sesión-por-petición*.
  - Es el patrón mas usado en Aplicaciones Multiusuario cliente/Servidor.

1. Una petición del cliente se envía al servidor (con persistencia de Hibernate).
2. Se abre una nueva Session de Hibernate y todas las operaciones de la base de datos se ejecutan en esta unidad de trabajo.
3. Usaremos una sola transacción para servir la petición del cliente.
4. Una vez completado el trabajo y preparada la respuesta, se cierra la sesión.
5. Devolución de la respuesta al Cliente.

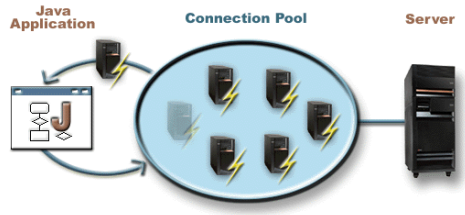
Características de las Transacciones

- En Hibernate el objeto Session utiliza un mecanismo de ***Lazy Loading***.
  - Hibernate no consume ningún recuro a menos que sea absolutamente necesario.
- Si estamos trabajando con Pool de conexiones, la conexión JDBC sólo se obtiene cuando comienza la transacción.
- La llamada a ***beginTransaction ()*** hace que la conexión JDBC obtenida recientemente sea de tipo ***setAutoCommit (falso)*** .
- A partir de este momento todas las sentencia SQL que se envíen, se hacen en la misma transacción y conexión.

Características de las Transacciones

- Después de confirmar la transacción (Commit) o deshacerla (Rollback) , la conexión de base de datos es liberada y desvinculada de la Sesión.

- Nueva Transacción → Nueva Conexión del Pool.



- Cuando la *Session* es liberada/cerrada, todas las instancias persistentes son Detached (Independientes)

Transacciones y Excepciones

- Todas las excepciones producidas por Hibernate son CRITICAS.

- Ante una excepción (incluyendo SQLException) debemos:
  1. Deshacer inmediatamente la transacción de la base de datos (*Rollback*)
  2. Descartar la instancia actual de Session. (*Session.close()*)
     - √ Ciertos métodos *Session no* dejarán la sesión en un estado consistente.
  3. Reiniciar otra nueva Session para seguir con el código.

- Hibernate envuelve las excepciones *SQLExceptions* en una subclase de *JDBCException*

Transacciones y TimeOut

- Uno de los elementos básicos en las transacciones es el tiempo de espera de la transacción.
- Definir este tiempo de espera asegura que ninguna transacción pueda comportarse de forma "inapropiada"
  - Que tarde mucho tiempo en ejecutarse.
  - Que la transacción bloquee muchos recursos durante mucho tiempo.
- Hibernate define este TimeOut en el objeto *Transaction*.

```
Session sess = factory.openSession();
try {
    sess.getTransaction().setTimeout(3);  // timeout 3 seg
    sess.getTransaction().begin();
    .....
    sess.getTransaction().commit()
} catch (RuntimeException e) {
    sess.getTransaction().rollback();
}
finally {    sess.close();   }
```


310

Conexión

Abrir conexión	Procesar datos	Cerrar conexión
• Socket • Interrogar servidor • Autenticación • Reserva e inicialización de recursos • Muy costoso	• Consultas • Transacciones • Ligero	• Liberar recursos • Cerrar comunicación • Costoso

311

Conexiones

- Problemas de las conexiones:
  - Número de usuarios concurrentes
  - Cierre de la conexión
- Escenarios:
  - Aplicaciones de escritorio:
    - √ Número limitado de usuarios
    - √ Duración determinada de la conexión
    - √ Distribuido
  - Aplicaciones web:
    - √ Número potencialmente ilimitado de usuarios
      - – Si cada usuario tiene una conexión dedicada se podría saturar el servidor de base de datos
    - √ Duración indeterminada de la conexión
      - – Es complejo saber cuando el usuario abandona el sitio (que no la página) para cerrar la conexión
    - √ Centralizado

Pool de Conexiones (General)

- En general, no es aconsejable crear una conexión cada vez que quiera interactuar con la base de datos.
- En su lugar, las aplicaciones Java deben utilizar un *pool de conexiones.*
- Funcionamiento:
  1. Cada sub-proceso de aplicación que tiene que trabajar con la BBDD solicita una conexión.
  2. Esta conexión es concedida desde el POOL de Conexiones.
  3. Cuando el sub-proceso acaba (se han ejecutado todas las operaciones SQL), la conexión es devuelta al POOL.

