Hibernate

http://www.hibernate.org

INSTALACIÓN

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

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>
```

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

Introducción

- 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"

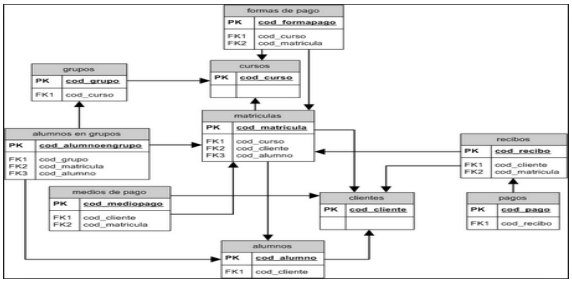
Introducción a Hibernate

- 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:



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 motores de base de datos SQL en general, no son compatibles con subtipos y herencia de tablas.

Discrepancia del Paradigma

- Problemas de identidad
  - Objetos Java definen dos nociones diferentes de identidad:
    - Identidad de objeto o referencia (equivalente a la posición de memoria, comprobar con un ==).
    - 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.
    - Relación de pertenencia → Elementos contenidos
    - Modelo de composición (colecciones) → Modelo jerárquico
  - Las asociaciones en BBDD están representados mediante la migración de claves.
    - Todas las asociaciones en una base de datos relacional son bidireccional

ORM

- Las siglas ORM significan “Object-Relational mapping” (Mapeo Objeto-Relacional): técnica de mapear datos desde una representación de modelo de objeto a una representación de modelo de datos relacional (y viceversa).
- Un ORM es un framework de persistencia de nuestros datos (objetos) a una base de datos relacional, es decir, código que escribimos para guardar y recuperar 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	OpenJPA	TopLink	Kodo
CocoBase	DataNucleus	EclipseLink	Amber

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 (SessionFactory) para interactuar con objetos transaccionales y realizar operaciones diversas (conexión, validación, optimización, etc)
  4. Un mecanismo para especificar los metadatos de mapeo (fichero XML, anotaciones)

Java Persistence API

- Java Persistence API, más conocida por sus siglas JPA, es la API de persistencia desarrollada para la plataforma Java EE.
- Es la propuesta estándar que ofrece Java para implementar un framework del lenguaje de programación Java que maneja datos relacionales en aplicaciones usando la Plataforma Java en sus ediciones Standard (Java SE) y Enterprise (Java EE).
- La JPA se origina a partir del trabajo del JSR 220 Expert Group el cual correspondía a EJB3. JPA 2.0 sería el trabajo del JSR 317 y posteriormente JPA 2.1 en el JSR 338.
- La persistencia en este contexto cubre tres áreas:
  - La API en sí misma, definida en el paquete javax.persistence
  - El lenguaje de consulta Java Persistence Query Language (JPQL).
  - Metadatos objeto/relacional.
- El objetivo que persigue el diseño de esta API es no perder las ventajas de la orientación a objetos al interactuar con una base de datos (siguiendo el patrón de mapeo objeto-relacional), como sí pasaba con EJB2, y permitir usar objetos regulares (conocidos como POJO).

Introducción a Hibernate

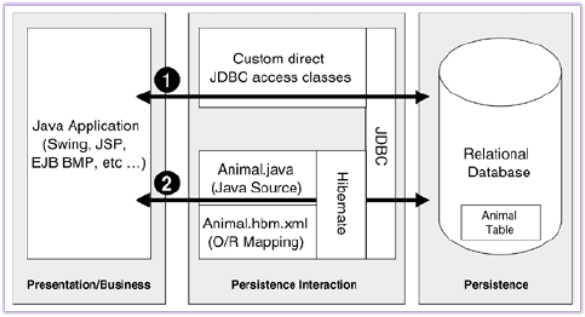
- La finalidad de Hibernate es:
  - Proporcionar un puente entre los datos relacionales (BBDD) y la Orientación a Objetos (Java) mediante su modelo de Mapeo Relacional/Objeto (ORM).
- Para poder desarrollar en Hibernate, deberemos tener conocimientos de:
  - Conocer la lógica de la aplicación, estructura de clases y objetos
  - Comprender los conceptos básicos de las transacciones y los patrones de diseño: Unit of Work (PoEAA) or Application Transaction
  - No se requiere tener una sólida formación en SQL (aunque ayuda) pero si los principios de modelado de datos 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 todas las operaciones de una aplicación Java enfrentada a su base de datos relacional.

Historia

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 Hibernate 2 - 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- …	EJB 3.2 en JSR-345	Hibernate 5.0

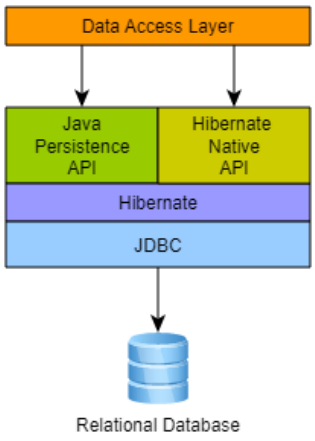
Introducción a Hibernate

- Objetivo de: "Aliviar al desarrollador del 95% de las tareas de programación relacionados con la persistencia de datos común" lo resuelve mediante la combinación de clases de Java y ficheros descriptores XML o anotaciones.
- 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.

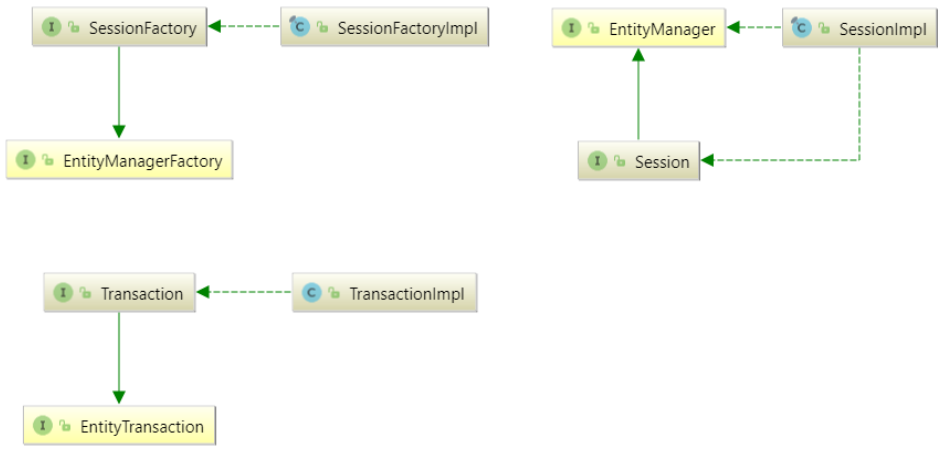


Arquitectura

- Hibernate, como solución ORM, "se encuentra" entre la capa de acceso a datos de la aplicación Java y la base de datos relacional, como se puede ver en el diagrama anterior.
- La aplicación Java hace uso de las API de Hibernate para consultar, recupera y almacenar sus datos de dominio.
- Como proveedor de JPA, Hibernate implementa las especificaciones de la API de persistencia de Java y la asociación entre las interfaces JPA con las implementaciones específicas de Hibernate.



Implementaciones



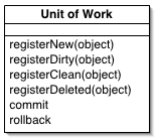
Infraestructura

- SessionFactory (org.hibernate.SessionFactory)
  - Una representación segura para subprocesos (e inmutable) de la asignación del modelo de dominio de la aplicación a una base de datos. Actúa como una fábrica de instancias org.hibernate.Session. El EntityManagerFactory es básicamente el equivalente JPA del SessionFactory.
  - Un SessionFactory es muy costoso de crear, por lo que, para cualquier base de datos, la aplicación debe tener solo un SessionFactory asociado. El SessionFactory mantiene servicios que Hibernate utiliza en todos los Session(s): memorias caché de segundo nivel, los conjuntos de conexiones, integraciones de sistema de transacciones, etc.
- Session (org.hibernate.Session)
  - Objeto de un solo subproceso y de corta duración que modela conceptualmente una "Unidad de trabajo" (PoEAA). En la nomenclatura de JPA, el Session está representado por un EntityManager.
  - Tras la escena, Hibernate Session envuelve un JDBC java.sql.Connection y actúa como una fábrica de instancias org.hibernate.Transaction. Mantiene un contexto de persistencia generalmente de "lectura repetible" (caché de primer nivel) del modelo de dominio de aplicación.
- Transaction (org.hibernate.Transaction)
  - Objeto de un solo subproceso y de corta duración utilizado por la aplicación para demarcar los límites de las transacciones físicas individuales. EntityTransaction es el equivalente de JPA y ambos actúan como una API de abstracción para aislar la aplicación del sistema de transacciones subyacente en uso (JDBC o JTA).

Patrón Unidad de trabajo (UoW)

https://martinfowler.com/eaaCatalog/unitOfWork.html

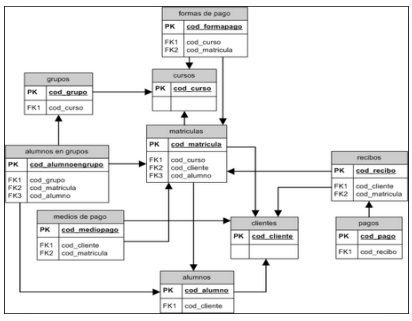
- El patrón UNIT OF WORK fue definido por Martin Fowler (Fowler, Patterns of Enterprise Application Architecture, 184).
- De acuerdo con Martin, "Un UNIT OF WORK mantiene una lista de objetos afectados por una transacción de negocio, coordina la actualización de cambios y la resolución de problemas de concurrencia".



- El patrón de UNIT OF WORK encaja perfectamente con las transacciones, pues podemos hacer coincidir un UoW con una transacción, de forma que justo antes del "commit" de una transacción aplicaríamos con el UoW las diferentes operaciones, agrupadas todas de una vez, con lo que el rendimiento se optimiza y especialmente se minimizan los bloqueos en base de datos.
- Por el contrario, si hiciéramos uso solamente de clases de acceso a datos (tradicionales DAL) dentro de una transacción, la transacción tendría una mayor duración y nuestros objetos estarían aplicando operaciones de la transacción mezcladas en el tiempo con lógica del dominio, por lo que el tiempo puramente para la transacción será siempre mayor con el consiguiente aumento de tiempo en bloqueos.

Mapeos

- Mapear objetos a tablas.
- Mapear objetos a columnas.
- Mapear propiedades a columnas
- Expresar relaciones de 1 a 1
- Expresar relaciones de 1 a N
- Expresar relaciones de N a N
- Mapear árbol de herencia



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 Hibernate.
    - Se utiliza para crear y eliminar instancias de entidad persistentes.

ORM vs JDBC

JPA, Hibernate

- Ventajas
  - Nos permite desarrollar mucho mas rápido.
  - Permite trabajar con la base de datos por medio de entidades en vez de consultas.
  - Nos ofrece un paradigma 100% orientado a objetos.
  - Elimina errores en tiempo de ejecución.
  - Mejora el mantenimiento del software.
- Desventajas
  - No ofrece toda la funcionalidad que ofrecería utilizar consultas nativas.
  - El rendimiento puede ser mucho mas bajo que realizar las consultas por JBDC.
  - Puede representar una curva de aprendizaje mas grande.

JDBC

- Ventajas
  - Permite explotar al máximo las funcionalidades de la base de datos.
  - Ofrece un rendimiento superior ya que es la forma mas directa de mandar instrucciones la base de datos.
  - Permite la optimización de consultas.
- Desventajas
  - Introduce errores “sintácticos” en tiempo de ejecución.
  - El mantenimiento es mucho mas costoso.
  - El desarrollo es mucho mas lento.
  - Incompatibilidad de los “dialectos” SQL.

OPERACIONES BÁSICAS CON HIBERNATE

Estrategias

- Hibernate permite el uso de diferentes estrategias a la hora de realizar el mapeo y el acceso:
  - Uso de las API nativas de Hibernate y mapeo hbm.xml
  - Uso de las API nativas de Hibernate y asignaciones de anotaciones
  - Uso de la API de persistencia de Java (JPA)

API nativas de Hibernate y mapeo hbm.xml

- 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).
  - Entidad (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)

Estructura de datos

- 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 Profesores (
    id INT NOT NULL AUTO_INCREMENT,
    nombre VARCHAR(255) NOT NULL,
    apellidos VARCHAR(255) NULL,
    edad INT NULL,
    PRIMARY KEY (id)
) COLLATE='latin1_spanish_ci'

INSERT INTO curso.profesores (nombre, apellidos, edad) VALUES ('Pepito', 'Grillo', '55');
INSERT INTO curso.profesores (nombre, apellidos, edad) VALUES ('Carmelo', 'Coton', '25');
```

- La base de datos se puede generar directamente con Hibernate (Code First)

Entidades de Dominio

- Una entidad es cualquier objeto del dominio que mantiene un estado y comportamiento más allá de la ejecución de la aplicación y que necesita ser distinguido de otro que tenga las mismas propiedades y comportamientos.
- Es un tipo de clase dedicada a representar un modelo de dominio persistente que:
  - 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: una propiedad.
  - Debería sobrescribir los métodos equals y hashCode: comparar la clave primaria.
  - Debería implementar el interfaz Serializable para utilizar de forma remota.
- Habitualmente, el nombre de la clase de Java (en singular) es igual al de la tabla que vamos hacer referencia.

Patrón Agregado (Aggregate)

- Una Agregación es un grupo de entidades asociadas que deben tratarse como una unidad a la hora de manipular sus datos.
- El patrón Agregado es ampliamente utilizado en los modelos de datos basados en Diseños Orientados al Dominio (DDD).
- Proporciona un forma de encapsular nuestras entidades así como los accesos y relaciones que se establecen entre las mismas de manera que se simplifique la complejidad del sistema en la medida de lo posible.
- Cada Agregación cuenta con una Entidad Raíz (root) y una Frontera (boundary):
  - La Entidad Raíz es una Entidad contenida en la Agregación de la que colgarán el resto de entidades del agregado y será el único punto de entrada a la Agregación.
  - La Frontera define qué está dentro de la Agregación y qué no.
- La Agregación es la unidad de persistencia, se recupera toda y se almacena toda.

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 principal significa que el método equals() solamente debe comparar las propiedades que forman la clave de principal.

```
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();
   }
}
```


Entidad

```
@SuppressWarnings("serial")
public class Profesor implements Serializable {
	private int id;
	private String nombre, apellidos;
	private int edad;

	public Profesor() {}
	public Profesor(int id, String nombre, String apellidos, int edad) {
		this.id = id;
		this.nombre = nombre;
		this.apellidos = apellidos;
		this.edad = edad;
	}

	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 getApellidos() { return apellidos; }
	public void setApellidos(String apellidos) { this.apellidos = apellidos; }
	public int getEdad() { return edad; }
	public void setEdad(int edad) { this.edad = edad; }

	@Override
	public int hashCode() {
		return id;
	}
	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null || getClass() != obj.getClass())
			return false;
		return id != ((Profesor) obj).id;
	}
	@Override
	public String toString() {
		return "Persona [id=" + id + ", nombre=" + nombre + " " + apellidos +
			"]";
	}
}
```

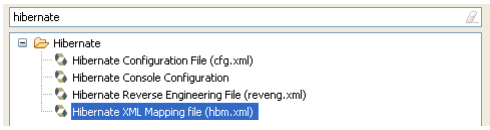
Repositorio

- Un repositorio es una clase que actúa de mediador entre el dominio de la aplicación y los datos que le dan persistencia.
- Su objetivo es abstraer y encapsular todos los accesos a la fuente de datos.
- Oculta completamente los detalles de implementación de la fuente de datos a sus clientes.
- El interfaz expuesto por el repositorio no cambia aunque cambie la implementación de la fuente de datos subyacente (diferentes esquemas de almacenamiento).
- Se crea un repositorio por cada entidad agregada de dominio que ofrece los métodos CRUD (Create-Read-Update-Delete), de búsqueda, ordenación y paginación.

Mapeado 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 →



Profesor.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
                    "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="com.example">
 <class name="Profesor" table="Profesores">
  <id column="ID" name="id" type="integer" >
   <generator class="increment"/>
  </id>
  <property name="nombre"/>
  <property name="apellidos"/>
  <property name="edad"/>
 </class>
</hibernate-mapping>
```

Mapeo de la Clase
Name= Paquete.Clase
Table= Tabla de la BBDD

Propiedad de la clase que es la clave primaria
Column=Columna de la BBDD asociada
Name= Nombre de la propiedad JAVA
Type= Tipo de Java

Declarar de las demás propiedades, para su utilización.
Si no las declaramos no se leerán/guardarán en la BBDD.
Name= Nombre de la propiead JAVA
Column= Columna de la BBDD Asociada

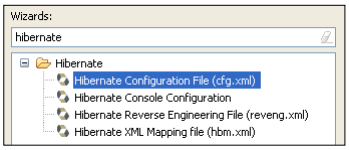
Configuración de Hibernate

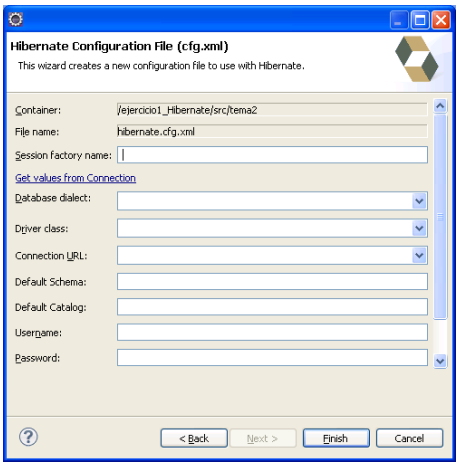
- Para poder realizar la conexión con la BBDD, necesitamos un fichero donde aparezca dicha información: hibernate.cfg.xml.
- Este fichero deberemos guardarlo en el paquete raíz de nuestras clases Java, es decir fuera de cualquier paquete.
- La información que contiene es la siguiente:
  - Propiedades 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

Configuración de Hibernate

NEW→ Others → Find Hibernate →





Configuración 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 name="">
  <!-- Database connection settings -->
  <property name="connection.driver_class">org.h2.Driver</property>
  <property name="connection.url">jdbc:h2:mem:db1;DB_CLOSE_DELAY=-1</property>
  <property name="connection.username">sa</property>
  <property name="connection.password"/>
  <property name="dialect">org.hibernate.dialect.H2Dialect</property>
  <!-- JDBC connection pool (use the built-in) -->
  <property name="connection.pool_size">1</property>
  <!-- Disable the second-level cache  -->
  <property name="cache.provider_class">org.hibernate.cache.internal.NoCacheProvider</property>
  <!-- Echo all executed SQL to stdout -->
  <property name="show_sql">true</property>
  <!-- Drop and re-create the database schema on startup -->
  <property name="hbm2ddl.auto">create</property>
  <mapping resource="com/example/Profesor.hbm.xml"/>
 </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.

Configuración 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

Configuración de Hibernate

- <mapping>
  - Para indicar que clase queremos mapear y contra qué tabla
  - Para utilizar el fichero de mapeo definido anteriormente (tabla.hbm.xml) se utiliza el atributo resource:

    <mapping resource="com/example/Profesor.hbm.xml"/>
  - Para mapear una clase que define las anotaciones utilizamos el atributo class:

    <mapping class="com.example.Profesor"/>

SessionFactory

- Para poder dar funcionalidad a todo esto, necesitaremos 3 elementos:
  - Un objeto Session, contiene métodos para leer, guardar o borrar entidades sobre la base de datos.
  - Un objeto SessionFactory, que nos permite crear los objetos Session para toda la comunicación.
  - Un objeto Configuration nos permite leer el fichero de configuración de Hibernate (hibernate.cfg.xml) y crear el objeto SessionFactory.
- La primera operación a realizar es la creación del Objeto SessiónFactory. Este objeto, al ser muy costoso, debería ser único por aplicación y lo podremos reutilizar en todas las llamadas que necesitemos.
- Se puede implementar en una clase asistente utilizando el patrón singleton.

HibernateUtil

```
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) {
            StandardServiceRegistryBuilder.destroy( registry );
            System.err.println("Initial SessionFactory creation failed." + ex);
            throw new ExceptionInInitializerError(ex);
        }
    }
    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }
}
```

HibernateUtil (JPA)

```
public class HibernateUtil {
    private static final EntityManagerFactory emf = buildEntityManagerFactory();

    private static EntityManagerFactory buildEntityManagerFactory() {
        try {
            return Persistence.createEntityManagerFactory("JPA_PU");
        }
        catch (Exception ex) {
            System.err.println("Initial EntityManagerFactory creation failed. " + ex);
            throw new ExceptionInInitializerError(ex);
        }
    }
    public static EntityManagerFactory getEntityManagerFactory() {
        return emf;
    }
}
```


Configuración por código

- Una instancia de org.hibernate.cfg.Configuration representa un conjunto completo de asignaciones de tipos de Java de una aplicación a una base de datos SQL. Las asignaciones se compilan a partir de varios archivos de asignación XML.
- La Configuration se puede obtener creando una instancia directamente y especificar luego los metadatos de mapeo (documentos de mapeo XML, clases anotadas) que describen el modelo de objetos de la aplicación y su mapeo a una base de datos SQL.