Pool de Conexiones (General)

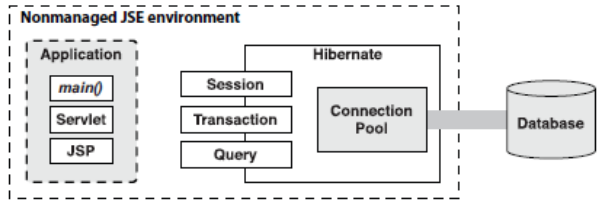
- Hay varias razones para utilizar Pool de Conexiones:
  1. El Pool de Conexiones mantiene las conexiones y reduce al mínimo el coste de apertura y cierre de las conexiones.
  2. Mediante Pool, podemos optimizar el uso de las conexiones, activando/desconectando conexiones si hay/no hay peticiones.
  3. La creación de Sentencias Preparadas es caro para algunos DRIVERS, mediante el POOL podemos cachear las sentencias entre peticiones.

Estrategias de configuración

- Esquema centralizado:
  - Pool de conexiones compartido entre las aplicaciones web alojadas en el servidor JEE.
  - Un único pool con una única configuración
  - Un único tipo de pool de conexiones
  - Configurado a nivel de servidor (requiere permisos)
  - Usuario de base de datos único
  - Se publica como un recurso de tipo datasource con JNDI.
- Esquema distribuido:
  - Pool de conexiones independientes administrados por cada aplicación web alojadas en el servidor JEE.
  - Múltiples pool con múltiples configuraciones
  - Múltiples tipos de pool de conexiones, uno por aplicación
  - Configurado a nivel de aplicaciones (no requiere permisos)
  - Usuario de base de datos por aplicación

Pool de Conexiones con Hibernate

- Cuando trabajamos con Hibernate, éste actúa como un cliente del POOL de conexiones.



- Hibernate define una arquitectura plug-in que permite la integración con "cualquier" software de Pool de Conexiones (pero comercialmente no es recomendable)
- Existe un Software libre llamado C3P0, que se integra totalmente con Hibernate y ampliamente usado (c3p0.jar)

Pool de Conexiones con Hibernate

- La configuración del POOL de conexiones se realiza dentro del fichero de configuración de Hibernate (hibernate.cfg.xml).

```
hibernate.connection.driver_class = org.hsqldb.jdbcDriver
hibernate.connection.url = jdbc:hsqldb:hsql://localhost
hibernate.connection.username = sa
hibernate.dialect = org.hibernate.dialect.HSQLDialect

hibernate.c3p0.min_size = 5                  // Número mínimo de CX que mantiene c3p0

hibernate.c3p0.max_size = 20                 // Número máximo de CX permitidas

hibernate.c3p0.timeout = 300                 // Segundos de TIMEOUT para eliminar una CX

hibernate.c3p0.max_statements = 50           // Nº máximo de comandos guardados en Caché
                                             para la utilización en Hibernate.

hibernate.c3p0.idle_test_period = 3000       // Tiempo máximo para la validación de un CX

hibernate.show_sql = true
hibernate.format_sql = true
```

Pool de Conexiones con Hibernate

- Una vez definido nuestro Pool de conexiones podríamos activar diferentes opciones:
  - Seguimiento de las Operaciones
    - √ hibernate.show_sql
    - √ hibernate.format_sql
    - √ hibernate.use_sql_comments

```
<property name="hibernate.show_sql">true</property>
<property name="hibernate.format_sql">true</property>
<property name="hibernate.use_sql_comments">true</property>
```

  - Estadísticas
    - √ Apache Log4j *(log4j.jar)*

persistence.xml

```
<properties>
        <property name="javax.persistence.jdbc.url“ value="jdbc:oracle:thin:@localhost:1521:xe" />
        <property name="javax.persistence.jdbc.user" value="hr" />
        <property name="javax.persistence.jdbc.password" value="hr" />
        <property name="javax.persistence.jdbc.driver"  value="oracle.jdbc.OracleDriver" />
        <property name="hibernate.show_sql" value="true" />
        <property name="hibernate.format_sql" value="true" />
        <property name="hibernate.dialect" value="org.hibernate.dialect.Oracle10gDialect" />
        <property name="hibernate.c3p0.min_size" value="5" />
        <property name="hibernate.c3p0.max_size" value="20" />
        <property name="hibernate.c3p0.timeout" value="500" />
        <property name="hibernate.c3p0.max_statements" value="50" />
        <property name="hibernate.c3p0.idle_test_period" value="300"/>
        <property name="hibernate.c3p0.preferredTestQuery" value="SELECT 1 from dual;" />
        <property name="hibernate.c3p0.testConnectionOnCheckout" value="true" />
        <property name="hibernate.c3p0.maxIdleTime" value="10800" />
        <property name="hibernate.c3p0.maxIdleTimeExcessConnections" value="600"/>
</properties>
```