```
Configuration cfg = new Configuration()
    .addResource( "Item.hbm.xml" ) // addResource does a classpath resource lookup
    .addClass( com.example.User.class ) // calls addResource using "/org/hibernate/auction/User.hbm.xml"
    .addAnnotatedClass( Address.class ) // parses Address class for mapping annotations
    .addPackage( "com.example.entities" )
    .setProperty( "hibernate.dialect", "org.hibernate.dialect.H2Dialect" )
    .setProperty( "hibernate.connection.datasource", "java:comp/env/jdbc/test" )
    .setProperty( "hibernate.order_updates", "true" );
```

- Otras alternativas son:
  - Colocar un archivo con nombre hibernate.properties en un directorio raíz de la ruta de clases.
  - Establecer propiedades System usando java -Dproperty=value.

Configuración por código (v5.0)

- Una instancia de org.hibernate.cfg.Configuration representa un conjunto completo de asignaciones de tipos de Java de una aplicación a una base de datos SQL. Las asignaciones se compilan a partir de varios archivos de asignación XML.
- El StandardServiceRegistry es altamente configurable a través de los metadatos:

```
ServiceRegistry standardRegistry = new StandardServiceRegistryBuilder().build();
MetadataSources sources = new MetadataSources( standardRegistry )
    .addAnnotatedClass( MyEntity.class )
    .addAnnotatedClassName( "com.example.User" )
    .addResource( "com/example/Item.hbm.xml" )
    .addPackage( MyEntity.class.getPackage() );
SessionFactory sessionFactory = sources.getMetadataBuilder().build()
    .getSessionFactoryBuilder().build();
```

- Otras alternativas son:
  - Colocar un archivo con nombre hibernate.properties en un directorio raíz de la ruta de clases.
  - Establecer propiedades System usando java -Dproperty=value.

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();
```

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 (confirmación).
    session.getTransaction().commit();
  - Descarte de la transacción actual (cancelación).
    session.getTransaction().rollback();
  - Cierre de la transacción actual.
    session.closeTransaction();

Operaciones CRUD

- Entendemos por Operaciones CRUD las operaciones básicas que se realizan en toda tabla:
  - Create: Insertar un nuevo objeto en la base de datos.
  - Read: Leer los datos de un objeto de la base de datos.
  - Update: Actualizar los datos de un objeto de la base de datos.
  - Delete: 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.
  - Create: session.save(…..);
  - Read: session.get(….);
  - Update: session.update(…..);
  - Delete: session.delete(….);
  - Actualizar o Insertar

Creación / Inserción

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

```
Session session = HibernateUtil.getSessionFactory().openSession();
session.beginTransaction();
session.save(new Profesor(1, "Pepito", "Grillo", 55));
session.save(new Profesor(2, "Carmelo", "Coton", 25));
session.getTransaction().commit();
session.close();
```

SI aparece el error: Error parsing JNDI name

- Eliminar el elemento name del fichero hibernate.cfg.xml
- <session-factory name=" "> → <session-factory >

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 (null si no lo encuentra)

```
Session session = HibernateUtil.getSessionFactory().openSession();
Profesor profesor = session.get(Profesor.class, 2);
// Profesor profesor = session.byId(Profesor.class).load(2); // Excepción si no lo encuentra
// Optional<Profesor> optionalProfesor = session.byId(Profesor.class ).loadOptional(0);
System.out.println(profesor);
session.close();
```

- En Hibernate existen 3 maneras para realizar consultas de múltiples elementos.
  - Hibernate Query Language (HQL) o Java Persistence Query Language (JPQL)
    - session.createQuery("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);

Lectura múltiple

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

session = HibernateUtil.getSessionFactory().openSession();
session.createQuery("from Profesores")
      .stream().forEach(item -> System.out.println(item));
session.close();
```


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
- Siempre deberemos validar la transacción antes de su finalización.

```
session = HibernateUtil.getSessionFactory().openSession();
session.beginTransaction();
Profesor profesor= session.get(Profesor.class, 1);
profesor.setNombre("Pedro");
// Profesor profesor= new Profesor(1, "Pedro", "Pica Piedra", 65);
session.update(profesor);
// session.flush();
session.getTransaction().commit();
session.close();
```

- Para actualizaciones múltiples se requiere una consulta:

```
int updatedEntities = session.createQuery("update Profesores set nombre = :act  where nombre = :ant" )
      .setParameter( "ant", oldName ).setParameter( "act", newName ).executeUpdate();
```

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.

```
session = HibernateUtil.getSessionFactory().openSession();
session.beginTransaction();
Profesor profesor= new Profesor(101, "Pedro", "Pica Piedra", 65);
session.saveOrUpdate(profesor);
session.getTransaction().commit();
session.close();
```


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
- Siempre deberemos validar la transacción antes de su finalización.

```
session = HibernateUtil.getSessionFactory().openSession();
session.beginTransaction();
Profesor profesor= session.get(Profesor.class, 1);
session.delete(profesor);
session.getTransaction().commit();
session.close();
```

- Para borrados múltiples se requiere una consulta:

```
int updatedEntities = session.createQuery("delete Profesores set nombre = :nombre" )
      .setParameter( "nombre", "Pedro").executeUpdate();
```

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
  - Las anotaciones permiten especificar de una forma compacta y sencilla la información de mapeo de las clases Java
- 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

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

    ~~<mapping resource="com/example/Profesor.hbm.xml"/>~~

    <mapping class="com.example.Profesor"/>
- En los Ficheros de mapeo.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 (campos o atributos que tengan los métodos get/set), de hecho es necesario marcar con @Transient para indicar que un atributo no es persistente.

Estrategias de acceso

- Hibernate puede interactuar tanto con los atributos de entidad (campos de instancia) como los accesores (propiedades de instancia). De forma predeterminada, la ubicación de la anotación @Id proporciona la estrategia de acceso predeterminada.
- Cuando se coloca en un campo, Hibernate asumirá el acceso basado en el campo:
  - Agregar otros métodos a nivel de entidad es mucho más flexible porque Hibernate no los considerará como parte del estado de persistencia.
  - Para excluir un campo de ser parte del estado persistente de la entidad debe estar marcado con la anotación @Transient.
  - Permite ocultar atributos de la entidad desde fuera de la entidad simplemente omitiendo los accesores.
- Cuando se coloca en el captador de identificador (get), Hibernate utilizará el acceso basado en propiedades:
  - Usa los accesores tanto para leer como para escribir el estado de la entidad, que al ser métodos permiten definir la entrada/salida de los datos.
  - Todos los demás métodos que se agregarán a la entidad deberán marcarse con la anotación @Transient.
- El mecanismo de estrategia de acceso predeterminado se puede anular con la anotación @Access.

Entidad Anotada

```
@SuppressWarnings("serial")
@Entity
@Table(name="Profesores")
public class Profesor implements Serializable {
	@Id
	@Column(name="Id")
	@GeneratedValue(strategy=GenerationType.IDENTITY)
	private int id;
	private String nombre, apellidos;
	private int edad;

	public Profesor() {}
	public Profesor(int id, String nombre, String apellidos, int edad) {
		this.id = id;
		this.nombre = nombre;
		this.apellidos = apellidos;
		this.edad = edad;
	}
	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 getApellidos() { return apellidos; }
	public void setApellidos(String apellidos) { this.apellidos =
apellidos; }
	public int getEdad() { return edad; }
	public void setEdad(int edad) { this.edad = edad; }

	@Override
	public int hashCode() {
		return id;
	}

	// …
}
```


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

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.2" 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_2.xsd">
    <persistence-unit name="JPA_PU" transaction-type="RESOURCE_LOCAL">
        <class>com.example.entities.Actor</class>
        <!-- … -->
        <class>com.example.entities.Store</class>
        <properties>
            <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/sakila"/>
            <property name="javax.persistence.jdbc.user" value="root"/>
            <property name="javax.persistence.jdbc.password" value="root"/>
            <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
            <property name="hibernate.show_sql" value="true" />
        </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("JPA_PU");
- 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 = "JPA_PU")
  - 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();

Entidades sobre consultas SQL

- Se puede asignar una entidad a una consulta SQL nativa mediante la anotación @Subselect.

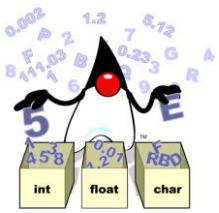
```
@Entity
@Subselect(
    "select a.id as id, concat(concat(c.first_name, ' '), c.last_name) as name, sum(atr.cents) as balance " +
    "from account a join client c on c.id = a.client_id join account_transaction atr on a.id = atr.account_id " +
    "group by a.id, concat(concat(c.first_name, ' '), c.last_name)")
@Synchronize( {"client", "account", "account_transaction"} )
public static class AccountSummary {
    @Id
    private Long id;
    private String name;
    private int balance;
    :
```

- La anotación @Synchronize permite indicar a Hibernate qué tablas de base de datos son necesarias para la consulta SQL subyacente (a diferencia de las consultas JPQL y HQL, Hibernate no puede analizar la consulta SQL nativa) e invalidar la consulta si alguna cambia.

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.



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"/>

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.

Standard Basic Types

Hibernate type	JDBC type	Java type	BasicTypeRegistry key(s)
StringType	VARCHAR	java.lang.String	string, java.lang.String
MaterializedClob	CLOB	java.lang.String	materialized_clob
TextType	LONGVARCHAR	java.lang.String	text
CharacterType	CHAR	char, java.lang.Character	character, char, java.lang.Character
BooleanType	BOOLEAN	boolean, java.lang.Boolean	boolean, java.lang.Boolean
NumericBooleanType	INTEGER, 0 is false, 1 is true	boolean, java.lang.Boolean	numeric_boolean
YesNoType	CHAR, 'N'/'n' is false, 'Y'/'y' is true. The uppercase value is written to the database.	boolean, java.lang.Boolean	yes_no
TrueFalseType	CHAR, 'F'/'f' is false, 'T'/'t' is true. The uppercase value is written to the database.	boolean, java.lang.Boolean	true_false
ByteType	TINYINT	byte, java.lang.Byte	byte, java.lang.Byte
ShortType	SMALLINT	short, java.lang.Short	short, java.lang.Short
IntegerType	INTEGER	int, java.lang.Integer	integer, int, java.lang.Integer
LongType	BIGINT	long, java.lang.Long	long, java.lang.Long

Standard Basic Types

Hibernate type	JDBC type	Java type	BasicTypeRegistry key(s)
FloatType	FLOAT	float, java.lang.Float	float, java.lang.Float
DoubleType	DOUBLE	double, java.lang.Double	double, java.lang.Double
BigIntegerType	NUMERIC	java.math.BigInteger	big_integer, java.math.BigInteger
BigDecimalType	NUMERIC	java.math.BigDecimal	big_decimal, java.math.bigDecimal
TimestampType	TIMESTAMP	java.util.Date	timestamp, java.sql.Timestamp, java.util.Date
DbTimestampType	TIMESTAMP	java.util.Date	dbtimestamp
TimeType	TIME	java.util.Date	time, java.sql.Time
DateType	DATE	java.util.Date	date, java.sql.Date
CalendarType	TIMESTAMP	java.util.Calendar	calendar, java.util.Calendar, java.util.GregorianCalendar
CalendarDateType	DATE	java.util.Calendar	calendar_date
CalendarTimeType	TIME	java.util.Calendar	calendar_time
CurrencyType	VARCHAR	java.util.Currency	currency, java.util.Currency

Standard Basic Types

Hibernate type	JDBC type	Java type	BasicTypeRegistry key(s)
LocaleType	VARCHAR	java.util.Locale	locale, java.util.Locale
TimeZoneType	VARCHAR, using the TimeZone ID	java.util.TimeZone	timezone, java.util.TimeZone
UrlType	VARCHAR	java.net.URL	url, java.net.URL
ClassType	VARCHAR (class FQN)	java.lang.Class	class, java.lang.Class
BlobType	BLOB	java.sql.Blob	blob, java.sql.Blob
ClobType	CLOB	java.sql.Clob	clob, java.sql.Clob
BinaryType	VARBINARY	byte[]	binary, byte[]
MaterializedBlobType	BLOB	byte[]	materialized_blob
ImageType	LONGVARBINARY	byte[]	image
WrapperBinaryType	VARBINARY	java.lang.Byte[]	wrapper-binary, Byte[], java.lang.Byte[]
CharArrayType	VARCHAR	char[]	characters, char[]
CharacterArrayType	VARCHAR	java.lang.Character[]	wrapper-characters, Character[], java.lang.Character[]
CharacterNCharType	NCHAR	java.lang.Character	ncharacter

Standard Basic Types

Hibernate type	JDBC type	Java type	BasicTypeRegistry key(s)
UUIDBinaryType	BINARY	java.util.UUID	uuid-binary, java.util.UUID
UUIDCharType	CHAR, can also read VARCHAR	java.util.UUID	uuid-char
PostgresUUIDType	PostgreSQL UUID	java.util.UUID	pg-uuid
SerializableType	VARBINARY	implementors of java.lang.Serializable	Unlike the other value types, multiple instances of this type are registered.
StringNVarcharType	NVARCHAR	java.lang.String	nstring
NTextType	LONGNVARCHAR	java.lang.String	ntext
NClobType	NCLOB	java.sql.NClob	nclob, java.sql.NClob
MaterializedNClobType	NCLOB	java.lang.String	materialized_nclob
PrimitiveCharacterArrayNClobType	NCHAR	char[]	N/A
CharacterArrayNClobType	NCLOB	java.lang.Character[]	N/A
RowVersionType	VARBINARY	byte[]	row_version
ObjectType	VARCHAR	implementors of java.lang.Serializable	object, java.lang.Object

Standard Basic Types

Hibernate type	JDBC type	Java type	BasicTypeRegistry key(s)
Java 8 BasicTypes			
DurationType	BIGINT	java.time.Duration	Duration, java.time.Duration
InstantType	TIMESTAMP	java.time.Instant	Instant, java.time.Instant
LocalDateTimeType	TIMESTAMP	java.time.LocalDateTime	LocalDateTime, java.time.LocalDateTime
LocalDateType	DATE	java.time.LocalDate	LocalDate, java.time.LocalDate
LocalTimeType	TIME	java.time.LocalTime	LocalTime, java.time.LocalTime
OffsetDateTimeType	TIMESTAMP	java.time.OffsetDateTime	OffsetDateTime, java.time.OffsetDateTime
OffsetTimeType	TIME	java.time.OffsetTime	OffsetTime, java.time.OffsetTime
ZonedDateTimeType	TIMESTAMP	java.time.ZonedDateTime	ZonedDateTime, java.time.ZonedDateTime
Hibernate Spatial BasicTypes			
JTSGeometryType	depends on the dialect	com.vividsolutions.jts.geom.Geometry	jts_geometry, and the class names of Geometry and its subclasses
GeolatteGeometryType	depends on the dialect	org.geolatte.geom.Geometry	geolatte_geometry, and the class names of Geometry and its subclasses

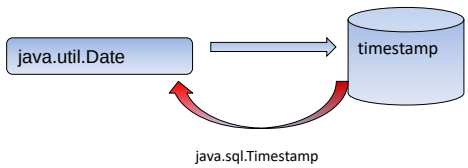
Fecha y Hora

- 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

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()

@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
- Para generar las fecha automáticamente:
  - @CreationTimestamp indica a Hibernate que establezca el atributo con el valor de marca de tiempo actual de la JVM cuando la entidad se persiste (solo cuando se crea).
  - @UpdateTimestamp indica a Hibernate que establezca el atributo con el valor de marca de tiempo actual de la JVM cuando la entidad se persiste (siempre).

Boolean

- Hibernate permite múltiples formas distintas de almacenar un booleano de Java en la base de datos.
- Para ello existen 4 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.
  - numeric_boolean:
    - El valor se guardará como un INT con los valores de 1 y 0 para true y false respectivamente.

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.

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

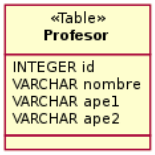
```
@Lob
@Column(name = "ITEM_DESCRIPTION")
private String description;
```

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.



Generador de identificadores

- Hibernate puede generar y completar valores de identificadores automáticamente. Este es el enfoque recomendado para las claves artificiales.
- Hibernate ofrece varias estrategias de generación (estandarizadas por JPA):
  - IDENTITY: admite columnas de identidad en DB2, MySQL, MS SQL Server, Sybase y HypersonicSQL. El identificador devuelto es de tipo long, short o int.
  - SECUENCIA (llamado seqhilo en Hibernate): utiliza una hi / lo algoritmo para generar eficientemente identificadores de tipo long, short o int, dada una secuencia de base de datos llamada.
  - TABLE (llamado MultipleHiLoPerTableGenerator en Hibernate): utiliza un algoritmo hi / lo para generar de manera eficiente identificadores de tipo long, short o int, dada una tabla y columna como fuente de valores hi. El algoritmo hi / lo genera identificadores que son únicos solo para una base de datos en particular.
  - AUTO: Selecciona IDENTITY, SEQUENCE o TABLE dependiendo de las capacidades de la base de datos subyacente.

Generador de identificadores

- SEQUENCE y TABLE requieren configuraciones adicionales que puede establecer usando @SequenceGenerator y @TableGenerator:
  - name: nombre del generador
  - table/sequenceName: nombre de la tabla o de la secuencia (por defecto respectivamente a hibernate_sequencesy hibernate_sequence)
  - catalog/schema: catálogo o esquema de la tabla o secuencia
  - initialValue: el valor a partir del cual el id debe comenzar a generar
  - allocationSize: la cantidad a incrementar al asignar números de identificación desde el generador
- Además, la estrategia TABLE también te permite personalizar:
  - pkColumnName: el nombre de la columna que contiene el identificador de la entidad
  - valueColumnName: el nombre de la columna que contiene el valor del identificador
  - pkColumnValue: el identificador de la entidad
  - uniqueConstraints: cualquier restricción de columna potencial en la tabla que contiene los identificadores

Generadores adicionales

- Increment: genera identificadores de tipo long, short o int que son únicos solo cuando ningún otro proceso está insertando datos en la misma tabla. No usar en un grupo.
- Uuid: genera un UUID de 128 bits basado en un algoritmo personalizado. El valor generado se representa como una cadena de 32 dígitos hexadecimales. Se pueden configurar para usar un separador que separe los dígitos hexadecimales.
- Uuid2: genera un UUID de 128 bits compatible con IETF RFC 4122 (variante 2).
- Guid: utiliza una cadena GUID generada por la base de datos en MS SQL Server y MySQL.
- Assigned: permite que la aplicación asigne un identificador al objeto antes de que se llame a save(). Esta es la estrategia predeterminada si no se especifica ningún elemento <generator>.
- Select: recupera una clave primaria, asignada por un disparador de la base de datos, seleccionando la fila por alguna clave única y recuperando el valor de la clave primaria.
- Foreign: utiliza el identificador de otro objeto asociado. Por lo general, se usa junto con una asociación <one-to-one> de clave primaria.

Establecer la clave primaria

- Para definir las Claves primarias en Hibernate lo hacemos en el fichero de mapeo de la clase Java donde esté la PK

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.

Establecer la clave primaria

- 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.
- 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")
  - @org.hibernate.annotations.GenericGenerator (.....)

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.

Claves Primarias Compuestas

- Los identificadores compuestos corresponden a uno o más atributos persistentes. Estas son las reglas que rigen los identificadores compuestos, según lo definido por la especificación JPA:
  - El identificador compuesto debe estar representado por una "clase de clave primaria". La clase de clave primaria se puede definir usando la anotación javax.persistence.EmbeddedId (@EmbeddedId), o definirse usando la anotación javax.persistence.IdClass (@IdClass).
  - La clase de clave primaria debe ser pública y debe tener un constructor público sin argumentos.
  - La clase de clave primaria debe ser serializable.
  - La clase de clave primaria debe definir métodos equals y hashCode, de acuerdo con la igualdad para los tipos de base de datos subyacentes a los que se asigna la clave primaria.

- La imposición de que un identificador compuesto debe estar representado por una "clase de clave primaria" (por ejemplo, @EmbeddedIdo @IdClass) es específica de JPA, Hibernate permite definir identificadores compuestos sin la "clase de clave primaria" mediante múltiples atributos @Id.

Identificadores compuestos: @EmbeddedId