Configurar el pool de conexiones en Tomcat

- El pool se configura añadiendo un fichero llamado context.xml en la carpeta META-INF.

```
<Resource name="jdbc/TestDB“ auth="Container"
        type="javax.sql.DataSource"
        factory="org.apache.tomcat.jdbc.pool.DataSourceFactory"
        testWhileIdle="true“ testOnBorrow="true“ testOnReturn="false"
        validationQuery="SELECT 1“ validationInterval="30000" timeBetweenEvictionRunsMillis="30000"
        maxActive="100“ minIdle="10“ maxWait="10000“ initialSize="10"
        removeAbandonedTimeout="60“ removeAbandoned="true"
        logAbandoned="true"
        minEvictableIdleTimeMillis="30000"
        jmxEnabled="true"
        jdbcInterceptors="org.apache.tomcat.jdbc.pool.interceptor.ConnectionState;
          org.apache.tomcat.jdbc.pool.interceptor.StatementFinalizer"
        username="root“ password="password"
        driverClassName="com.mysql.jdbc.Driver"
        url="jdbc:mysql://localhost:3306/mysql"/>
```

- Sustituir en la sección <properties> en persistence.xml por:

```
<jta-data-source>java:/com/env/jdbc/TestDB</jta-data-source>
```

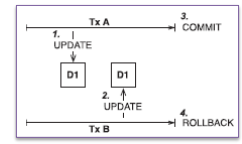

Concurrencia

Introducción

- Los Sistemas Transaccionales y BBDD tratan de asegurar el aislamiento de transacciones.
- Deben asegurar que durante una transacción no haya otra transacción implicada en los mismos elementos.
- Tradicionalmente, la concurrencia ha sido implementada con:
  - Bloqueos
  - Regiones Críticas
  - Semáforos.
- Las aplicaciones heredan el aislamiento que proporciona el sistema de gestión de base de datos.

Introducción

- Si no controlamos estos accesos, podremos encontrarnos con:
  - Actualizaciones perdidas (**Lost Update**)
    - √ Si dos transacciones actualizan una fila y luego la segunda transacción se anula, causando cambios tanto que se pierdan.

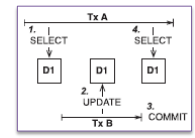


  - Lecturas Sucias (**Dirty Reads**)
    - √ Ocurre cuando una transacción lee datos modificados por otros procesos concurrentes que aún no han realizado commit.

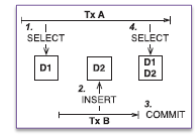
Introducción

- Sino controlamos estos accesos, podremos encontrarnos con:

  - Análisis Contradictorios (**Nonrepeatable Reads**)
    - √ Se produce si una transacción lee una fila dos veces y lee diferente datos cada vez.



  - Lecturas Fantasmas (**Phantom Reads**)
    - √ Se producen cuando una transacción se ejecuta una consulta en dos ocasiones, y el segundo conjunto de resultados incluye filas que no eran visibles en el primer conjunto de resultados



Niveles de Aislamiento

- El estándar ANSI SQL define una serie de niveles de aislamiento de transacción que coinciden con los mismos que JTA.

- Nivel de Aislamiento:
  - Define el grado en que se aísla una transacción de las modificaciones de datos realizadas por otras transacciones.
  - Este nivel de aislamiento afecta a los tipos y duración de los Bloqueos.
  - El nivel de Aislamiento cobra mayor importancia cuando mayor sea la concurrencia.

Niveles de Aislamiento

Niveles de Aislamiento:

- READ UNCOMMITED
  - √ Una transacción NO puede escribir en una fila si otra transacción no confirmada ya ha escrito a la misma.
  - √ Cualquier transacción puede leer cualquier fila (confirmados o no)
  - √ Es la opción menos restrictiva, recomendable si se manejan datos de solo lectura o muy pocas actualizaciones.
- READ COMMITED
  - √ Evita Lecturas Sucias, pero permite que se produzcan Análisis Contradictorios y Lecturas Fantasmas
- REPETEABLE READ
  - √ Sólo permite Lecturas Fantasmas.
- SERIALIZABLE
  - √ Garantiza que una transacción recuperará exactamente los mismos datos cada vez que se repita una Select .
  - √ Se Aplica un nivel de bloqueo que puede afectar a los demás usuarios.
  - √ Es la opción mas restrictiva.

Niveles de Aislamiento. Definición

- Si no indicamos nada, cada conexión JDBC tiene el nivel de aislamiento definido por el SGBD.
- En Hibernate podemos elegir el nivel de aislamiento de las transacciones mediante la opción:

hibernate.cfg.xml

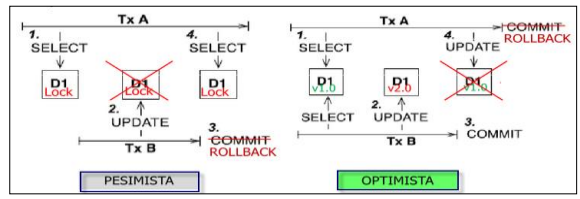
*<property name=**"hibernate.connection.isolation"**>8</property>*

1 - Read uncommitted
2 - Read committed
4 - Repeatable read
8 - Serializable

- Hibernate configura el nivel de aislamiento en cada conexión JDBC obtenidas del Pool de conexiones antes de iniciar una transacción.
- Establecer el nivel de aislamiento es una opción global que afecta a todas las conexiones y transacciones.

Control de Concurrencia: Bloqueos

- Hay 2 técnicas de concurrencia:
  - Bloqueo Pesimista
  - Bloqueo Optimista
- Es importante determinar qué técnica es mejor para conversaciones largas dentro de nuestras aplicaciones.

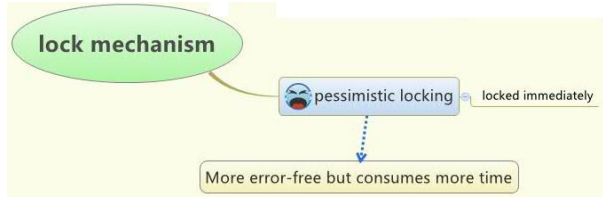


- Hibernate y Java Persistence utilizan el control de concurrencia optimista, porque:
  - √ Maximiza la eficiencia (menor aislamiento, menos bloqueos en BD)
  - √ Es adecuada para la mayor parte de las transacciones.

328

Control de Concurrencia: Bloqueos

- Bloqueo Pesimista
  - Los datos son bloqueados previos a su modificación para evitar que nadie los modifique.
  - Una vez que los datos a actualizar han sido bloqueados la aplicación puede acometer los cambios, con *commit* o *rollback*, en ese caso el bloqueo es automáticamente eliminado.
  - Si alguien intenta adquirir un bloqueo de los mismos datos durante el proceso será obligado a esperar hasta que la primera transacción finalice.
  - Menos errores pero consume mas tiempo en su implementación.



329

Control de Concurrencia: Bloqueos

- Bloqueo Pesimista
  - Los bloqueos pesimistas con obtenidos mediante el método *session.lock()*

    sess.lock (Elemento , LockMode.XXXXX);

  - Ninguna transacción simultánea puede obtener un bloqueo en los mismos datos.

Tipo de Bloqueo	Obtención	Descripción
NONE	Bajo Petición	- Representa la ausencia de un bloqueo. - Todos los objetos se pasan a este modo de bloqueo al final de una Transaction
UPGRADE	Bajo Petición	- Obtiene un bloqueo pesimista de UPDATE a nivel de BBDD si es posible. - Equivalente a un SELECT ….. FOR UPDATE
UPGRADE_NOWAIT	Bajo Petición	- Equivalente a un SELECT ….. FOR UPDATE WAIT si se puede realizar a nivel de BBDD
WRITE	Automático	- Se adquiere automáticamente cuando Hibernate actualiza o inserta una fila.

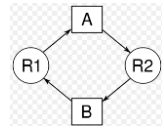
Control de Concurrencia: Bloqueos

- Bloqueo Pesimista

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

Item i = (Item) session.get(Item.class, 123);

session.lock(i, LockMode.UPGRADE);