```
@Embeddable
public class FilmActorPK implements Serializable {
    private int actorId;
    private int filmId;
    public FilmActorPK() { }
    public int getActorId() { return this.actorId; }
    public void setActorId(int actorId) { this.actorId = actorId; }
    public int getFilmId() { return this.filmId;    }
    public void setFilmId(int filmId) { this.filmId = filmId; }
    public boolean equals(Object other) {
        if (this == other) return true;
        if (!(other instanceof FilmActorPK))  return false;
        FilmActorPK castOther = (FilmActorPK)other;
        return (this.actorId == castOther.actorId) &&
            (this.filmId == castOther.filmId);
    }
    public int hashCode() {
        // …
    }
}
```

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

    @EmbeddedId
    private FilmActorPK id;
```

Identificadores compuestos: @IdClass

```
public class FilmActorPK implements Serializable {
        private int actorId;
        private int filmId;
        public FilmActorPK() { }
        public FilmActorPK(int actorId, int filmId) {
                this.actorId = id.getActorId();
                this.filmId = id.getFilmId();
        }
        public int getActorId() { return this.actorId; }
        public void setActorId(int actorId) { this.actorId = actorId; }
        public int getFilmId() { return this.filmId;            }
        public void setFilmId(int filmId) { this.filmId = filmId; }
        public boolean equals(Object other) {
                if (this == other) return true;
                if (!(other instanceof FilmActorPK))  return false;
                FilmActorPK castOther = (FilmActorPK)other;
                return (this.actorId == castOther.actorId) &&
                        (this.filmId == castOther.filmId);
        }
        public int hashCode() {
                // …
        }
}
```

```
@Entity
@Table(name="film_actor")
@IdClass(FilmActorPK.class)
public class FilmActor implements Serializable {
        @Id
        private int actorId;
        @Id
        private int filmId;

        public FilmActorPK getId() {
                return new FilmActorPK(actorId, filmId);
        }

        public void setId(FilmActorPK id) {
                actorId = id.getActorId();
                filmId = id.getFilmId();
        }
```


Claves naturales

- Los identificadores naturales (claves alternativas) representan identificadores únicos del modelo de dominio que también tienen un significado en el mundo real. Incluso si un identificador natural no es una buena clave primaria (generalmente se prefieren las claves sustitutas), es útil informar a Hibernate al respecto.
- En Hibernate, las claves naturales se utilizan a menudo para búsquedas. El identificador primario será autogenerado en la mayoría de los casos. Pero esta identificación es bastante inútil para las búsquedas, ya que siempre se consultará por campos como nombre, número de seguridad social o cualquier otra cosa del mundo real.
- Al utilizar las funciones de almacenamiento en caché de Hibernate, esta diferencia es muy importante: si la caché está indexada por su clave principal (ID sustituta), no habrá ninguna ganancia de rendimiento en las búsquedas. Es por eso se pueden definir un conjunto de campos con los que se consultará la base de datos: claves alternativas. Hibernate puede entonces indexar los datos por su clave natural y mejorar el rendimiento de búsqueda.
- Para marcar un atributo como clave natural o alternativa:

```
@NaturalId
private String username;
```

Otras claves

- Si se anota una entidad determinada con la @RowId y la base de datos subyacente admite la obtención de un registro por ROWID (por ejemplo, Oracle), entonces Hibernate puede usar la pseudocolumna ROWID para operaciones CRUD.

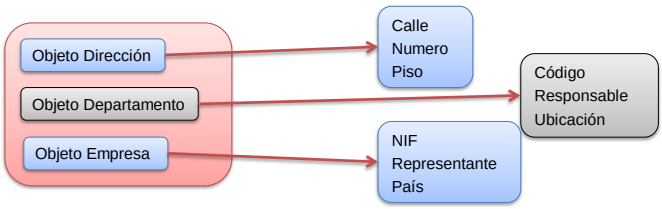
```
@Entity
@RowId("ROWID")
public static class Profesor {
    private Long id;
```

- Con la anotación @MapsId, JPA 2.0 agregó soporte para identificadores derivados que permiten a una entidad tomar prestado el identificador de una asociación de uno a uno o muchos a uno.

```
@Entity
public static class ProfesorDetails  {
    @Id
    private Long id;
    @OneToOne
    @MapsId
    private Profesor profesor;
```


Componentes

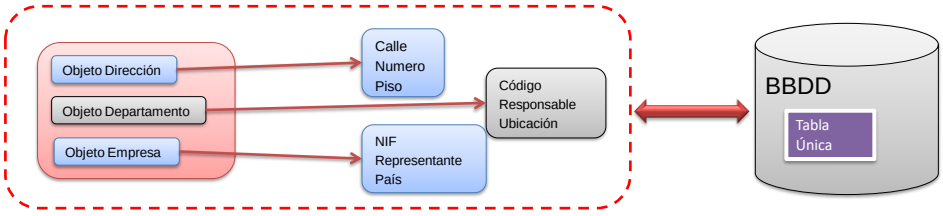
- Lenguajes orientados a objetos como Java hacen más fácil definir objetos complejos como composición de otros ya definidos.
  - Nombre Cliente, Dirección, Departamento → Objeto cliente



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

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.

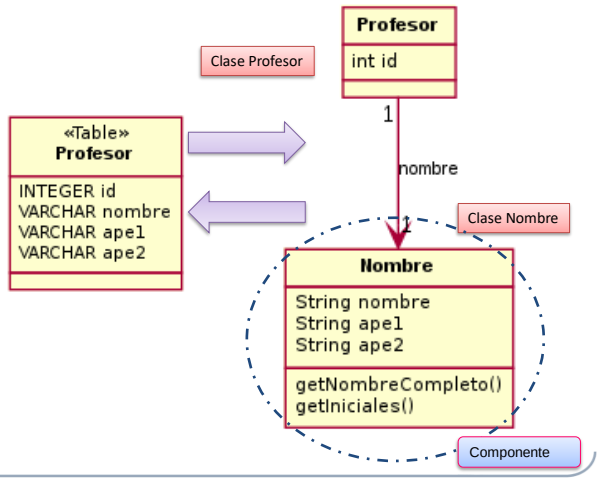


- Estos componentes son tratados como un «value type», como un tipo primitivo, String, etc
  - Los "value types" forman parte de una única clase

Clase Dirección → String (Calle+Numero+Piso)

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.



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.

```
<?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

- 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;
```

- 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="nom")
   private String nombre;
   private String ape1;
   private String ape2;
```

Enumerados

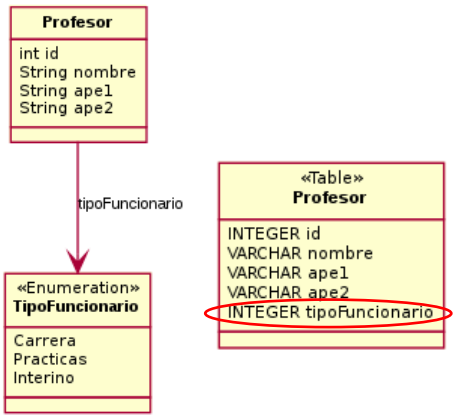
- Los tipos enumerados en Java, permiten que una variable tenga un conjunto de valores restringidos.
- Son utilizados con valores fijos (meses del año, colores, etc)

  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.
- Hibernate admite el mapeo de enumeraciones de Java como tipos de valores básicos de acuerdo con una de las 2 estrategias indicadas por javax.persistence.EnumType:
  - ORDINAL: almacenado de acuerdo con el valor numérico (base 1) de la posición ordinal del valor de enumeración dentro de la clase de enumeración
  - STRING: almacenado de acuerdo con el nombre del valor de enumeración

Enumerados

```
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

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

Enumerados

```
<?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

Anotación:
@Enumerated(EnumType.ORDINAL)
private TipoFuncionario tipo;

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 (depende de los dialectos).

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

- Es necesario especificar las columnas calculadas, de solo lectura, cuyas propiedades debe ignorar la persistencia:
  - NEVER (por defecto): el valor de propiedad no se genera dentro de la base de datos.
  - 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;
```


Especiales

- La anotación @GeneratorType se utiliza para que pueda proporcionar un generador personalizado para establecer el valor de la propiedad anotada actualmente.

```
public static class CurrentUser {
        public static final CurrentUser INSTANCE = new CurrentUser();
        private static final ThreadLocal<String> storage = new ThreadLocal<>();
        public void logIn(String user) { storage.set( user );                }
        public void logOut() { storage.remove(); }
        public String get() { return storage.get(); }
}
public static class LoggedUserGenerator implements ValueGenerator<String> {
        @Override
        public String generateValue(Session session, Object owner) {
                return CurrentUser.INSTANCE.get();
        }
}

        @GeneratorType( type = LoggedUserGenerator.class, when = GenerationTime.ALWAYS)
        private String updatedBy;
```

Especiales

- Hibernate permite personalizar el SQL que usa para leer y escribir los valores de las columnas asignadas a los tipos @Basic.

```
@Column(name = "pswd")
@ColumnTransformer(
        read = "decrypt( 'AES', '00', pswd  )",
        write = "encrypt('AES', '00', ?)"
)
private String password;
```

- Para implementar las transformaciones personalizadas en Java se utilizan los conversores. Para crear un conversor es necesario crear una clase que implemente AttributeConverter<Propiedad, Columna> y este anotada con @Converter. Los conversores se aplican con la anotación @Convert.

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;
```

Tipos personalizados

- Hibernate hace que sea relativamente fácil para los desarrolladores crear su propio tipo de mapeo de tipos básico.
- Hay dos enfoques para desarrollar un tipo personalizado:
  - implementar un BasicType y registrarlo
  - implementar un UserType que no requiere registro de tipo

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, HashMap o TreeMap, en función del tipo de interfaz.

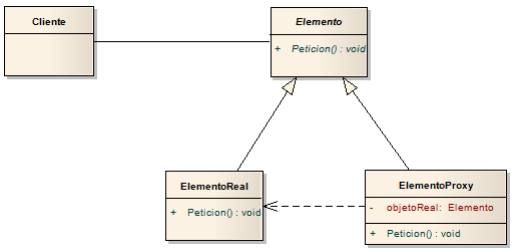
Tipos de Proxy

- 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.



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();
        }
}
```


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
```

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()) {
   // …
}
```


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, que 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)

Modelo dinámico

- Las entidades persistentes no tienen que representarse necesariamente como clases POJO / JavaBean, aunque JPA solo reconoce el mapeo del modelo de entidad, Hibernate también soporta modelos dinámicos (utilizando Map de Maps en tiempo de ejecución). Con este enfoque, no escribe clases persistentes, solo archivos de mapeo.

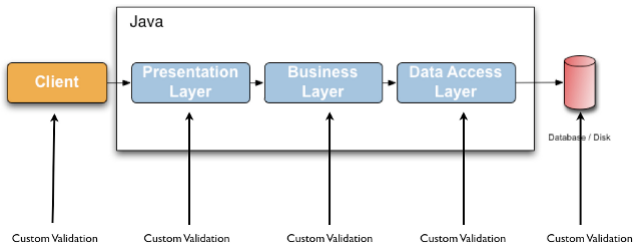
```
<hibernate-mapping>
   <class entity-name="Book">
      <id name="isbn" column="isbn" length="32" type="string"/>
      <property name="title" not-null="true" length="50" type="string"/>
      <property name="author" not-null="true" length="50" type="string"/>
   </class>
</hibernate-mapping>
```

```
Map<String, String> book = new HashMap<>();
book.put( "isbn", "978-9730228236" );
book.put( "title", "High-Performance Java Persistence" );
book.put( "author", "Vlad Mihalcea" );
entityManager.unwrap(Session.class).save( "Book", book );
```

VALIDACIONES

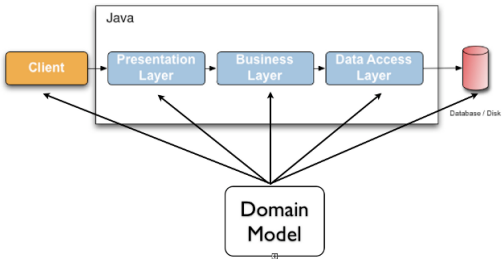
Validaciones

- La validación de datos es una tarea común que ocurre en todas las capas de la aplicación, desde la presentación hasta la capa de persistencia. A menudo, se implementa la misma lógica de validación en cada capa, lo que requiere mucho tiempo y es propenso a errores. Para evitar la duplicación de estas validaciones, los desarrolladores a menudo agrupan la lógica de validación directamente en el modelo de dominio, saturando las clases de dominio con código de validación que en realidad son metadatos sobre la clase en sí.



Validaciones

- Jakarta Bean Validation 2.0: define un modelo de metadatos y una API para la validación de entidades y métodos. La fuente de metadatos predeterminada son las anotaciones, con la capacidad de anular y ampliar los metadatos mediante el uso de XML. La API no está vinculada a un nivel de aplicación específico ni a un modelo de programación. Específicamente, no está vinculado a ningún nivel web o de persistencia, y está disponible tanto para la programación de aplicaciones del lado del servidor como para los desarrolladores de aplicaciones Swing y JavaFX de cliente enriquecido.



Validaciones

- Hibernate Validator es la implementación de referencia de Jakarta Bean Validation basada en Bean Validation 2.0. (JSR 303, JSR 349, JSR 380).
- Hibernate Validator 6 y Jakarta Bean Validation 2.0 requieren Java 8 o posterior.
- Instalación:

```
<dependency>
   <groupId>org.hibernate.validator</groupId>
   <artifactId>hibernate-validator-cdi</artifactId>
   <version>7.0.0.Final</version>
</dependency>
<dependency>
   <groupId>org.glassfish</groupId>
   <artifactId>jakarta.el</artifactId>
   <version>4.0.0</version>
</dependency>
```

Validaciones

- Este API permite validar los datos de manera declarativa, con el uso de anotaciones. Para centralizar las validaciones y evitar inconsistencias se recomiendan establecer las restricciones en las entidades de dominio.
- Las restricciones se establecen mediante anotaciones a nivel de clase, atributo de instancia, propiedades (getter), elementos de colecciones y parámetros de métodos.

```
@AssertFalse
private boolean isDeleted;

@NotNull @Positive
public int getId() { return id; }

private Set<@NotBlank String> nombres = new HashSet<>();
```


Validaciones

- Se puede exigir la validez mediante la anotación @Valid en el elemento a validar.

```
@Valid private Direccion direccion;
private Set<@NotNull @Valid CorreoElectronico> correosElectronicos;
public ResponseEntity<Object> create(@Valid @RequestBody Persona item)
```

- El primer paso para validar una instancia de entidad es obtener una instancia Validator.

```
private Validator validator = Validation.buildDefaultValidatorFactory().getValidator();
```

- Se puede validar una instancia completa, propiedades individuales de la entidad o un posible valor para una propiedad de la entidad :

```
Set<ConstraintViolation<Persona>> constraintViolations =  validator.validate( persona );
constraintViolations = validator.validateProperty( persona,  "nombre" );
constraintViolations = validator.validateValue( persona,  "nombre", "    " );
```

- Los tres métodos devuelven un Set<ConstraintViolation>: estará vacío, si la validación tiene éxito, o se agregara una instancia por cada restricción violada. ConstraintViolation contiene información útil sobre la causa del error de validación (getMessage(), getInvalidValue(), ...).

Restricciones (JSR 303)

@Null : Comprueba que el valor anotado es null

@NotNull : Comprueba que el valor anotado no sea null

@NotEmpty : Comprueba si el elemento anotado no es nulo ni está vacío

@NotBlank : Comprueba que la secuencia de caracteres anotados no sea nula y que la longitud recortada sea mayor que 0. La diferencia @NotEmpty es que esta restricción solo se puede aplicar en secuencias de caracteres y que los espacios en blanco finales se ignoran.

@AssertFalse : Comprueba que el elemento anotado es falso.

@AssertTrue : Comprueba que el elemento anotado es verdadero

Restricciones (JSR 303)

@Max(value=) : Comprueba si el valor anotado es menor o igual que el máximo especificado

@Min(value=) : Comprueba si el valor anotado es mayor o igual que el mínimo especificado

@Negative : Comprueba si el elemento es estrictamente negativo. Los valores cero se consideran inválidos.

@NegativeOrZero : Comprueba si el elemento es negativo o cero.

@Positive : Comprueba si el elemento es estrictamente positivo. Los valores cero se consideran inválidos.

@PositiveOrZero : Comprueba si el elemento es positivo o cero.

@DecimalMax(value=, inclusive=) : Comprueba si el valor númerico anotado es menor que el máximo especificado, cuando inclusive= falso. De lo contrario, si el valor es menor o igual al máximo especificado.