String description = (String)

session.createQuery("select i.description from Item i" +
        " where i.id = :itemid").setParameter("itemid", i.getId() ).uniqueResult();

tx.commit();

session.close();
```

Control de Concurrencia: Bloqueos

- Bloqueo Pesimista
  - Problemas fundamentales:
    - √ Bloqueos:
      - – ¿Qué ocurre si un usuario adquiere un bloqueo y no finaliza transacción?
    - √ Deadlock
      - – Bloqueos mutuos.



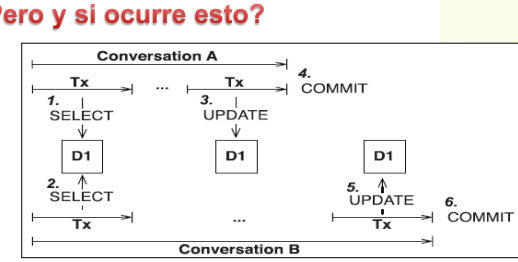
    - √ En general los Sistemas RDBMS ofrecen cláusulas para este bloqueo y como tratarlo
      - – SELECT FOR UPDATE
      - – SELECT FOR UPDATE WAIT
      - – DEADLOCK con tiempo definido y ruptura

332

Control de Concurrencia: Bloqueos

- Bloqueo Optimista
  - No bloquea los registros que se van a actualizar y asume que los datos que están siendo actualizados no van a cambiar desde que se han leído.
  - Este tipo de mecanismo, si falla, proporciona una EXCEPTION sólo al final de la unidad de trabajo, cuando se van a guardar los datos.
  - Un bloqueo optimista siempre asume que todo va a estar bien y que las modificaciones de datos en conflicto son raros.

¿Pero y si ocurre esto?



333

Control de Concurrencia: Bloqueos

- Bloqueo Optimista
  - Para asegura que los datos que están siendo escritos son consistentes *(es decir que ninguna otra transacción ha actualizado los datos después de la lectura),* con los leídos en primera instancia se utiliza un VERSIONADO
  - VERSIONADO:
    - √ Se procede al leer un valor junto al registro, se actualizará ese valor a la BD cuando el registro es actualizado.
    - √ El chequeo de versión utiliza números de versión, o sellos de fecha (*timestamps*) para detectar actualizaciones en conflicto y para prevenir la pérdida de actualizaciones
  - Hibernate chequea el versionado de forma automática en el momento de vaciado, lanzando una excepción si se detecta una modificación concurrente.
  - Debemos capturar y manejar esta excepción (*StaleObjectStateException*)

Control de Concurrencia: Bloqueos

- Bloqueo Optimista
  - Para especificar una versión a un objeto persistente, debemos añadir un nuevo atributo que haga de versionado (*long o timestamp*) y definirlo como *version* dentro del archivo de mapeo de la clase.

```
public class Product {
private long id;
private String name;
private long version;
:
:
```

Hibernate.cfg.xml

```
<hibernate-mapping package="com.intertech.domain">
<class name="Product">
<id name="id">
<generator class="native" />
</id>
<version name="version" type="long" column=ldver/>
<property name="name" not-null="true" />
<property name="model" not-null="true" />
<property name="cost" type="double"/>
</class>
</hibernate-mapping>
```

  - Para hacerlo con anotaciones, deberemos marcar el método ***getVersion*** como **@Version**