@DecimalMin(value=, inclusive=) : Comprueba si el valor anotado es mayor que el mínimo especificado, cuando inclusive= falso. De lo contrario, si el valor es mayor o igual al mínimo especificado.

Restricciones (JSR 303)

@Digits: El elemento anotado debe ser un número cuyo valor tenga el número de dígitos especificado.

@Past : Comprueba si la fecha anotada está en el pasado

@PastOrPresent : Comprueba si la fecha anotada está en el pasado o en el presente

@Future : Comprueba si la fecha anotada está en el futuro.

@FutureOrPresent : Comprueba si la fecha anotada está en el presente o en el futuro

@Email : Comprueba si la secuencia de caracteres especificada es una dirección de correo electrónico válida.

@Pattern(regex=, flags=) : Comprueba si la cadena anotada coincide con la expresión regular regex considerando la bandera dadamatch.

@Size(min=, max=) : Comprueba si el tamaño del elemento anotado está entre min y max (inclusive)

Restricciones (Hibernate)

@CreditCardNumber(ignoreNonDigitCharacters=): Comprueba que la secuencia de caracteres pasa la prueba de suma de comprobación de Luhn.

@Currency(value=): Comprueba que la unidad monetaria de un javax.money.MonetaryAmount forma parte de las unidades monetarias especificadas.

@DurationMax(days=, hours=, minutes=, seconds=, millis=, nanos=, inclusive=): Comprueba que el elemento java.time.Duration no sea mayor que el construido a partir de los parámetros de la anotación.

@DurationMin(days=, hours=, minutes=, seconds=, millis=, nanos=, inclusive=): Comprueba que el elemento java.time.Duration no sea menor que el construido a partir de los parámetros de anotación.

@EAN: Comprueba que la secuencia de caracteres sea un código de barras EAN válido

@ISBN: Comprueba que la secuencia de caracteres sea un ISBN válido.

Restricciones (Hibernate)

@Length(min=, max=): Valida que la secuencia de caracteres esté entre min e max incluidos

@CodePointLength(min=, max=, normalizationStrategy=): Valida que la longitud del punto de código de la secuencia de caracteres esté entre min e max incluidos.

@LuhnCheck(startIndex= , endIndex=, checkDigitIndex=, ignoreNonDigitCharacters=): Comprueba que los dígitos de la secuencia de caracteres pasan el algoritmo de suma de comprobación de Luhn.

@Mod10Check(multiplier=, weight=, startIndex=, endIndex=, checkDigitIndex=, ignoreNonDigitCharacters=): Comprueba que los dígitos dentro de la secuencia de caracteres pasan el algoritmo genérico de suma de comprobación mod 10.

@Mod11Check(threshold=, startIndex=, endIndex=, checkDigitIndex=, ignoreNonDigitCharacters=, treatCheck10As=, treatCheck11As=): Comprueba que los dígitos dentro de la secuencia de caracteres pasan el algoritmo de suma de comprobación mod 11.

Restricciones (Hibernate)

@Normalized(form=): Valida que la secuencia de caracteres anotados se normalice de acuerdo con lo dado form.

@Range(min=, max=): Comprueba si el valor anotado se encuentra entre (inclusive) el mínimo y el máximo especificados

@SafeHtml(additionalTags=, additionalTagsWithAttributes=, baseURI=, whitelistType=): Comprueba si el valor anotado contiene fragmentos potencialmente maliciosos como <script/>.

@ScriptAssert(lang=, script=, alias=, reportOn=): Comprueba si la secuencia de comandos proporcionada se puede evaluar correctamente con el elemento anotado.

@UniqueElements: Comprueba que la colección solo contiene elementos únicos.

@URL(protocol=, host=, port=, regexp=, flags=): Comprueba si la secuencia de caracteres anotada es una URL válida según RFC2396.

Validaciones personalizadas

- Anotación personalizada para la validación:

```
import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import javax.validation.Constraint;
import javax.validation.Payload;

@Target({ ElementType.FIELD })
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy = NifValidator.class)
@Documented
public @interface NIF {
        String message() default "{validation.NIF.message}";
        Class<?>[] groups() default {};
        Class<? extends Payload>[] payload() default {};
}
```


Validaciones personalizadas

- Clase del validador

```
import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;
public class NifValidator implements ConstraintValidator<NIF, String>  {
        @Override
        public boolean isValid(String value, ConstraintValidatorContext context) {
            if(value == null) return true;
            value = value.toUpperCase();
            if(!value.matches("^\\d{1,8}[A-Z]$")) return false;
            return "TRWAGMYFPDXBNJZSQVHLCKE".charAt(Integer.parseInt(
                    value.substring(0, value.length() - 1)) % 23) == value.charAt(value.length() - 1);
        }
}
```

- En el fichero ValidationMessages.properties

```
validation.NIF.message=${validatedValue} no es un NIF válido.
```

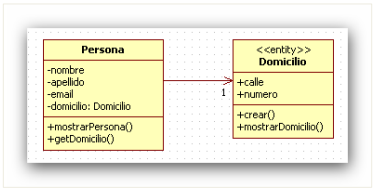
ASOCIACIONES CON HIBERNATE

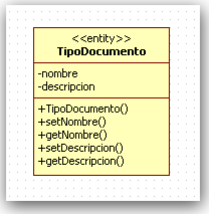
Asociaciones en Hibernate

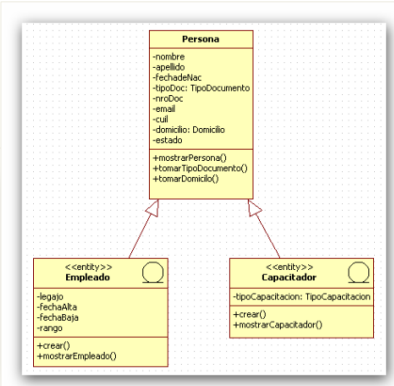
- Existen 2 temas importantes cuando empezamos a desarrollar en Hibernate:
  - Mapeo de asociaciones entre las clases de entidad.
  - Mapeo de colecciones
- Hasta ahora hemos visto el mapeo simple entre una clase y una entidad, pero la base de datos no contienen solo tablas aisladas, la tablas tienen 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







Transiciones en cascada del estado de entidad

- Qué se debe hacer con las entidades relacionadas cuando realizamos alguna acción con la entidad principal:
  - ¿Si borramos en la Principal? → ¿Tenemos que borrar en las clases dependiente? ¿Impedirlo si tiene dependientes?
- JPA permite propagar la transición de estado de una entidad padre a un hijo. Para este propósito, define varios tipos de comportamientos en cascada (javax.persistence.CascadeType):
  - ALL : cualquier operación.
  - PERSIST : la operación de persistencia de entidades.
  - MERGE : la operación de fusión de entidades.
  - REMOVE : la operación de eliminación de entidades.
  - REFRESH : la operación de actualización de la entidad.
  - DETACH : la operación de separación de entidades.
- Además, con org.hibernate.annotations.CascadeType, Hibernate define varios tipos de comportamientos adicionales ():
  - SAVE_UPDATE : conecta en cascada la operación saveOrUpdate de la entidad.
  - REPLICATE : conecta en cascada la operación de réplica de la entidad.
  - LOCK : conecta en cascada la operación de bloqueo de entidad.
- Con javax.persistence.CascadeType.ALL se propagará cualquier operación específica de Hibernate.
- Si una entidad secundaria está vinculado a la principal (no puede existir sin la principal), la asociación se puede mapear con orphanRemoval = true que desencadenará la eliminación de la fila en la tabla secundaria.

Cascada con anotaciones

- Parámetro cascade de la anotación:

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

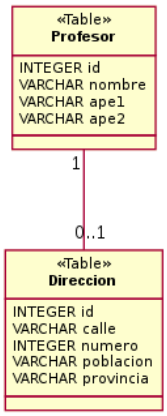
- Enumeración de tipo CascadeType:
  - ALL = {PERSIST, MERGE, REMOVE, REFRESH, DETACH}
  - DETACH
  - MERGE
  - PERSIST
  - REFRESH
  - REMOVE
- Acepta múltiples valores:

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


Uno a uno

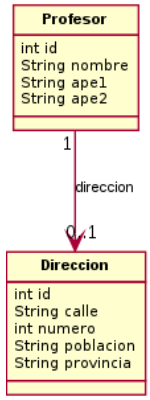
- La relación uno a uno en Hibernate consiste en que un objeto tenga una referencia a otro objeto de forma que al persistirse el primer objeto también se persista el segundo.
- Si en el Modelo Relacional encontramos PK-FK entre entidades, entonces deberemos crear una Asociación de 1 a muchos.
- Este tipo de relación se puede crear cuando "necesitemos" establecer una relación entre entidades que no tengan columnas comunes ni relacionadas.
- 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.

Uno a uno (Unidireccional)



- Cuando 1 objeto tiene una referencia a otro objeto, cuando hacemos persistir el primero, también se hace la 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) {
 :
```



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

```
<?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

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>
- 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.
- La clave primaria no puede ser autogenerada y debería estar marcada como clave ajena.

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>
```


Uno a uno (Unidireccional)

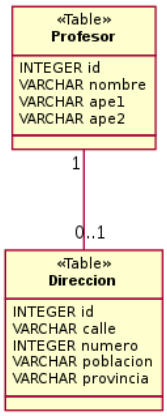
- Al igual que en los casos anteriores, cuando utilizamos anotaciones JPA deberemos modificar el código fuente de las clases Java y no usar los ficheros .hbm.xml.
- Sólo deberemos introducir las anotaciones relacionadas con one-to-one en la clase Profesor y en la propiedad de unión de ambas.

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

- @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

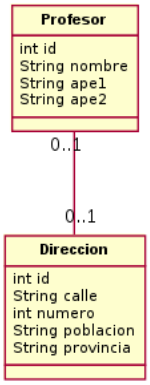
Uno a uno (Bidireccional)

- 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




```
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, St
 :
```

```
public class Direccion implements Serializable {
  private int id;
    private String calle;
    private int numero;
    private String poblacion;
    private String provincia;
    private Profesor profesor;
```



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>
```

```
<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"/>
    <one-to-one name="profesor" cascade="all" />

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

Direccion.hbm.xml

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

Uno a uno (Bidireccional)

- Sólo deberemos introducir las anotaciones relacionadas con one-to-one en las clases Profesor y Direccion en la propiedad de unión de ambas.

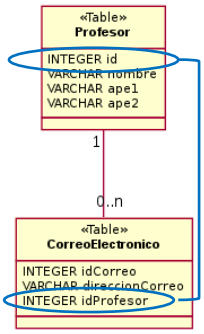
```
@Entity
public class Profesor implements Serializable {
        // …
        @OneToOne(cascade=CascadeType.ALL)
        @PrimaryKeyJoinColumn
        private Direccion direccion;
        // …
}

@Entity
public class Direccion implements Serializable {
        // …
        @OneToOne(cascade=CascadeType.ALL)
        @PrimaryKeyJoinColumn
        private Profesor profesor;
        // …
}
```


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
- Podemos distinguir 2 tipos de Asociaciones:
  - Unidireccional
    - El objeto “uno” contiene una colección de objetos “muchos”. Cuando se hace la persistencia del objeto “uno” también se hace la de los objetos “muchos”, pero no a la inversa.
  - Bidireccional
    - El objeto “uno” contiene una colección de objetos “muchos” y el objeto “muchos” tiene una referencia al objeto “uno”. Cuando se hace la persistencia del objeto “uno” o del objeto “muchos” también se hace la del otro.
- Habitualmente la lista de objetos usadas son colecciones (List, Set, etc) y dependiendo de la colección utilizada se designan Ordenados o Desordenados.

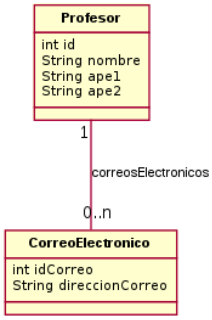
Uno a Muchos Desordenados

- En Java almacenamos la serie de objetos hijos mediante una colección del interface Set o cualquier otra colección que no implique orden de los objetos.



- En este tipo de relaciones existe una PK y una FK en las tablas relacionales:
  - Id(profesor) →PK
  - idProfesor(correo) → FK

```
public class Profesor implements Serializable  {
:
   private Set<CorreoElectronico> correosElectronicos;
:
public class CorreoElectronico implements Serializable {
:
  private Profesor profesor;
:
```



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

<key>:
Nombre de la columna de la BBDD que actúa como clave ajena FK en la relación

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

Uno a Muchos Desordenados

- Al estar utilizando 2 entidades necesitamos 2 archivos de persistencia .hbm.xml
  - Profesor.hbm.xml
  - **CorreoElectronico.hbm.xml**
- Si se establece una relación bidireccional, 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

Uno a Muchos Desordenados

- 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.
- Con @OneToMany se indica la relación uno a muchos en la clase contenedora:
  - mappedBy: Este atributo contendrá el Nombre de la PROPIEDAD JAVA de la clase HIJA que enlaza con el objeto PADRE, es decir, la referencia inversa.
  - cascade: Este atributo tiene el mismo significado que el del fichero de mapeo de Hibernate.

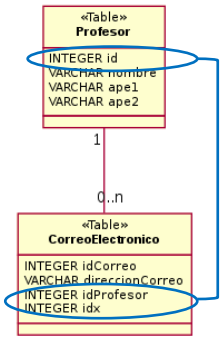
```
@OneToMany(mappedBy="profesor",cascade= CascadeType.ALL)
private Set<CorreoElectronico> correosElectronicos;
```

- En la clase contenida, de forma similar a las uno a uno, con la anotación @ManyToOne se establece la relación de muchos a uno y con @JoinColumn indicaremos el nombre de la columna que en la tabla hija contiene la clave ajena a la tabla padre.

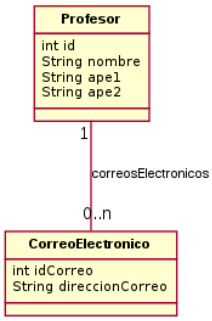
```
@ManyToOne
@JoinColumn(name="IdProfesor")
 private Profesor profesor;
```

Uno a Muchos Ordenados

- En Java almacenamos la serie de objetos hijos mediante una colección del interface List o cualquier otra colección que no implique orden de los objetos.



- En este tipo de relaciones existe una PK y una FK en las tablas relacionales:
  - Id(profesor) → PK
  - IdProfesor(correo) → FK
  - Idx → Columna adicional para indicar orden de los hijos



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

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.

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

<key>:
Nombre de la columna de la BBDD que actúa como clave ajena FK en la relación

<list-index>
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.

Uno a Muchos Ordenados

- Al estar utilizando 2 entidades necesitamos 2 archivos de persistencia .hbm.xml
  - Profesor.hbm.xml
  - **CorreoElectronico.hbm.xml**
- Si se establece una relación bidireccional, 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" />
    <property name="orden" column="Idx" />
    <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

Uno a Muchos Ordenados

- 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.
- Con @OneToMany se indica la relación uno a muchos en la clase contenedora.
- Para conservar el orden de los elementos de la colección, existen dos posibilidades:
  - @OrderBy: la colección se ordena al recuperarla utilizando una propiedad de entidad secundaria
  - @OrderColumn: la colección utiliza una columna de orden dedicada en la tabla de enlaces de la colección

```
@OneToMany(mappedBy="profesor", cascade= CascadeType.ALL)
@OrderColumn(name = "idx")
private List<CorreoElectronico> correosElectronicos;
```

- En la clase contenida, de forma similar a las uno a uno, con la anotación @ManyToOne se establece la relación de muchos a uno y con @JoinColumn indicaremos el nombre de la columna que en la tabla hija contiene la clave ajena a la tabla padre.

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

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 tabla (Renacida) que sirve de conexión entre ambas dos. En una renacida pura, la tabla solo contiene las claves migradas (sin datos adicionales) y no requiere una clase entidad especifica. Si no es una renacida pura hay que descomponer la relación muchos a muchos en dos relaciones uno a muchos.
- 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

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 o List)


```
public class Profesor implements Serializable {
    private int id;
    private String nombre;
    private String ape1;
    private String ape2;
    private Set<Asignatura> asignaturas=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;
    }
}
```

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 de un extremo

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="asignaturas"  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

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 de un extremo

Asignatura.hbm.xml

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

  <set name= "profesores"  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.

Muchos a Muchos

- 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.
- Con @ManyToMany se indica la relación muchos a muchos en cada clase contenedora. La anotación @JoinTable se utiliza para especificar la tabla de vínculos entre otras dos tablas de la base de datos.

```
public class Profesor implements Serializable  {
        :
        @ManyToMany(cascade= CascadeType.ALL)
        @JoinTable(name = "ProfesorAsignatura",
                        joinColumns = @JoinColumn(name = "idProfesor"),
                        inverseJoinColumns = @JoinColumn(name = "idAsignatura"))
        private Set<Asignatura> asignaturas;

public class Asignatura implements Serializable {
        :
        @ManyToMany(mappedBy = "asignaturas", cascade= CascadeType.ALL)
        @JoinTable(name = "ProfesorAsignatura",
                        joinColumns = @JoinColumn(name = "idAsignatura"),
                        inverseJoinColumns = @JoinColumn(name = "idProfesor"))
        private Set<Profesor> profesores;
```


Bidireccionalidad

- Siempre que se forme una asociación bidireccional, el desarrollador de la aplicación debe asegurarse de que ambos lados estén sincronizados en todo momento.
- Se recomienda añadir métodos de utilidad que sincronicen los dos extremos cada vez que se añade o elimina un elemento hijo.

```
public CorreoElectronico addCorreoElectronico(CorreoElectronico item) {
    getCorreosElectronicos().add(item);
    item.setProfesor(this);
    return item;
}
public CorreoElectronico removeCorreoElectronico(CorreoElectronico item) {
    getCorreosElectronicos().remove(item);
    item.setProfesor(null);
    return item;
}
```

Estrategias de recuperación

- La recuperación es el proceso de obtener datos de la base de datos y ponerlos a disposición de la aplicación. Ajustar cuando y cómo hacer la recuperación es uno de los factores más importantes para optimizar el rendimiento.
  - Obtener demasiados datos, en términos de ancho (valores/columnas) o profundidad (resultados/filas), agrega una sobrecarga innecesaria en términos de comunicación JDBC y procesamiento ResultSet.
  - Obtener muy pocos datos puede hacer que sea necesario realizar múltiples consultas a la base de datos.
- Se dispone de dos posibles estrategias:
  - EAGER (entusiasta o inmediata): se recuperan todos los datos cuando se recupera al propietario.
  - LAZY (perezosa o diferida): espera a que se produzca el primer acceso para recuperar los datos.
- Las estrategias se pueden establecer para asociaciones o campos muy largos.

Estrategias de recuperación

EAGER

- Ventajas:
  - Una única operación de lectura
  - No requiere mantener la sesión abierta
- Desventajas:
  - Puede obtener demasiados datos que luego no se usen con el correspondiente desperdicio de comunicaciones, memoria y procesamiento.

```
@OneToMany(fetch=FetchType.EAGER)
private List<CorreoElectronico> correosElectronicos;
```

```
<one-to-many lazy="false" class="CorreoElectronico" />
```

LAZY

- En la primera petición, se obtendrá un proxy que se encargará de hacer las consultas a la base de datos cuando realmente sea necesario.
- Ventajas:
  - Obtiene exclusivamente los datos necesarios
  - Ajusta los consumos de comunicaciones, memoria y procesamiento.
- Desventajas:
  - Requiere mantener la sesión abierta
  - Múltiples operaciones de lectura
  - Limita la opciones de optimización

```
@OneToMany(fetch=FetchType.LAZY)
private List<CorreoElectronico> correosElectronicos;
```

```
<one-to-many lazy="true" class="CorreoElectronico" />
```

Estrategias de extracción

- El fetch, o estrategia de extracción, define el SQL que se va a generar para obtener los datos de la base de datos, en una o varias consultas.
- Se dispone de tres posibles estrategias:
  - JOIN: Inherente al EAGER, realiza una única consulta mediante el uso de una combinación externa (left outer join) de SQL.
  - SUBSELECT: Realiza una consulta separada para cargar los datos asociados según la restricción de SQL utilizada para cargar al propietario.
  - SELECT: Realiza consultas separadas para cargar los datos. Esta es la estrategia generalmente denominada N + 1.

```
@OneToMany(fetch=FetchType.LAZY)
@Fetch(FetchMode.SUBSELECT)
private List<CorreoElectronico> correosElectronicos;
<one-to-many lazy="true" class="CorreoElectronico" fetch="join" />
```

- La anotación @BatchSize realiza consultas separadas para cargar una serie de elementos de datos relacionados utilizando una restricción IN como parte de la cláusula WHERE de SQL basada en un tamaño de lote.

@NotFound

- Cuando se trata de asociaciones que no están impuestas por una clave externa, es posible encontrar inconsistencias si el registro secundario no puede hacer referencia a una entidad principal.
- De forma predeterminada, Hibernate generará una excepción cada vez que una asociación secundaria haga referencia a un registro principal que no existente.
- Para ignorar las referencias a entidades principales no existentes (y simplemente asignar un null), aunque no se recomienda en la mayoría de los casos, es posible usar la anotación de NotFound con valor NotFoundAction.IGNORE.

```
@ManyToOne
@NotFound ( action = NotFoundAction.IGNORE )
private Profesor profesor;
```

- La anotación @NotFound(action = NotFoundAction.IGNORE) hace que las asociaciones @ManyToOne y @OneToOne siempre se obtengan como EAGER incluso si está configurada como FetchType.LAZY.

Estrategias de combinación

- La anotación @PrimaryKeyJoinColumn se utiliza para especificar que la columna de clave principal de la entidad anotada actualmente es también una clave externa para alguna otra entidad.
- La anotación @JoinColumn se usa para especificar la columna FOREIGN KEY que se usa al unirse a una asociación de entidad o una colección insertable.
- La anotación @JoinFormula se utiliza para personalizar la unión entre una clave externa secundaria y una clave principal de la fila principal.

```
@ManyToOne
@JoinFormula( "REGEXP_REPLACE(phoneNumber, '\\+(\\d+)-.*', '\\1')::int" )
private Country country;
```

- La anotación @JoinColumnOrFormula se utiliza para personalizar la unión entre una clave externa secundaria y una clave principal de la fila principal cuando necesitamos tener en cuenta un valor de columna y un @JoinFormula.

```
@ManyToOne
@JoinColumnOrFormula(column = @JoinColumn(name = "language", referencedColumnName = "primaryLanguage")
@JoinColumnOrFormula( formula = @JoinFormula(value = "true", referencedColumnName = "is_default" ))
private Country country;
```


Colecciones

- Hibernate usa sus propias implementaciones de colección que están enriquecidas con semánticas de carga diferida, almacenamiento en caché o detección de cambios de estado. Es importante que las colecciones se definan utilizando la interfaz adecuada en lugar de una implementación específica.
- Para colecciones de tipos de valor, JPA 2.0 define la anotación @ElementCollection. El ciclo de vida de la colección de tipo de valor está completamente controlado por su entidad propietaria.

```
@ElementCollection
@CollectionTable(name = "ProfesorTelefonos", joinColumns = @JoinColumn(name = "idProfesor"))
private List<Telefono> telefonos = new ArrayList<>();

@Embeddable
public static class Telefono {
    private String type;
    @Column(name = "`number`")
    private String number;
    :
}
```

- Los conjuntos, implementaciones de la interfaz Set, son colecciones que no permiten entradas duplicadas.

Matrices

- Cuando se habla de matrices, es importante comprender la distinción entre los tipos de matrices SQL y las matrices Java que se asignan como parte del modelo de dominio de la aplicación.
- No todas las bases de datos implementan el tipo ARRAY SQL-99 y, por esta razón, Hibernate no admite tipos matriz de bases de datos nativos.
- Hibernate admite el mapeo de matrices en el modelo de dominio de Java, conceptualmente lo mismo que mapear una lista. Es imposible la carga diferida de matrices de entidades, por lo que se recomienda mapear a una colección en lugar de una matriz.
- Si se desea mapear matrices como String[] o int[] a tipos de matrices específicos de la base de datos como PostgreSQL integer[] o text[], se debe escribir un tipo de hibernación personalizado.

Colecciones como tipo de valor

- Las colecciones no marcadas como tales (ElementCollection, OneToMany o ManyToMany) requieren un tipo personalizado de Hibernate y los elementos de la colección deben almacenarse en una sola columna de la base de datos.

```
@Type(type = "comma_delimited_strings")
private List<String> phones = new ArrayList<>();

public class CommaDelimitedStringsType extends AbstractSingleColumnStandardBasicType<List> {
   public CommaDelimitedStringsType() {
      super(VarcharTypeDescriptor.INSTANCE, new CommaDelimitedStringsJavaTypeDescriptor());
   }
   @Override
   public String getName() {
      return "comma_delimited_strings";
   }
}
```

Colecciones como tipo de valor

```
public class CommaDelimitedStringsJavaTypeDescriptor extends AbstractTypeDescriptor<List> {
    public static final String DELIMITER = ",";
    public CommaDelimitedStringsJavaTypeDescriptor() {
        super(List.class, new MutableMutabilityPlan<List>() {
                @Override
                protected List deepCopyNotNull(List value) { return new ArrayList( value ); }
            }
        );
    }
    @Override
    public String toString(List value) {
        return ( (List<String>) value ).stream().collect( Collectors.joining( DELIMITER ) );
    }
    @Override
    public List fromString(String string) {
        List<String> values = new ArrayList<>();
        Collections.addAll( values, string.split( DELIMITER ) );
        return values;
    }
    @Override
    public <X> X unwrap(List value, Class<X> type, WrapperOptions options) {
        return (X) toString( value );
    }
    @Override
    public <X> List wrap(X value, WrapperOptions options) {
        return fromString( (String) value );
    }
}
```


Mapeo de Herencia

- Hibernate soporta las tres estrategias básicas de mapeo de herencia:
  - tabla por jerarquía de clases (subclass)
  - tabla por subclases (joined-subclass)
  - tabla por clase concreta (union-subclass)
- 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>
```

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>
```


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>
```

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 { … }
```


@MappedSuperclass

- La anotación @MappedSuperclass que la herencia solo se implemente en el modelo de dominio sin reflejarla en el esquema de la base de datos.

```
@MappedSuperclass
public abstract class Account { … }

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

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

- Aunque similar a “Tabla por clase concreta”, debido a que el modelo de herencia con @MappedSuperclass no se refleja a nivel de la base de datos, no es posible utilizar consultas polimórficas que hagan referencia al @MappedSuperclass cuando se obtienen objetos persistentes por su clase base.

@Any: Asociación polimórfica

- El mapeo @Any es útil para emular una asociación unidireccional @ManyToOne cuando puede haber múltiples entidades objetivo.
- Debido a que el mapeo @Any define una asociación polimórfica a clases de varias tablas, este tipo de asociación requiere la columna FK que proporcione el identificador principal asociado e información de metadatos para el tipo de entidad asociado.
- La anotación @Any describe la columna que contiene la información de metadatos. Para vincular el valor de la información de metadatos y un tipo de entidad real, se utilizan las anotaciones @AnyDef y @AnyDefs. El atributo metaType permite especificar un tipo personalizado que asigna valores de columna de base de datos a clases persistentes que tienen propiedades de identificador del tipo especificado por idType. Se debe especificar la asignación de los valores metaType a los nombres de clase.

@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;
```

CONSULTAS CON HQL/JPQL

Lenguajes de consulta

- Hibernate Query Language (HQL) y Java Persistence Query Language (JPQL) son lenguajes de consulta enfocados en modelos de objetos de naturaleza similar a SQL.
- JPQL es un subconjunto de HQL. Una consulta JPQL es siempre una consulta HQL válida, pero no lo contrario.
- HQL es muy parecido al SQL estándar, sigue las mismas convenciones sintácticas, con la diferencia fundamental: esta orientado a objetos
  - Se usan clases y atributos en lugar de tablas y columnas
- Al ser orientado a objetos se puede usar herencia, polimorfismo y asociaciones:
  - Trabajamos realmente sobre las clase Java y es el Hibernate quien redacta las consultas SQL sobre las tablas mapeadas.

Lenguajes de consulta

- Las consultas se redactan en formato cadena por lo que no se puede realizar la comprobación de tipos en tiempo de compilación, los errores son siempre en tiempo de ejecución. Las consultas por criterios (API Criteria) ofrecen un enfoque con seguridad de tipos para las consultas.
- Al trabajar sobre Java, debemos respetar la diferencia sintáctica entre mayúsculas y minúsculas:
  - Las clausulas como SELECT, FROM, etc, no distingue entre mayúsculas y minúsculas.
  - Los nombres (clases, atributos, propiedades, parámetros, …) si que diferencian entre mayúsculas y minúsculas
- HQL está diseñado exclusivamente para realizar la recuperación y modificación de datos (DQL y DML).

Características

- 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

- Una consulta HQL debe ser preparada antes de ser ejecutada.
- La preparación implica varios pasos:
  1. Crear la consulta (objeto), con cualquier restricción arbitraria o proyección de los datos que se desean recuperar.
  2. Opcionalmente: 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 (materializar)
  4. Recuperar y tratar 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).

Crear la consulta

- Hibernate tiene el interfaz Query (org.hibernate.query) para definir los objetos consulta que nos dan 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 través de la unidad de trabajo:

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

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

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.

```
<query name="TodosLosProfesores"><![CDATA[
    SELECT p FROM Profesor p
    ]]>
```

- En JPA se realizaría mediante la anotación @NamedQuery:

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

- Para crear una nueva instancia de Hibernate Query, referenciada a una consulta con nombre:

```
Query query = session.getNamedQuery("TodosLosProfesores");
Query query = entityManager.createNamedQuery("Profesor.findAll");
```


Establecer restricciones

- La interfaz Query permite definir la ejecución de la consulta. Por ejemplo, es posible que queramos especificar un tiempo de espera de ejecución o controlar el almacenamiento en caché.

```
query.setHint( "javax.persistence.query.timeout", 2000 )
    .setFlushMode( FlushModeType.COMMIT );
```

- Para paginar los resultados podemos especificar el número máximo de filas a recuperar y saltar un determinado número de filas:

```
query.setMaxResults(filasPorPagina)
    .setFirstResult(pagina * filasPorPagina);
```

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“) }
)
```


Parametrización de consultas

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

```
String queryString = "delete from Item i where i.id = " + leido;
String leido = "1 or 1=1";  // se esperaba "1"
```

- El lenguaje de consultas permite el uso de parámetros con nombre y parámetros posicionales, aunque con nombre facilitan la legibilidad.
- Los parámetros con nombre se declaran usando un signo de dos punto seguido de un nombre (:id) y se vinculan solo con el nombre:

```
Query query = entityManager.createQuery("from Person p where p.nombre like :name" ).setParameter(
"name", "J%" );
```

- Los parámetros posicionales se declaran usando un signo de interrogación seguido de un ordinal (?1, ?2), los ordinales comienzan en 1, y se vinculan con el número del ordinal:

```
Query query = session.createQuery("delete from Item i where i.id = ?1").setParameter( 1, 666 );
```

- Los parámetros con nombre y posicionales pueden aparecer varias veces en una consulta.
- Por defecto setParameter() infiere el tipo del parámetro pero se puede explicitar:

```
query.setParameter( "timestamp", timestamp, TemporalType.TIMESTAMP);
```

Materializar la consulta

- Una vez que hayamos creado y preparado una consulta, ya estamos listos para para ejecutarla contra la base de datos y recuperar un conjunto de resultados.
- JPA ofrece 3 métodos para recuperar un conjunto de resultados.
  - Query.getResultList() : ejecuta la consulta de selección y devuelve la lista de resultados.
    ```
    List<Profesor> profesores = query.getResultList();
    ```
  - Query.getResultStream(): ejecuta la consulta de selección y devuelve un Stream sobre los resultados.
    ```
    List<Profesor> profesores = query.getResultStream().skip( 30 ).limit( 10 ).collect( Collectors.toList() );
    ```
  - Query.getSingleResult(): ejecuta la consulta de selección y devuelve un único resultado o lanza una excepción si no hay o hay más de un resultado.
    ```
    Profesor profesor = query.setParameter( "id", 1 ).getSingleResult();
    ```
- Hibernate ofrece 4 métodos adicionales (heredados) precursores a los anteriores: list(), stream(), uniqueResult() y uniqueResultOptional().

Materializar la consulta

- Para grandes conjuntos de resultados, scroll() permite aprovechar un cursor del lado del servidor (JDBC scrollable ResultSet):

```
try ( ScrollableResults scrollableResults = query.scroll()) {
    while(scrollableResults.next()) {
        Profesor profesor = (Profesor) scrollableResults.get()[0];
        // …
    }
}
```

- Dado que mantiene abierto un ResultSet JDBC, se debe indicar cuándo se ha terminado con el ScrollableResults llamando a su método close() (como hereda de java.io.Closeable funciona en los bloques try-with-resources).
- Si se deja sin cerrar, Hibernate cerrará automáticamente los recursos subyacentes utilizados internamente por el ScrollableResults cuando finalice la transacción actual (ya sea para confirmar o deshacer). Sin embargo, es una buena práctica cerrar ScrollableResults explícitamente.

Tipos de declaraciones

- Tanto HQL como JPQL permiten ejecutar sentencias SELECT, UPDATE y DELETE. Además, HQL permite declaraciones INSERT, de una forma similar al INSERT FROM SELECT de SQL.
- La consulta mas habitual será la de recuperación de datos con la versión propia de la instrucción SELECT:

  [select_clause]
  from_clause
  [where_clause]
  [groupby_clause]
  [having_clause]
  [orderby_clause]

Cláusula FROM

- La cláusula FROM es responsable de definir el alcance de los tipos de modelos de objetos disponibles para el resto de la consulta. Es la única obligatoria (excepto en JPQL que también requiere la select_clause).
- También se encarga de definir todas las "variables de identificación" (alias) disponibles para el resto de la consulta (en la mayoría de los casos son opcionales aunque es una buena práctica declararlas).

```
queryStr = "from Profesor";
queryStr = "from Profesor p";
```

- Puede contener múltiples referencias de entidad como combinaciones JOIN del estilo CROSS, INNER o LEFT OUTER.

```
queryStr = "from Profesor p, Asignatura a"; // CROSS JOIN
queryStr = "from Profesor p join p.correosElectronicos e"; // INNER JOIN
queryStr = "from Profesor p left join p.correosElectronicos e"; // LEFT OUTER JOIN
```

- Las condiciones del JOIN se infieren de las asociaciones de las entidades y se pueden completar con clausulas ON (generan un ON en SQL, no un WHERE). WITH es un sinónimo de ON en HQL.

```
queryStr = "from Profesor p join p.correosElectronicos e on e.tipo = :tipo"
queryStr = "from Profesor p join p.correosElectronicos e with e.tipo = :tipo"
```

Literales

- Los literales de cadena se incluyen entre comillas simples (para escapar una comilla simple dentro de un literal de cadena se usan dos comillas simples).

```
queryStr = "from Customer c where c.name = 'Joe''s Bar'";
```

- Los literales numéricos se permiten la forma de notación científica y la tipificación con sufijos del Java (L para long, D para double, F para float). No distingue entre mayúsculas y minúsculas.
- Los booleanos son TRUE y FALSE, tampoco distingue entre mayúsculas y minúsculas.
- Se puede hacer referencia a valores de las enumeraciones como literales usando el nombre de clase de enumeración completamente calificado.
- Los literales de fecha / hora se pueden especificar utilizando la sintaxis de escape de JDBC, solo funcionan si el controlador JDBC subyacente los admite:
  - {d 'yyyy-mm-dd'} para las fechas
  - {t 'hh:mm:ss'} por tiempos
  - {ts 'yyyy-mm-dd hh:mm:ss[.millis]'} (millis opcional) para las marcas temporales.

Expresiones

- Las expresiones son referencias que se resuelven en valores básicos o de tupla y se utilizan en las clausulas SELECT, JOIN, WHERE, GROUP, HAVING y ORDER.
- En las expresiones pueden intervenir atributos o propiedades de las entidades, parámetros, literales, operadores y funciones.
- La expresión acepta operadores:
  - Aritméticos: + - * /
  - Concatenación: | |
  - De comparación: = != <> > >= < <= BETWEEN LIKE IN
  - Lógicos: AND OR NOT
  - Comprobación: IS [NOT] NULL, EXISTS, IS [NOT] EMPTY, [NOT] MEMBER [OF]
  - Condicionales:
    - CASE {operand} WHEN {test_value} THEN {match_result} ELSE {miss_result} END
    - CASE [WHEN {test_conditional} THEN {match_result}]* ELSE {miss_result} END
    - NULLIF (primero si son distintos, si no, NULL), COALESCE (el primer operando no nulo)

Funciones de agregación

COUNT (incluidos los calificadores distintos / todos): El tipo de resultado es siempre Long.

AVG: El tipo de resultado es siempre Double.

MIN: El tipo de resultado es el mismo que el tipo de argumento.

MAX: El tipo de resultado es el mismo que el tipo de argumento.

SUM: El tipo de resultado depende del tipo de valores que se suman. Para valores enteros (distintos de BigInteger), el tipo de resultado es Long. Para valores de coma flotante (distintos de BigDecimal), el tipo de resultado es Double.

Funciones estandarizadas de JPQL

- concat: concatenación de cadenas (longitud variable: 2 o más valores).
- substring: Extrae una parte de un valor de cadena. El segundo argumento denota la posición inicial, donde 1 es el primer carácter de la cadena. El tercer argumento (opcional) denota la longitud.
- upper: En mayúsculas la cadena especificada.
- lower: En minúsculas la cadena especificada.
- trim: Sigue la semántica de la función de recorte de SQL.
- length: La longitud de una cadena.
- locate: Localiza una cadena dentro de otra cadena. El tercer argumento (opcional) se utiliza para indicar una posición desde la que empezar a buscar.
- abs: Calcula el valor absoluto.
- mod: Calcula el resto de división entera.
- sqrts: Calcula la raíz cuadrada.
- type: Tipo de entidad como expresión. Ej.: type(p) = ClaseDerivada
- current_date: Devuelve la fecha actual de la base de datos (pseudo variable, sin paréntesis).
- current_timestamp: Devuelve la hora actual de la base de datos (pseudo variable, sin paréntesis).