```
@Version
public long getVersion() {
return version;
}
```

Hibernate 4.X

```
@ Version
@ Column (name = "OBJ_VERSION")
versión int privado;
```

Control de Concurrencia: Bloqueos

- Bloqueo Optimista. INCONVENIENTES
  - Si deseamos que el versionado sea automática, deberemos modificar la Tabla correspondiente para añadirle una nueva columna para la versión.

¿Y si no podemos modificar nuestra Tabla para añadir el Versionado?

Hibernate propone la siguiente solución

- Utilizar todos los campos del objeto persistente para comprobar que nada en una fila se ha modificado antes de actualizar una fila.
- Sólo es válido dentro del mismo Contexto de Persistencia (*Session*)
- Deberemos añadir lo siguiente en el fichero ***Tabla.hbm.xml*** correspondiente

  <class name="CLASE" **dynamic-update="true" optimistic-lock="all"**>

CACHE

Tipos de caché en Hibernate

- El uso de técnicas de caché puede reducir el número de consultas que se lanzan contra la BD
- Caché de primer nivel
  - Dentro de una transacción, Hibernate cachea los objetos que se recuperan de BD en la sesión (objeto Session)
    - √ Nunca se recupera más de una vez un mismo objeto
    - √ Sólo se cachean mientras dura la transacción
  - Este tipo de caché es implícita al modelo de Hibernate
- Caché de segundo nivel
  - Permite cachear el estado de objetos persistentes a nivel de proceso
  - Este tipo de caché es opcional y requiere configuración
- Caché de resultados de búsqueda
  - Permite cachear resultados de una búsqueda particular
  - Este tipo de caché está integrada con la caché de segundo nivel

Cache de Primer Nivel

- Es el que mantiene automáticamente Hibernate cuando dentro de una transacción interactuamos con la base de datos, en éste caso se mantienen en memoria los objetos que fueron cargados y si mas adelante en el flujo del proceso volvemos a necesitarlos van a ser retornados desde el cache, ahorrando accesos sobre la base de datos.
- Lo podemos considerar como un cache de corta duración ya que es válido solamente entre el begin y el commit de una transacción, en forma aislada a las demás.
- Hibernate lo maneja por defecto, no hay que configurar nada, están vinculadas a las Session o EntityManager.
- Para deshabilitar o evitar el uso del cache, hay que utilizar una StatelessSession, no crea la cache de Primer Nivel, es casi como si se utilizara JDBC directamente.

Caché de Segundo Nivel

- Una sesión (Session) de Hibernate, es un caché de datos persistentes a nivel de la transacción.
- Es posible configurar un caché a nivel de cluster o a nivel de la JVM (a nivel de la SessionFactory), denominada caché de Segundo Nivel, sobre una base de clase-por-clase o colección-por-colección.
- Hay que tener en cuenta que los cachés nunca están al tanto de los cambios que otra aplicación haya realizado al almacén persistente (se pueden configurar para que los datos en caché expiren regularmente).
- Existe la opción de decirle a Hibernate qué implementación de cacheo usar, especificando el nombre de una clase que implemente org.hibernate.cache.CacheProvider, usando la propiedad hibernate.cache.provider_class.
- Hibernate trae incorporada una buena cantidad de integraciones con proveedores de cachés open-source. Además, se puede implementar un caché propio y enchufarlo.

Estrategias de almacenamiento en caché

- read-only:
  - Si su aplicación necesita leer pero nunca modificar las instancias de una clase persistente, se puede usar un caché read-only. Esta es la estrategia más simple y la de mejor rendimiento. También es perfectamente segura de utilizar en un cluster.
- read-write:
  - Si la aplicación necesita actualizar datos, puede ser apropiado usar una caché de lectura-escritura (read-write). Esta estrategia de cacheo nunca debería ser usada si se requiere aislamiento de transacciones serializables.
  - Si el caché se usa en un entorno JTA, se debe especificar la propiedad hibernate.transaction.manager_lookup_class, especificando una estrategia para obtener la propiedad TransactionManager de JTA.
  - En otros entornos, hay que asegurarse de que la transacción esté completa para cuando ocurran Session.close() o Session.disconnect().
  - Si se quiere usar esta estrategia en un cluster, hay que asegurarse de que la implementación subyacente de caché soporta "locking". Los proveedores de caché que vienen ya incorporados no lo soportan.

Estrategias de almacenamiento en caché

- nonstrict-read-write:
  - Si la aplicación necesita actualizar datos, pero sólo ocasionalmente (es decir, si es improbable que dos transacciones traten de actualizar el mismo ítem simultáneamente), y no se requiere un aislamiento de transacciones estricto, puede ser apropiado un caché "de lecto-escritura no estricta" (nonstrict-read-write).
  - Si el caché se usa en un entorno JTA, se debe especificar hibernate.transaction.manager_lookup_class.
  - En otros entornos, hay que asegurarse de que la transacción esté completa para cuando ocurran Session.close() o Session.disconnect().
- transactional:
  - La estrategia transaccional (transactional) provee soporte para proveedores de caché enteramente transaccionales, como JBoss TreeCache. Tales cachés sólo pueden ser usados en un entorno JTA, y se debe especificar hibernate.transaction.manager_lookup_class.

Administrar los cachés

- Acciones que agregan un elemento al caché interno de la sesión.
  - Guardar o actualizar un artículo: save(), update(), saveOrUpdate())
  - Recuperando un artículo: load(), get(), list(), iterate(), scroll()
- El estado de un objeto se sincroniza con la base de datos cuando llama al método flush(). Para evitar esta sincronización, puede eliminar el objeto y todas las colecciones de la caché de primer nivel con el método evict() o eliminar todos los elementos de la caché de sesión con el método clear().
- La sesión proporciona un método contains() para determinar si una instancia pertenece a la caché de la sesión.
- Para el caché de segundo nivel, hay métodos definidos en Factory para expulsar el estado en caché de una instancia, todas las instancias de una clase, una colección de una instancia o todas las instancias de una clase.

CacheMode

El CacheMode controla la manera en que interactúa una sesión en particular con el caché de segundo nivel:

- CacheMode.NORMAL:
  - √ JPA: CacheStoreMode.USE, CacheRetrieveMode.USE
  - √ lee ítems desde y escribe ítems hacia el caché del segundo nivel
- CacheMode.GET:
  - √ JPA: CacheStoreMode.BYPASS, CacheRetrieveMode.USE
  - √ lee ítems del caché del segundo nivel. No escribe al caché de segundo nivel excepto cuando actualiza datos
- CacheMode.PUT:
  - √ JPA: CacheStoreMode.USE, CacheRetrieveMode.BYPASS
  - √ escribe ítems al caché de segundo nivel. No lee del caché de segundo nivel
- CacheMode.REFRESH:
  - √ JPA: CacheStoreMode.REFRESH, CacheRetrieveMode.BYPASS:
  - √ escribe ítems al caché de segundo nivel. No lee del caché de segundo nivel, saltándose el efecto de hibernate.cache.use_minimal_puts, forzando la actualización del caché de segundo nivel para todos los ítems leídos de la base de datos
- CacheMode.IGNORE
  - √ JPA: CacheStoreMode.BYPASS, CacheRetrieveMode.BYPASS
  - √ No lee ítems desde y ni escribe ítems hacia el caché del segundo nivel

Regiones

- Un principio básico es dividir la caché en diversas "regiones" donde se ubican los objetos. Estas regiones son configurables de forma independiente.

- Hibernate necesita regiones de caché para 4 usos distintos:
  - datos de entidades: representación de los datos de las entidades
  - datos de colecciones: referencia a las entidades de las colecciones
  - resultados de consultas:
    - √ StandardQueryCache: representación de los resultados de consultas,
    - √ UpdateTimestampsCache: marcas timestamps para el control de versiones de los datos almacenados en StandardQueryCache para ser poder "invalidarlos" cada vez que se modifiquen los datos subyacentes. (*no debe caducar*)

Configuración de caché

- hibernate.cache.region.factory_class: Establece el proveedor a utilizar.
- hibernate.cache.use_second_level_cache: Habilitar el almacenamiento en caché de segundo nivel en general.
- hibernate.cache.use_query_cache: Habilitar el almacenamiento en caché de segundo nivel de los resultados de la consulta. El valor predeterminado es falso.
- hibernate.cache.query_cache_factory: Implementación encargada de la invalidación de los resultados de consultas.
- hibernate.cache.use_minimal_puts: Optimiza las operaciones de caché de segundo nivel para minimizar las escrituras, a costa de lecturas más frecuentes.
- hibernate.cache.region_prefix: Nombre que se utilizará como prefijo para todos los nombres de región de caché de segundo nivel.
- hibernate.cache.default_cache_concurrency_strategy: read-only, read-write, nonstrict-read-write, transactional.

Configuración de caché

- javax.persistence.sharedCache.mode: Modo de cacheo de las entidades. Los siguientes valores son posibles:
  - ENABLE_SELECTIVE (Valor por defecto y recomendado):
    - √ Las entidades no se almacenan en caché a menos que se marquen explícitamente como almacenables (con la anotación @Cacheable).
  - DISABLE_SELECTIVE:
    - √ Las entidades se almacenan en caché a menos que se marque explícitamente como no almacenable en caché.
  - ALL:
    - √ Las entidades siempre se almacenan en caché incluso si están marcadas como no almacenables en caché.
  - NONE:
    - √ Ninguna entidad se almacena en caché, incluso si se marca como almacenable en caché. Esta opción básicamente desactiva el almacenamiento en caché de segundo nivel.

Cache de entidades

- Para indicar que una entidad se debe cachear se utiliza la anotación @Cacheable (@Cacheable(false) no se debe cachear) y se configura con la anotación @Cache:
  - usage: estrategia de concurrencia de caché dada, que puede ser: NONE, READ_ONLY, NONSTRICT_READ_WRITE, READ_WRITE, TRANSACTIONAL.
  - region: La región de caché. Este atributo es opcional y toma como valor predeterminado el nombre de clase completamente calificado de la clase, o el nombre de rol calificado de la colección.
  - include: all incluye todas las propiedades (predeterminado) o non-lazy solo incluye propiedades no perezosas.