Funciones HQL (no son portables a otros proveedores de JPA)

- bit_length: Devuelve la longitud de los datos binarios.
- cast: Realiza una conversión SQL a un tipo de mapeo Hibernate. Ej.: cast(c.duration as string)
- extract: Realiza una extracción SQL en valores de fecha y hora. Una extracción extrae partes de la fecha y hora. Ej.: extract(YEAR from c.timestamp)
- year: Forma abreviada de extracto para extraer el año.
- month: Forma de extracto abreviado para extraer el mes.
- day: Forma de extracto abreviado para extraer el día.
- hour: Forma de extracto abreviado para extraer la hora.
- minute: Forma abreviada de extracto para extraer el minuto.
- second: Forma de extracto abreviada para extraer el segundo.
- str: Forma abreviada de conversión a cadena.

Expresiones relacionadas con colecciones

- size: Calcula el tamaño de una colección (equivale a una subconsulta). Ej: size(p.phones) = 2
- maxelement: Disponible para uso en colecciones de tipo básico. Se refiere al valor máximo que se determina aplicando la función de agregación max del SQL.
- maxindex: Disponible para su uso en colecciones indexadas. Se refiere al índice máximo (clave/posición) según se determina aplicando la función de agregación max del SQL.
- minelement: Disponible para uso en colecciones de tipo básico. Se refiere al valor mínimo que se determina aplicando la función de agregación min del SQL.
- minindex: Disponible para su uso en colecciones indexadas. Se refiere al índice mínimo (clave/posición) según se determina aplicando la función de agregación min del SQL.
- elements: Se utiliza para referirse a los elementos de una colección en su conjunto. Solo permitido en la cláusula where. A menudo se utiliza en conjunción con restricciones ALL, ANY o SOME. Ej: current_date() > all elements(p.repairTimestamps)
- indices: Similar a elements excepto que la expresión se refiere a los índices de colecciones (claves/posiciones) como un todo. Ej: 1 in indices(p.phones)

Expresiones relacionadas con colecciones

- value: Se refiere al valor de la colección (igual que no especificar un calificador, util para mostrar explícitamente la intención). Válido para cualquier tipo de colección.
- index: De acuerdo con las reglas de HQL, esto es válido tanto para mapas como para listas que especifican una anotación javax.persistence.OrderColumn para referirse a la clave del mapa o la posición en la lista (también conocido como el valor de OrderColumn). Sin embargo, JPQL reserva index() para su uso en listas y agrega key() para su uso en mapas.
- key: Válido solo para mapas. Se refiere a la clave del mapa. Si la clave es en sí misma una entidad, se puede seguir navegando.
- entry: Solo válido para mapas. Se refiere a la tupla lógica java.util.Map.Entry del mapa (la combinación de su clave y valor) y solo es aplicable como ruta de terminal en la cláusula SELECT (devolver la tupla).
- El operador de índice puede hacer referencia a elementos de colecciones indexadas (matrices, listas y mapas).
  - indexed lists: p.phones[0]
  - maps: p.addresses['HOME']

Cláusula SELECT

- La cláusula SELECT identifica qué entidad/proyección o valores devolver como resultado de la consulta. Las consultas HQL y JPQL son inherentemente polimórficas: devolverá la entidad o uno de sus herederos.

```
queryStr = "select a from Asignatura a";
queryStr = "select a.id, a.name from Asignatura a";
```

- Hay un tipo de expresión particular, que solo es válida en la cláusula SELECT, llamada “instanciación dinámica“ (HQL) o “expresión de constructor” (JPQL): crea un nuevo objeto (comúnmente denominado proyección).
  - queryStr = "select new com.example.Elemento(a.id, a.name) from Asignatura a";
- La clase de proyección debe estar completamente calificada en la consulta de entidad (los genéricos sin <>) y debe definir un constructor coincidente. La clase proyección no necesita estar mapeada, puede ser una clase DTO. Si representa una entidad, las instancias resultantes se devuelven en el estado Transient (¡no administrado!). Devolverá una lista del tipo de la proyección.

Cláusula SELECT

- En caso de definir una lista de valores, se encapsularan en un array de objetos, un array por fila de resultados con tantos elementos como “columnas” hayamos definido en la cláusula SELECT. El resultado de la consulta será un List<Object[]>.
- La lista de valores pueden ser entidades, atributos o propiedades de las entidades, parámetros o expresiones y pueden estar asociadas a una variable de identificación (alias) para poder hacer referencia en la clausula ORDER BY.

```
queryStr = "select a.id, a.name, a.precio * a.iva as total from Asignatura a";
```

- Se puede usar DISTINCT en la cláusula SELECT pero para JPQL y HQL, DISTINCT tiene dos significados:
  - Se puede pasar a la base de datos para eliminar los duplicados al obtener el conjunto de resultados.

```
queryStr = "select distinct a.name from Asignatura a"; // SQL: SELECT DISTINCT NAME FROM ASIGNATURAS
```

  - Se puede usar para filtrar el conjunto de resultados de las mismas referencias de entidad principal duplicadas cuando se ha realizado un left join de una colección secundaria.

```
queryStr = " select distinct p fetch from Profesor p left join p.correosElectronicos e on e.tipo = :tipo";
```


Filtrado y agrupamiento

- La cláusula WHERE restringe los resultados devueltos de una consulta de selección y limita el alcance de las consultas de actualización y eliminación. La cláusula WHERE solo cuenta con una expresión condicional.
- La cláusula GROUP BY permite generar resultados agregados para varios grupos de valores. La cláusula GROUP BY cuenta la lista de valores de agrupamiento.
- La cláusula HAVING restringe los grupos, opera sobre los valores agregados. La cláusula HAVING solo cuenta con una expresión condicional.

```
queryStr = "select a.nombre , p.tipo, count( p ) as total " +
    "from Asignatura a join a.alumnos p " +
    "where a.modalidad = com.example.Modalidades.PRESENCIAL " +
    "group by a.nombre, p.tipo " +
    "having count(p) > 0 ";
```

Ordenación

- La cláusula ORDER BY se utiliza para especificar los valores que se utilizarán para ordenar el resultado (JPQL obliga a que aparezcan en la cláusula SELECT). Los tipos de expresiones que se consideran válidas como parte de la cláusula ORDER BY incluyen:
  - campos, atributos, propiedades, parámetros
  - expresiones escalares como operaciones aritméticas, funciones, etc.
  - variables de identificación (declarada en la cláusula select)
- Las expresiones individuales en el orden pueden calificarse con ASC(ascendente) o DESC(descendente) para indicar la dirección de orden deseada. Los valores nulos se pueden colocar al principio o al final del conjunto ordenado usando la cláusulas NULLS FIRST o NULLS LAST respectivamente.

```
queryStr = "select a.nombre , p.tipo, count( p ) as total " +
      "from Asignatura a join a.alumnos p " +
      "group by a.nombre, p.tipo " +
      "order by a.nombre NULLS LAST, total DESC"
```

- La ordenación es costosa en términos de rendimiento, no debe solicitarse datos ordenados salvo que sea necesario.

Optimización de consultas

- 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, como las redactamos y como optimizarlas.
- Desde Hibernate, inicialmente, no podemos hacerlo ya que es el propio Hibernate el que se encarga de generar y ejecutar las consultas SQL por nosotros.
- Hay 2 situaciones que debemos atender:
  - Lazy Load: "n+1" SELECTs
  - Consultas nativas

Optimización: "n+1" SELECTs

- Es uno de los mayores problemas del ORM.
- Este problema consiste en que al lanzar un consulta redactada en HQL/JPQL que accede a sus dependencias y devuelve 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 mas la que recupera al profesor.
- Suelen aparecer cuando estamos tratando Asociaciones:
  - 1 a N
  - N a M

Optimización: "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_, direccic
Hibernate: select direccion0_.Id as Id1_0_0_, direccion0_.calle as calle2_0_0_, direccic
Hibernate: select direccion0_.Id as Id1_0_0_, direccion0_.calle as calle2_0_0_, direccic
Hibernate: select direccion0_.Id as Id1_0_0_, direccion0_.calle as calle2_0_0_, direccic
Hibernate: select direccion0_.Id as Id1_0_0_, direccion0_.calle as calle2_0_0_, direccic
```

Optimización: "n+1" SELECTs

- La primera consulta se lanza cuando se recuperan los profesores, el resto de las consultas se lanzan cuando se accede a las dependencias.
- La solución pasa por la precarga de los datos en la primera consulta, como el lenguaje SQL ESTANDAR no permite consultas jerárquicas, habrá que realizarlo mediante un JOIN con FECTH para forzar la carga de los datos.

```
queryStr = "SELECT p FROM Profesor p LEFT JOIN FETCH p.correosElectronicos";
```

- Al ser un JOIN, 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 de resultados desde la consulta se aplica el distinct:

```
queryStr = "SELECT DISTINCT p FROM Profesor p LEFT JOIN FETCH p.correosElectronicos";
```


Optimización: Consultas Nativas

- Otra forma de optimizar las consultas es realizando Consultas Nativas (SQL) 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) e implementar las optimizaciones del motor de base de datos.
- La desventaja de utilizar consultas nativas es que están vinculadas a un determinado motor de base de datos por lo que impide la portabilidad.
- La ejecución de consultas SQL nativas se controla por medio de la interfaz SQLQuery, la cual se obtiene llamando a Session.createSQLQuery().
  - Query sqlQuery = Session.createSQLQuery("SELECT * FROM PROFESOR");
- Este tipo de consulta puede devolver 2 tipos de elementos:
  - Valores CRUDOS/escalares básicos
  - Objetos de Entidades Mapeadas en Hibernate

Optimización: Consultas Nativas

- Valores CRUDOS/escalares básicos
  - Son valores independientes y se devolverán en una lista de arrays de objetos (List<Object[]>) con valores escalares para cada columna en la tabla

```
List<Object[]> rslt = Session.createSQLQuery("SELECT *  FROM PROFESOR").list();
```

  - Podemos especificar las columnas de forma predeterminada o indicar el tipo de Hibernate a devolver:

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

- Objetos de Entidades Mapeadas en Hibernate
  - La otra forma de trabajar con las consultas nativas es forzándolas a que devuelvan los datos como objetos de Entidades mapeadas en Hibernate. Utilizaremos el método addEntity para conseguirlo:

```
List<Profesor> rslt = Session.createSQLQuery("SELECT * FROM PROFESOR")
        .addEntity(Profesor.class).list();
```


Estrategias de 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: Centrada en los casos de uso.
      - HQL / JPQL
      - Perfiles de búsqueda
      - Grafos de entidades

@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;
```

Definición Dinámica: Perfiles de búsqueda

- Se pueden predefinir diferentes perfiles para sustituir dinámicamente las definiciones estáticas en todas las consultas de la session.

```
@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:
  - em.unwrap(Session.class).enableFetchProfile("region.all");

Definición Dinámica: Grafos de entidades

- Se pueden predefinir 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();
```

DML

- El lenguaje de consulta también permite realizar operaciones DML.
- 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();
```

- Se debe tener precaución al ejecutar operaciones de actualización o eliminación masivas porque pueden provocar inconsistencias entre la base de datos y las entidades en el contexto de persistencia activo. En general, las operaciones de actualización y eliminación masivas solo deben realizarse dentro de una transacción en un nuevo contexto de persistencia o antes de obtener o acceder a entidades cuyo estado podría verse afectado por tales operaciones.

Procesamiento por lotes

- Inserciones masivas de datos son operaciones que NO deberían ser realizadas a través de Hibernate.
- El siguiente ejemplo muestra un anti-patrón para inserciones por lotes: 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();
```

- Hibernate mantiene en caché todas las instancias recién insertadas a nivel de sesión por lo que este código podría generar una excepción OutOfMemoryException. Las transacciones de larga duración pueden agotar un grupo de conexiones, por lo que otras transacciones no tienen la oportunidad de continuar. El procesamiento por lotes de JDBC no está habilitado de forma predeterminada, por lo que cada instrucción de inserción requiere un viaje de ida y vuelta a la base de datos.

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
- JDBC ofrece soporte para agrupar sentencias SQL juntas que se pueden representar como un único PreparedStatement. En cuanto a la implementación, esto generalmente significa que los controladores enviarán la operación por lotes al servidor en una llamada, lo que puede ahorrar en llamadas de red a la base de datos. Hibernate puede aprovechar el procesamiento por lotes de JDBC.

Procesamiento por lotes

- La primera recomendación para un procesamiento masivo de filas (batch processing), es habilitar el procesamiento por lotes de JDBC configurando el batch_size.
- Deberemos de introducirlo dentro del fichero de configuración de Hibernate (hibernate.cfg.xml). El valor recomendable es entre 10 y 50.

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

- Este valor sólo se debe de indicar cuando estemos seguro que una transacción realiza inserciones/actualizaciones de filas en conjunto.
- El valor indica el número máximo de declaraciones que Hibernate agrupará antes de pedirle al controlador que ejecute el lote. Cero o un número negativo desactiva esta función.
- Desde la versión 5.2, Hibernate permite establecer batch_size por Session.

```
entityManager.unwrap( Session.class ).setJdbcBatchSize( 10 );
```

Procesamiento por lotes

- La utilización de Session.flush() y Session.close () permite controlar el consumo de la memoria cache de la Session.
  - Session.flush(): vuelca las modificaciones a base de datos (obliga a escribir los cambios) haciendo innecesario mantenerlos en memoria.
  - Session.clear(): Limpia la cache de la sesión (borra los datos y libera la memoria).
- El algoritmo debe emplear los métodos flush() y clear() de la sesión con regularidad:

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
int BATCH_SIZE = 25;
for ( int i=0; i<100000; i++ ) {
   Customer customer = new Customer(.....);
   session.save(customer);
   if ( i > 0 && i % BATCH_SIZE == 0 ) { session.flush(); session.clear(); }
}
tx.commit();
session.close();
```


Procesamiento por lotes

- 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 sesión: el commit conlleva un flush() y un clear().
- Inconvenientes:
  - Deberemos estar seguro que la transacción se puede validar por partes y no de forma completa.

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
int BATCH_SIZE = 25;
for ( int i=0; i<100000; i++ ) {
   Customer customer = new Customer(.....);
   session.save(customer);
   if ( i > 0 && i % BATCH_SIZE == 0 ) { tx.commit(); tx = session.beginTransaction(); }
}
tx.commit();
session.close();
```

Procesamiento por lotes

- StatelessSessiones una API orientada a comandos proporcionada por Hibernate. Permite transmitir datos hacia y desde la base de datos en forma de objetos separados. Un StatelessSession no tiene ningún contexto de persistencia asociado y no proporciona muchas de las semánticas del ciclo de vida de nivel superior.
- Algunas de las cosas que no proporciona un StatelessSession son:
  - un caché de primer nivel o interacción con cualquier caché de consulta o de segundo nivel
  - escritura diferida transaccional o verificación sucia automática
- Limitaciones de StatelessSession son:
  - Las operaciones realizadas mediante una sesión sin estado nunca se transmiten en cascada a instancias asociadas.
  - Las colecciones son ignoradas por una sesión sin estado.
  - No se admite la carga diferida de asociaciones.
  - Las operaciones realizadas a través de una sesión sin estado omiten el modelo de eventos y los interceptores de Hibernate.
  - Es una abstracción de nivel inferior mucho más cerca del JDBC subyacente.

Procesamiento por lotes

```
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 o con anotaciones las instrucciones DML que debe ejecutar.
- Esto nos permite añadir elementos adicionales propios del gestor, dado que al ser definiciones SQL nativas se podría usar cualquier característica especifica que necesitemos de la base de datos que estemos usando.
- El fichero de mapeo de Hibernate puede incluir los siguientes tags para especificarlo
  - <sql-insert>
  - <sql-update>
  - <sql-delete>

Profesor.hbm.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>
```

- JPA dispone de sus propias anotaciones @SQLInsert, @SQLUpdate y @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();
```

CONSULTAS CON JPA CRITERIA API

Introducción

- La JPA Criteria API proporciona una forma alternativa a definir las consultas textuales de HQL, JPQL y SQL nativo, que es principalmente útil para crear consultas dinámicas cuya estructura exacta solo se conoce en tiempo de ejecución. La antigua Hibernate Criteria API está marcada como obsoleta.
- A diferencia de las consultas textuales, cadenas con la consulta, las consultas por criterios se definen mediante la instanciación de objetos Java que representan elementos de consulta. Las consultas por criterios son esencialmente un grafo de objetos, donde cada parte del grafo representa una parte cada vez más atómica (a medida que navegamos por este grafo) de la consulta.
- Una de las principales ventajas de utilizar criterios es que los errores se pueden detectar antes, durante la compilación en lugar de en tiempo de ejecución.
- Las consultas basadas en cadenas, que son muy similares a las consultas SQL, son más fáciles de comprender y crear si son estáticas, pero cuando son dinámicas, como las que dependen de selecciones de usuario o migran por las capas de la aplicación, componer la cadena se vuelve farragoso y es susceptible de cometer errores sintácticos (tiempo de ejecución) y se corre el riesgo de inyección de SQL.
- Las consultas basadas en criterios es una forma alternativa de definir el SQL nativo a generar por lo que son equivalentes en potencia y eficiencia a las consultas textuales. Por lo tanto, elegir un método sobre el otro es una cuestión de casos de uso: son complementarios.

Crear la consulta

- Las importaciones se realizaran desde javax.persistence.criteria y no desde org.hibernate.Criteria.
- Las consultas basadas en criterios requieren crear el criterio (instancia de CriteriaQuery) para generar la consulta. Para crear y definir el criterio se requiere un CriteriaBuilder que se obtiene a través del Entity Manager. El CriteriaBuilder suministra métodos para crear el criterio de consulta y crear los diferentes elementos del criterio. El CriteriaQuery suministra métodos para configurar las diferentes partes del criterio de consulta.
- Una vez creado el criterio, se puede crear y ejecutar la consulta:

```
CriteriaBuilder cb = entityManager.getCriteriaBuilder();
CriteriaQuery<Profesor> criteria = cb.createQuery(Profesor.class);
Root<Profesor> root = criteria.from(Profesor.class);
criteria.where( cb.equal( root.get("id"), "1" ) );
List<Profesor> results = entityManager.createQuery(criteria).getResultList();
```

- Dado que createQuery crea un objeto de con la interfaz Query, se pueden establecer las mismas restricciones de timeout, paginación, solo lectura, … utilizadas en las consultas textuales y materializar la consulta con los mismos método: Query.getResultList, Query.getResultStream(), Query.getSingleResult() y Query.scroll().

Parametrización de consultas

- Para flexibilizar la definición de las consultas y la reutilización de las mismas, se pueden utilizar parámetros para los valores dependientes.
- Para definir un parámetro:

```
ParameterExpression<Integer> paramId = cb.parameter(Integer.class);
```

- Para utilizar el parámetro en la definición de la consulta:

```
criteria.where( cb.equal( root.get("id"), paramId ) );
```

- El mismo parámetro puede emplearse en tantas partes de la definición la consulta como sea necesario.
- Mediante Query.setParameter() se establece el valor del parámetro antes de materializar la consulta:

```
List<Profesor> results = entityManager.createQuery(criteria)
    .setParameter(paramId, 1);
    .getResultList();
```

Cláusula FROM

- La cláusula FROM es responsable de definir el alcance de los tipos de modelos de objetos disponibles para el resto de la consulta.
- El método CriteriaQuery.from() permite establecer las entidades que participan en la cláusula FROM y devuelve un objeto Root tipado que representa el alias de la entidad. Las raíces definen la base a partir de la cual todas las combinaciones, rutas y atributos están disponibles en la consulta.