```
@Entity
@Cacheable
@org.hibernate.annotations.Cache(usage=CacheConcurrencyStrategy.READ_ONLY)
public static class Profesor {
    @OneToMany(mappedBy = “profesor", cascade = CascadeType.ALL)
    @Cache(usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
    private List<Phone> phones = new ArrayList<>( );
```


Caché de Consultas

- Los conjuntos de resultados de peticiones también pueden ponerse en caché. Esto solamente es útil para consultas que se ejecutan frecuentemente con los mismos parámetros.
- El poner en caché los resultados de una petición introduce algunos sobrecostos en términos del procesamiento transaccional normal de sus aplicaciones.
- Para utilizar el caché de peticiones primero necesita habilitar el caché de peticiones:
  - hibernate.cache.use_query_cache true
- Esta configuración crea dos nuevas regiones de caché:
  - org.hibernate.cache.StandardQueryCache, mantiene los resultados de la petición en caché
  - org.hibernate.cache.UpdateTimestampsCache, mantiene los sellos de fecha de las actualizaciones más recientes a las tablas de peticiones. Estas se utilizan para validar los resultados ya que se sirven desde el caché de peticiones.

Caché de Consultas

- Para cachear el resultado de una consulta:

```
List<Person> persons = entityManager.createQuery(
        "from Person p " +
        "where p.name = :name", Person.class)
    .setParameter( "name", "John Doe")
    .setHint( "org.hibernate.cacheable", "true")
    .getResultList();
```

- Para cachearlo en una región especifica:

```
List<Person> persons = entityManager.createQuery(
        "from Person p " +
        "where p.id > :id", Person.class)
    .setParameter( "id", 0L)
    .setHint( QueryHints.HINT_CACHEABLE, "true")
    .setHint( QueryHints.HINT_CACHE_REGION, "query.cache.person" )
    .getResultList();
```


Hints de Cache

- QueryHints.HINT_CACHEABLE:
  - especificar si los resultados de la consulta se deben almacenar en caché.
- QueryHints.HINT_CACHE_MODE:
  - especificar el modo de caché (CacheMode) que está en efecto para la ejecución de la consulta insinuada: CacheMode.NORMAL, CacheMode.GET, CacheMode.PUT, CacheMode.REFRESH, CacheMode.IGNORE
- QueryHints.HINT_CACHE_REGION:
  - especificar el nombre de la región de caché (dentro de la región de caché de resultados de la consulta de Hibernate).
- "javax.persistence.cache.retrieveMode”
  - especificar si se usa u omite la cache cuando los datos y las consultas recuperan: CacheRetrieveMode.USE, CacheRetrieveMode.BYPASS
- "javax.persistence.cache.storeMode”
  - especificar cuando se confirman los datos de la cache:
    - √ BYPASS: No insertar en el caché.
    - √ REFRESH: Insertar/actualizar los datos de la entidad en la memoria caché cuando se lee de la base de datos y cuando se confirma en la base de datos.
    - √ USE: Insertar los datos de la entidad en la memoria caché cuando se lea de la base de datos e inserte/actualice los datos de la entidad cuando se confirma en la base de datos: este es el comportamiento predeterminado.

Configuración propia del proveedor

- name: Nombre de la caché. Ha de ser único.
- maxElementsInMemory: Número máximo de objetos que se crearán en memoria.
- eternal: Marca que los elementos nunca expirarán de la caché. Serán eternos y nunca se actualizarán. Ideal para datos inmutables.
- overflowToDisk: Marca que los objetos se guarden en disco si se supera el límite de memoria establecido.
- timeToIdleSeconds: Es el tiempo de inactividad permisible para los objetos de una clase. Si un objeto sobrepasa ese tiempo sin volver a activarse, se expulsará de la caché.
- timeToLiveSeconds: Marca el tiempo de vida de un objeto. En el momento en que se sobrepase ese tiempo, el objeto se eliminará de la caché, independientemente de cuando se haya usado.
- diskPersistent: Establece si los objetos se almacenarán a disco. Esto permite mantener el estado de la caché de segundo nivel cuando se apaga la JVM, es decir cuando se cae la máquina o se para el servidor o la aplicación.
- diskExpiryThreadIntervalSeconds: Es el número de segundos tras el que se ejecuta la tarea de comprobación de si los elementos almacenados en disco han expirado.

VALORACIÓN DE LA ACCIÓN FORMATIVA

Tu opinión es muy importante para nosotros, por eso solicitamos tu colaboración para analizar la calidad de esta acción formativa y para identificar elementos de mejora .

ACCEDE AL CUESTIONARIO

Todos aprendemos. Gracias!!

iconotc TRAINING CONSULTING

¡Seguimos en contacto!

www.iconotc.com