```
Root<Profesor> root = criteria.from(Profesor.class );
```

- Invocar varias veces el método CriteriaQuery.from crea un CROSS JOIN.

```
Root<Profesor> profesor = criteria.from(Profesor.class );
Root<Asignatura> asignatura = criteria.from(Asignatura.class );
```

- Para establecer combinaciones INNER y LEFT OUTER se utiliza el Root.join().

```
Join< Profesor, CorreoElectronico> join = root.join("correosElectronicos", JoinType.INNER);
Join< Profesor, CorreoElectronico> join = root.join("correosElectronicos", JoinType.LEFT);
```

- Las consultas de criterios pueden especificar que los datos asociados se obtengan junto con el propietario. Si se quiere precargar de los datos mediante un JOIN FECTH:

```
Fetch< Profesor, CorreoElectronico> fetch = root.fetch("correosElectronicos");
```

- Root, Join y Fetch comparten los mismos métodos para la generación de elementos.

Expresiones

- Las expresiones representan de forma abstracta un conjunto de operadores y operaciones, son la base de la generación de las expresiones SQL:

```
Expression<String> nombreCompleto = cb.concat(root.get("nombre"), cb.concat(" ",
root.get("apellidos")));
```

- Las instancias de las expresiones se utilizan como argumentos de los diferentes métodos de configuración de las consultas. Las expresiones se deben generar a través de los métodos de las diferentes clase como CriteriaBuilder, Root, …
- Las expresiones de ruta (Path) son una pieza clave debido a la potencia y flexibilidad, indican el camino a seguir desde las raíces para localizar los diferentes elementos:

```
Path<Integer> cmp = root.get("id");
Path<String> cmp = root.get("correosElectronicos").get("direccion");
```

- Las expresiones de tipo de predicado, simple o compuesto, son una conjunción o disyunción de restricciones. Se considera que un predicado simple es una conjunción con una sola conjunción.

```
Predicate where = cb.lessThan( root.get("actorId"), "3" );
```

Expresiones

- Aritméticas:
  - sum(): Crea una expresión que devuelva la suma de sus argumentos.
  - diff(): Crea una expresión que devuelva la diferencia entre sus argumentos.
  - prod(): Crea una expresión que devuelva el producto de sus argumentos.
  - quot(): Crea una expresión que devuelva el cociente de sus argumentos.
  - mod(): Crea una expresión que devuelva el módulo de sus argumentos.
  - neg(): Crea una expresión que devuelva la negación aritmética de su argumento.
  - abs(): Cree una expresión que devuelva el valor absoluto de su argumento.
  - sqrt(): Crea una expresión que devuelva la raíz cuadrada de su argumento.
- Cadenas:
  - concat(): Crea una expresión para la concatenación de cadenas.
  - length(): Crea una expresión para devolver la longitud de una cadena.
  - lower(): Crea una expresión para convertir una cadena a minúsculas.
  - upper(): Crea una expresión para convertir una cadena a mayúsculas.
  - substring(): Crea una expresión para la extracción de subcadenas.
  - trim(): Crea una expresión para recortar el carácter de ambos extremos de una cadena.

Expresiones

- De comparación:
  - equal(): Crea un predicado para probar los argumentos de igualdad.
  - notEqual(): Crea un predicado para probar los argumentos de desigualdad.
  - ge(), greaterThanOrEqualTo(): Crea un predicado para probar si el primer argumento es mayor o igual que el segundo.
  - gt(), greaterThan(): Crea un predicado para probar si el primer argumento es mayor que el segundo.
  - le(), lessThanOrEqualTo(): Crea un predicado para probar si el primer argumento es menor o igual que el segundo.
  - lt(), lessThan(): Crea un predicado para probar si el primer argumento es menor que el segundo.
  - between(): Crea un predicado para probar si el primer argumento está entre el segundo y el tercer argumento en valor.
  - like(): Crea un predicado para probar si la expresión satisface el patrón dado.
  - notLike(): Crea un predicado para probar si la expresión no satisface el patrón dado.
  - in(): Crea un predicado para probar si la expresión dada está contenida en una lista de valores.

Expresiones

- Lógicas:
  - and(): Crea una conjunción de las expresiones booleanas dadas.
  - or(): Crea una disyunción de las expresiones booleanas dadas.
  - not(): Crea una negación de la restricción dada.
- Condicionales:
  - coalesce(): Crea una expresión que devuelva nulo si todos sus argumentos se evalúan como nulos y el valor del primer argumento no nulo en caso contrario.
  - nullif(): Crea una expresión que pruebe si sus argumentos son iguales, devolviendo nulos si lo son y el valor de la primera expresión si no lo son.
  - selectCase(): Crea una expresión de CASE simple.
- Literales:
  - literal(): Crea una expresión para un literal.
  - nullLiteral(): Crea una expresión para un literal nulo con el tipo dado.

Expresiones

- De Comprobación:
  - isNull(): Crea un predicado para probar si la expresión es nula.
  - isNotNull(): Crea un predicado para probar si la expresión no es nula.
  - isEmpty(): Crea un predicado que pruebe si una colección está vacía.
  - isNotEmpty(): Crea un predicado que pruebe si una colección no está vacía.
  - isFalse(): Crea una prueba de predicado para un valor falso.
  - isTrue(): Crea una prueba de predicado para obtener un valor real.
  - isMember(): Crea un predicado que pruebe si un elemento es miembro de una colección.
  - isNotMember(): Crea un predicado que pruebe si un elemento no es miembro de una colección.
- Sobre Subconsultas
  - all(): Crea una expresión completa sobre los resultados de la subconsulta.
  - any(): Crea una expresión any sobre los resultados de la subconsulta.
  - some(): Crea una expresión some sobre los resultados de la subconsulta.
  - exists(): Crea un predicado que pruebe la existencia de un resultado de subconsulta.

Expresiones

- De Agregación
  - avg(): Crea una expresión agregada aplicando la operación avg.
  - count(): Crea una expresión agregada aplicando la operación de recuento.
  - countDistinct(): Crea una expresión agregada aplicando la operación de conteo distinto.
  - greatest(): Crea una expresión agregada para encontrar el mayor de los valores (cadenas, fechas, …).
  - least(): Crea una expresión agregada para encontrar el menor de los valores (cadenas, fechas, …).
  - max(): Crea una expresión agregada aplicando la operación numérica máxima.
  - min(): Crea una expresión agregada aplicando la operación mínima numérica.
- Variadas:
  - function(): Crea una expresión para la ejecución de una función de base de datos.
  - size(): Crea una expresión que pruebe el tamaño de una colección.
  - currentDate(), currentTime(), currentTimestamp(): Crea una expresión para devolver la fecha actual, la hora actual o la marca temporal actual.

Cláusula SELECT

- La cláusula SELECT identifica qué entidad/proyección o valores devolver como resultado de la consulta.
- Al crear el criterio se establece el tipo de los valores del resultado:

```
CriteriaQuery<Profesor> criteria = cb.createQuery(Profesor.class);
Root<Profesor> root = criteria.from(Profesor.class );
```

- Por defecto devuelve las instancias del Root. Para modificar la selección se utiliza el método Root.select().

```
CriteriaQuery<CorreoElectronico> criteria = cb.createQuery(CorreoElectronico.class);
Root<Profesor> root = criteria.from(Profesor.class );
criteria.select(root.get("correosElectronicos"));
```

- Para seleccionar una expresión:

```
CriteriaQuery<String> criteria = cb.createQuery(String.class);
Root<Profesor> root = criteria.from(Profesor.class );
criteria.select(root.get("nombre"));
```

Cláusula SELECT

- Para devolver múltiples valores:

```
CriteriaQuery<Object[]> criteria = cb.createQuery(Object[].class);
Root<Profesor> root = criteria.from(Profesor.class );
criteria.select(cb.array(root.get("nombre"), root.get("direccion")));
```

- Como alternativa se dispone de multiselect que se encarga de agruparlos en un array:

```
root.multiselect(root.get("nombre"), root.get("direccion"));
```

- Para crear una nueva instancio o proyección:

```
CriteriaQuery<Elemento> criteria = cb.createQuery(Elemento.class);
Root<Profesor> root = criteria.from(Profesor.class );
criteria. select( cb.construct(Elemento.class, root.get("nombre"), root.get("direccion")));
```

- Para incluir DISTINCT en la cláusula SELECT:

```
criteria.select(root.get("nombre")).distinct(true);
```


Filtrado y agrupamiento

- La cláusula WHERE restringe los resultados devueltos de una consulta de selección y limita el alcance de las consultas de actualización y eliminación. La cláusula WHERE solo cuenta con una expresión condicional. El método where() establece la cláusula WHERE y acepta un predicado o varios (implícitamente es un AND).
- La cláusula GROUP BY permite generar resultados agregados para varios grupos de valores. La cláusula GROUP BY cuenta la lista de valores de agrupamiento. El método groupBy() establece la cláusula GROUP BY y acepta una o varias expresiones de ruta.
- La cláusula HAVING restringe los grupos, opera sobre los valores agregados. La cláusula HAVING solo cuenta con una expresión condicional. El método having() establece la cláusula HAVING y acepta un predicado o varios (implícitamente es un AND).

```
CriteriaBuilder cb = entityManager.getCriteriaBuilder();
CriteriaQuery<Object[]> criteria = cb.createQuery(Object[].class);
Root<Asignatura> root = criteria.from(Asignatura.class );
Join<Asignatura, Alumno> join = root.join("alumnos", JoinType.LEFT);
criteria.multiselect(root.get("nombre"), root.get("tipo"), cb.count(root.get("alumnos").get("tipo")));
criteria.where( cb.equal( root.get("modalidad"), Modalidades.PRESENCIAL ) );
criteria.groupBy(emp)
.having(cb.ge(cb.count(root.get("alumnos").get("tipo")), 0));
```

Ordenación

- La cláusula ORDER BY se utiliza para especificar los valores que se utilizarán para ordenar el resultado.
- Las expresiones de ordenación son:
  - asc(): Crea un orden por el valor ascendente de la expresión.
  - desc(): Crea un orden por el valor descendente de la expresión.
- Para que se genere la clausula ORDER BY se utiliza el método orderBy():

```
criteria.orderBy(cb.desc(root.get("firstName")),
      cb.desc(root.get("lastName")));
```

- Como la ordenación es costosa en términos de rendimiento, no debe solicitarse datos ordenados salvo que sea necesario.

DML

- El lenguaje de criterios también permite generar consultas de actualización y borrado.
- Con createCriteriaUpdate() crean consultas de actualización, el método set() permite establecer los nuevos valores de las columnas.

```
CriteriaBuilder cb = entityManager.getCriteriaBuilder();
CriteriaQuery<Profesor> criteria = cb.createCriteriaUpdate(Profesor.class);
Root<Profesor> root = criteria.from(Profesor.class );
criteria.set(root.get("salario"), cb.sum(root.get("salario"), 5000));
criteria.set(root.get("fechaRevision"), cb.currentTimestamp());
criteria.where( cb.equal( root.get("id"), "1" ) );
entityManager.createQuery(criteria).executeUpdate();
```

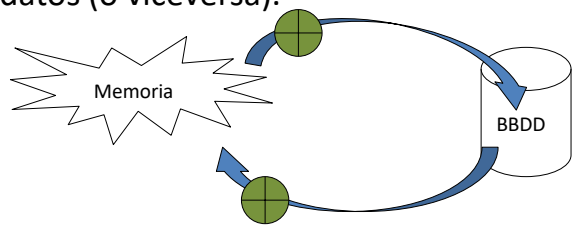
- Con createCriteriaDelete() se crean consultas de borrado:

```
CriteriaBuilder cb = entityManager.getCriteriaBuilder();
CriteriaQuery<Profesor> criteria = cb.createCriteriaDelete(Profesor.class);
Root<Profesor> root = criteria.from(Profesor.class );
criteria.where(cb.isNull(root.get("id")));
entityManager.createQuery(criteria).executeUpdate();
```

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

Contexto de Persistencia

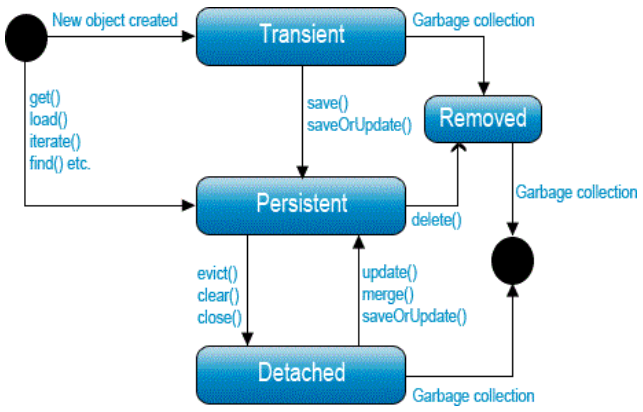
- Cuando se utilizan "conversaciones" el contexto de persistencia es crítico.
  - Unidades de trabajo de ejecución larga que requieren tiempo-para-pensar por parte del usuario.
- ¿Cómo funciona el contexto?
  - Se crea con la Aplicación Hibernate / Aplicación Java Persistent. (Session)
  - Cuando una 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())

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.

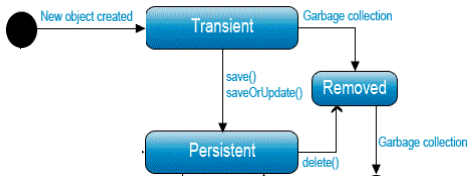
Estados de las entidades

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



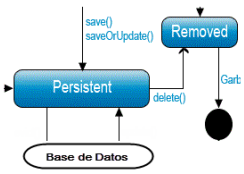
Estado Transient (Transitorio)

- 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()
  - session.saveOrUpdate()
  - entityManager.persist()



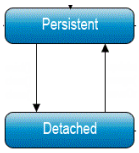
Estado Persistent (Persistente)

- Una instancia persistente tiene una representación en la base de datos y un valor identificador.
- Los objetos se convierten en persistentes cuando se han recuperado de la base de datos, se han añadido al contexto (save, saveOrUpdate, persist) o cuando se crea una referencia de otro objeto persistente ya gestionado.
- Todas las instancias Persistentes se encuentran dentro del ámbito de Session/EntityManager 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.



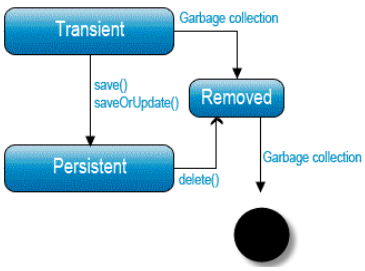
Estado Detached (Separado)

- 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.



Estado Removed (Borrado)

- Se puede eliminar una instancia de entidad de llamando a una operación explícita del gestor de persistencia:
  - session.delete
  - entityManager.remove
- 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.



Unidad de trabajo

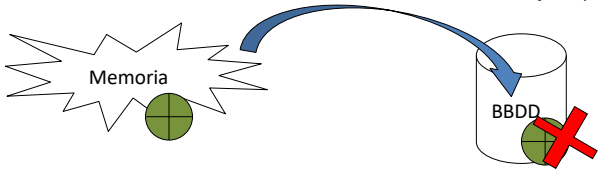
- En una aplicación de Hibernate, cargar (load) y almacenar (store) objetos se realiza básicamente mediante un cambio en su estado y esto se hace en unidades de trabajo.
- Una unidad de trabajo mantiene una lista de objetos afectados por una transacción de negocio y coordina la actualización de cambios y la resolución de problemas de concurrencia. Las unidades de trabajo son "similares" a las transacciones pero no idénticas.
- Una unidad de trabajo comienza con la obtención de una instancia de un objeto Session o EntityManager a través de su correspondiente factory: se crea un nuevo contexto de persistencia.
- Una unidad de trabajo debe ser atómica (una transacción de negocio): cargar los datos necesario, agregar o realizar las operaciones con dichos datos, volcar los cambios a la base de datos, cerrar la unidad. Idealmente una única fase de carga y otra única fase de volcado con el consecutivo cierre. Una unidad de trabajo no debe reutilizarse.
- La creación de una Factory es muy caro, por ello deben ser instancias únicas (salvo si accede a varias bases de datos), pero la creación de un objeto Session o EntityManager es barato y se pueden manejar varios a la vez para operaciones diferentes.

Unidad de trabajo

- El volcado es el proceso de sincronizar el estado del contexto de persistencia con la base de datos subyacente. El EntityManager y Session exponen un conjunto de métodos, a través de los cuales se puede cambiar el estado persistente de una entidad.
- El contexto de persistencia actúa como una caché transaccional de escritura diferida, poniendo en cola cualquier cambio de estado de la entidad. Como cualquier caché de escritura diferida, los cambios se aplican primero en la memoria y se sincronizan con la base de datos durante volcado. La operación de volcado toma cada cambio de estado de la entidad y lo traduce a una declaración INSERT, UPDATE o DELETE. Debido a que las declaraciones DML están agrupadas, Hibernate puede aplicar el procesamiento por lotes de forma transparente.
- La estrategia de volcado viene dada por el flushMode del Session actual (JPA define solo dos estrategias: AUTO y COMMIT):
  - ALWAYS: vuelca siempre antes de cada consulta.
  - AUTO: solo vuelca cuando es necesario (es el modo predeterminado).
  - COMMIT: intenta retrasar el volcado hasta que se confirma la transacción actual, aunque también podría vaciar prematuramente.
  - MANUAL: se debe llamar a explícitamente Session.flush() para volcar los cambios.

Unidad de trabajo

- Todas las operaciones que producen cambios en la base de datos deben estar dentro de una transacción.
- Las transacciones puede comenzar en cualquier punto de la unidad de trabajo.
  session.beginTransaction();
  entityManager.getTransaction().begin();
- La confirmación de una transacción conlleva un volcado de todos los cambios a la base de datos.
  session.getTransaction().commit();
  entityManager.getTransaction().commit();
- El volcado a la base de datos puede generar excepciones que deben ser tratadas. 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.

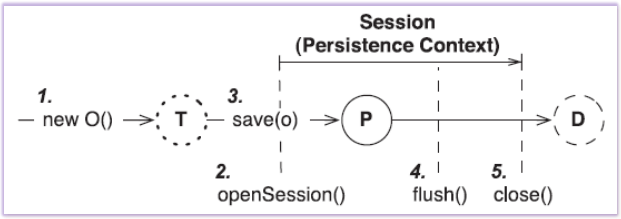


- Después de la confirmación de la transacción se puede abrir una nueva transacción.

Creando objetos Persistentes

- Cuando creamos un nuevo objeto mediante new, el objeto es Transient.
- Para hacerlo Persistent utilizamos el método session.save(objeto) o entityManager.persist(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 del 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

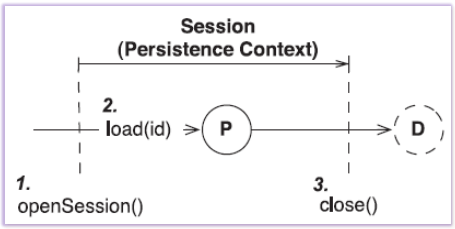
4.- Los cambios realizados en el objeto son sincronizados con la BBDD.
También se realiza un flush() de la memoria.

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

Recuperar de la base de datos

- Todos los objetos obtenidos de la base de datos quedan registrados en el contexto de persistencia como Persistent, que inicia el seguimiento de cambios.
- Hay múltiples métodos para obtener una entidad por su identificador con sus datos inicializados:
  - entityManager.find( )
  - session.get( ), session.load( )
  - session.byId( ).load( )
  - session.byId( ).loadOptional( )
- Para obtener una entidad por id natural.
  - session.bySimpleNaturalId(Book.class).getReference(isbn);
  - session.byNaturalId( ).using( ).load( );
- Para obtener una referencia de entidad sin tener que cargar sus datos (carga diferida), común para crear una asociación.
  - entityManager.getReference( ), session.getReference( )

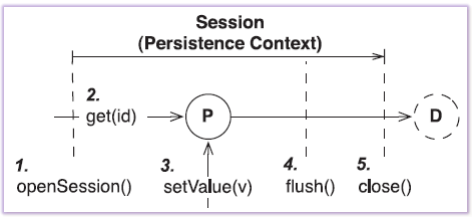
Cuando no existe el identificador:
- get: Devuelve NULL
- load: Genera una ObjectNotFoundException
- loadOptional: Devuelve un objeto Optional (Java 8)



Modificando objetos Persistentes

- Las entidades en estado administrado/persistente pueden ser manipuladas por la aplicación, y cualquier cambio se detectará y persistirá automáticamente cuando se vuelque el contexto de persistencia. No es necesario llamar a un método en particular para que las modificaciones sean persistentes.
- Posteriormente deberemos sincronizarla con la Base de Datos (commit).

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



Actualizaciones dinámicas

- De forma predeterminada, cuando modifica una entidad, todas las columnas, excepto el identificador, se establecen durante la actualización (UPDATE de todas las columnas).
- La instrucción UPDATE que contiene todas las columnas tiene dos ventajas:
  - permite beneficiarse mejor del almacenamiento en caché de sentencias JDBC.
  - permite habilitar actualizaciones por lotes incluso si varias entidades modifican propiedades diferentes.
- Como desventaja, si tiene varios índices, la base de datos puede actualizarlos de forma redundante incluso si no se modifican todos los valores de las columnas.
- La actualización dinámica le permite establecer solo las columnas que se modificaron en la entidad asociada. Para habilitar las actualizaciones dinámicas, se debe anotar la entidad con la @DynamicUpdate:

```
@Entity(name = "Product")
@DynamicUpdate
public static class Product {
```


Refrescar el estado de la entidad

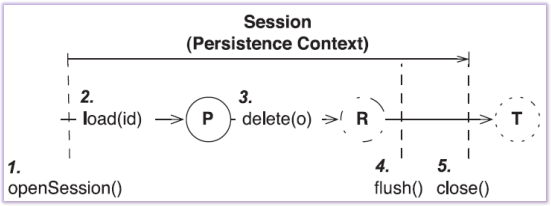
- Cuando se sabe que el estado de la base de datos ha cambiado desde que se leyeron los datos se puede volver a cargar una instancia de entidad y sus colecciones en cualquier momento.
- El refresco permite que el estado actual de la base de datos se introduzca en la instancia de la entidad y el contexto de persistencia.
- El refresco anula todos los cambios previos en la instancia de la entidad que no se hubieran volcado a la base de datos. Solo se actualizan la instancia de entidad y sus colecciones de tipo de valor, a menos que especifique REFRESH como estilo en cascada de cualquier asociación.
  - entityManager.refresh(objeto);
  - session.refresh(objeto);

Eliminación de objetos Persistentes

- Mediante el método session.delete(objeto) o entityManager.remove(objeto) podemos hacer que una entidad del contexto de persistencia se elimine de la base de datos.
- Después de la llamada al método delete, no se debería interacturar con el objeto.
- La orden SQL DELETE sólo es ejecutada cuando el contexto de persistencia se vuelca a la Base de Datos.
- Después del cierre de la sesión, el contexto libera la referencia y es destruida por el recolector de basura cuando ya no está referenciada como cualquier otro objeto.
- Cargar en el contexto una entidad solo para su eliminación puede ser muy costoso.

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();
```



Trabajar con objetos Separados

- Se considera un objeto como separado cuando esta fuera del alcance de cualquier contexto de persistencia.
- Los datos separados aún se pueden manipular, sin embargo, el contexto de persistencia ya no reconocerá automáticamente estas modificaciones, y habrá que intervenir para que los cambios sean persistentes nuevamente.
- Los datos se separan de varias formas:
  - Cuando se cierra el contexto de persistencia, todos los datos que estaban asociados con él se separan.
    - session.close(), entityManager.close();
  - Cuando se limpia el contexto de persistencia tiene el mismo efecto.
    - session.clear(), entityManager. clear();
  - Cuando se separan explícitamente:
    - session.evict(objeto), entityManager.detach(objeto);

Trabajar con objetos Separados

- Si desea guardar las modificaciones que haya realizado en un objeto individual, tenemos que volverlo a introducir en un contexto de persistencia de nuevo.
- Si se está seguro de que la sesión no contiene una instancia ya persistente con el mismo identificador, llamando a los métodos update, saveOrUpdate o lock añadimos un objeto separado a una session existente (modifica el contexto de persistencia). El método lock() permite volver a asociar un objeto pero la instancia separada no debe modificarse.
  - Session.update(objeto_separado)
- Si el contexto de persistencia contiene ya la instancia persistente con el mismo identificador, el método merge fusiona sus modificaciones: copia el valor de un objeto separado en un objeto persistente.
  - Session.merge(objeto_separado)
- Se puede verificar el estado de entidades y colecciones en relación con el contexto de persistencia:
  - entityManager.contains(person), session.contains(person)

Manejo de excepciones

- Si el EntityManager o Session arroja una excepción, incluido cualquier JDBC SQLException, se debe deshacer (rollback) inmediatamente la transacción y cerrar el EntityManager o Session.
- Revertir la transacción en la base de datos no devuelve los objetos del contexto de persistencia al estado en el que estaban al comienzo de la transacción por lo que no dejará el contexto en un estado consistente. Esto significa que el estado de la base de datos y los objetos del contexto no estarán sincronizados.
- Como regla general, ninguna excepción lanzada por Hibernate puede ser tratada como recuperable. Para asegurarse de que la sesión se cerrará, el método close() se debe invocar en un bloque finally.
- PersistenceException o HibernateException son envolturas de la mayoría de los errores que pueden ocurrir en una capa de persistencia Hibernate.
- Las excepciones mas comunes son:
  - LazyInitializationException: Cuando se accede por primera vez a una propiedad cuando la sesión ya se ha cerrado y esta activada la carga perezosa.
  - NonUniqueObjectException: Se produce cuando se intenta añadir un objeto al contexto que ya contiene una instancia persistente con el mismo identificador.
  - ObjectNotFoundException: Cuando el identificador suministrado no existe en la base de datos.

TRANSACCIONES

Introducción

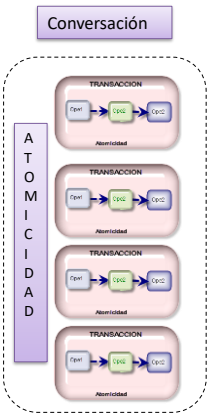
- Es importante comprender que el término transacción tiene muchos significados diferentes pero relacionados en lo que respecta a la persistencia y el mapeo de objeto/relación.
- En la mayoría de los casos de uso, estas definiciones se alinean, pero no siempre es así.
  - Podría referirse a la transacción física con la base de datos.
  - Podría referirse a la noción lógica de una transacción relacionada con un contexto de persistencia.
  - Podría referirse a la noción de aplicación de una unidad de trabajo, tal como la define el patrón arquetípico.

Concepto

- Una transacción es una secuencia de operaciones realizadas como una sola unidad lógica de trabajo.
- Una unidad lógica de trabajo debe exhibir cuatro propiedades, conocidas como propiedades ACID (atomicidad, coherencia, aislamiento y durabilidad), para ser calificada como transacción:
  - Atomicidad: Una transacción debe ser una unidad mínima de trabajo, tanto si se realizan todas sus modificaciones en los datos, como si no se realiza ninguna de ellas.
  - Coherencia: Cuando finaliza, una transacción debe dejar todos los datos en un estado coherente, es decir, se deben cumplir todas las reglas de integridad de todos los datos.
  - alslamiento: Las modificaciones realizadas por transacciones simultáneas deben ser independientes de las modificaciones llevadas a cabo por otras transacciones simultáneas.
  - Durabilidad: Una vez concluida una transacción, sus efectos son permanentes en el sistema. Las modificaciones persisten aún en el caso de producirse un error del sistema.

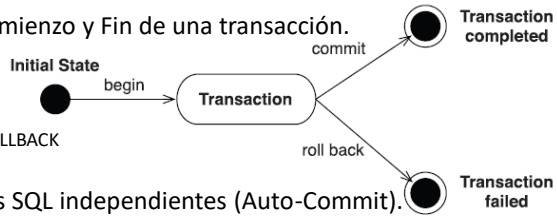
Granularidad

- Las transacciones permiten a varios usuarios trabajar simultáneamente con los mismos datos, sin comprometer la integridad y exactitud de los datos.
- 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 varios lotes.



Soporte

- La bases de datos implementan el concepto de una unidad de trabajo como una transacción de base de datos.
- Las transacciones de la base de datos nunca son opcionales. Toda la comunicación con una base de datos debe estar encapsulada por una transacción.
- Una transacción de BBDD termina de una de las siguientes maneras:
  - Confirmada/Validada (COMMIT)
  - No Validada (ROLLBACK)
- 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 DISTRIBUIDAS para asegurar la atomicidad.

Soporte

- 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/EntityManager, Hibernate proporciona la posibilidad de trabajar, eficientemente, con lecturas repetidas (cacheadas).
- 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)
  - sesión-por-petición, sesión-por-petición-con-objetos-separados, sesión-por-conversación

TimeOut

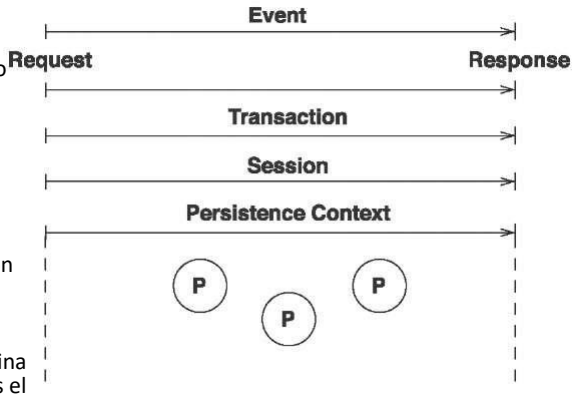
- Uno de los elementos básicos en las transacciones es el tiempo de espera de la transacción.
- Definir este tiempo de espera asegura que ninguna transacción pueda comportarse de forma "inapropiada"
  - Que tarde mucho tiempo en ejecutarse.
  - Que la transacción bloquee muchos recursos durante mucho tiempo.
- Hibernate define este TimeOut en el objeto Transaction para poder adecuarlo a las operaciones previstas.
  - session.getTransaction().setTimeout(5); // timeout 5 seg

Anti-patrón de sesión por operación

- Este anti-patrón se basa en abrir y cerrar un Session para cada llamada a la base de datos en un solo hilo.
- También es un anti-patrón en términos de transacciones de bases de datos.
- Es un anti-patrón por que:
  - Las llamadas a la base de datos son una secuencia independiente.
  - Debe utilizar la confirmación automática por instrucción, impide transacciones de mas de una instrucción.
  - No puede utilizar las caches de los elementos recuperados previamente ni realizar el seguimiento de cambios.
- Hibernate deshabilita o espera que el servidor de aplicaciones deshabilite el modo de confirmación automática inmediatamente.
- Hay que evitar el comportamiento de confirmación automática para leer datos porque es poco probable que muchas transacciones pequeñas funcionen mejor que una unidad de trabajo claramente definida, además son más difíciles de mantener y ampliar.

Patrón de sesión por petición

- Este es el patrón de transacción más común. El término petición aquí se relaciona con el concepto de un sistema que reacciona a una serie de solicitudes de un cliente/usuario en contextos cliente-servidor multiusuario (como las aplicaciones web). Se basa en la relación uno a uno entre la transacción y la sesión.
- Para cada petición (solicitud): abre una Sesión, inicia la transacción, realiza todo el trabajo relacionado con los datos, finaliza la transacción y cierra la Sesión.
- Dentro de este patrón, existe una técnica común para definir una sesión actual para simplificar la necesidad de pasar esta sesión a todos los componentes de la aplicación que puedan necesitar acceso a ella.
- Hibernate proporciona soporte para esta técnica a través del método getCurrentSession de SessionFactory. El concepto de sesión actual debe tener un alcance que defina los límites en los que es válida la noción de actual. Este es el propósito del contrato org.hibernate.context.spi.CurrentSessionContext y se puede establecer en dos ámbitos fiables.



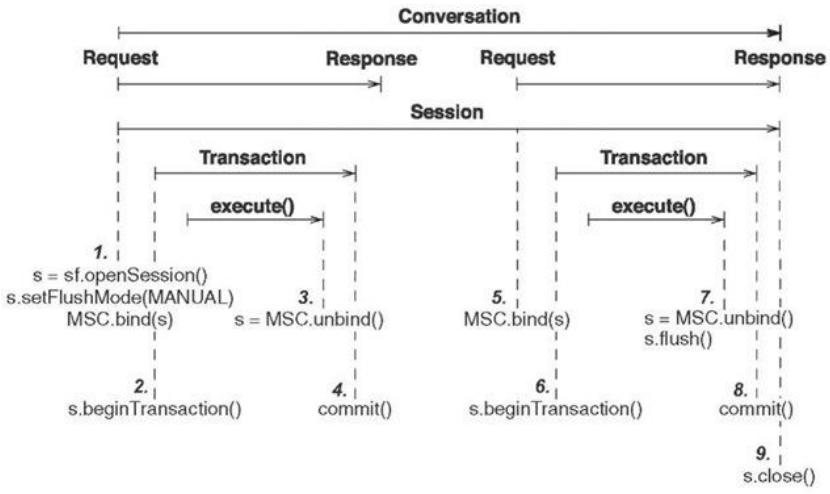
Patrón de sesión por petición

- org.hibernate.context.internal.JTASessionContext
  - En primer lugar está una transacción JTA porque permite que un hook (de devolución de llamada) sepa cuándo está terminando, lo que le da a Hibernate la oportunidad de cerrar la sesión y limpiar. Esto está representado por la implementación JTASessionContext del contrato CurrentSessionContext. Con esta implementación, se abrirá una sesión la primera vez que se llame a getCurrentSession dentro de la petición.
- org.hibernate.context.internal.ManagedSessionContext
  - En segundo lugar está el ciclo de solicitud de la aplicación. Esto se representa mejor con la implementación ManagedSessionContext del contrato CurrentSessionContext. Aquí, un componente externo es responsable de administrar el ciclo de vida y el alcance de la sesión actual. Al comienzo del alcance, se llama al método ManagedSessionContext#bind() pasando la sesión y, al final, se llama a su método unbind(). Algunos ejemplos comunes de dichos componentes externos incluyen:
    - Implementación de javax.servlet.Filter
    - Interceptor AOP con un punto en los métodos de servicio
    - Un contenedor de proxy / interceptación

Patrón sesión por conversación

- Muchos procesos de negocio requieren una serie completa de interacciones con el usuario que se entrelazan con los accesos a la base de datos. En aplicaciones web y empresariales, no es aceptable que una transacción de base de datos abarque la interacción del usuario.
- Mantener una transacción abierta en la base de datos de dependa de interacciones del usuario es un anti-patrón: utiliza bloqueos (de larga duración) a nivel de base de datos, para evitar que otros usuarios/procesos modifiquen los mismos datos y garantizar el aislamiento y la atomicidad, siendo la contención de bloqueo un cuello de botella que degrada la concurrencia y la escalabilidad, además puede producir errores por interbloqueos.
- Se utilizan varias transacciones de la base de datos para implementar la conversación. En este caso, mantener el aislamiento de los procesos de negocio se convierte en responsabilidad parcial del nivel de aplicación. Una sola conversación suele abarcar varias transacciones de la base de datos.
- Estos accesos múltiples a la base de datos solo pueden ser atómicos como un todo si solo una de estas transacciones de la base de datos (generalmente la última) almacena los datos actualizados. Todos las demás solo deben leer datos. Una forma común de recibir estos datos es a través de un diálogo estilo asistente que abarca varios ciclos de solicitud / respuesta.
- Hibernate incluye algunas características que facilitan la implementación.

Patrón sesión por conversación



Patrón sesión por conversación

- Control de versiones automático
  - Hibernate puede realizar un control de concurrencia optimista automático: detecta automáticamente (al final de la conversación) si se produjo una modificación simultánea durante el tiempo de reflexión del usuario.
- Objetos separados
  - Al utilizar el patrón de sesión por solicitud, todas las instancias cargadas se separan durante el tiempo de reflexión del usuario. Cuando se recrea la sesión, Hibernate permite volver a adjuntar los objetos y mantener las modificaciones. El control de versiones automático se utiliza para aislar modificaciones simultáneas. El patrón también se llama sesión-por-solicitud-con-objetos-separados.
- Sesión extendida (o larga)
  - El Session se puede desconectar de la conexión JDBC subyacente después de que la transacción de la base de datos se haya confirmado y se vuelva a conectar cuando se produzca una nueva solicitud de cliente. Mientras exista el objeto Session hace que incluso la reincorporación sea innecesaria. El control de versiones automático se utiliza para aislar modificaciones simultáneas y no se permitirá que se vacíe automáticamente, solo explícitamente. Este patrón se conoce como sesión por conversación.

Anti-patrón sesión por aplicación

- La sesión por aplicación también se considera un anti-patrón. Session o EntityManager no es un objeto seguro para subprocesos y está destinado a estar confinado a un solo subproceso a la vez. Si el objeto se comparte entre varios subprocesos, habrá efectos secundarios, condiciones de carrera y problemas de visibilidad.
- Una excepción lanzada por Hibernate significa que tiene que revertir la transacción de su base de datos y cerrar el Session. Si Session está vinculado a la aplicación, se debería detener la aplicación.
- La sesión almacena en caché cada objeto que se encuentra en un estado persistente (observado y verificado por Hibernate en busca de estado sucio):
  - Si se mantiene abierta durante mucho tiempo o simplemente se cargan demasiados datos, crecerá sin cesar hasta que obtenga una OutOfMemoryException.
  - Si los datos permanecen mucho tiempo en la cache se maximiza la probabilidad de datos obsoletos y errores de concurrencia.

Conexión

- En Hibernate el objeto Session utiliza un mecanismo de Lazy Loading.
  - Hibernate no consume ningún recuro a menos que sea absolutamente necesario.
- 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.

Manejo de conexiones

- De forma predeterminada, el modo de gestión de la conexión lo proporciona el coordinador de transacciones subyacente. Hay dos tipos de transacciones: RESOURCE_LOCAL (que implica una única conexión de base de datos y la transacción se controla a través de los métodos de conexión de confirmación y reversión) y JTA (que puede implicar varios recursos, incluidas conexiones de base de datos, colas JMS, etc.).
- Hibernate proporciona las siguientes estrategias:
  - INMEDIATE_ACQUISITION_AND_HOLD
    - La conexión se adquirirá tan pronto como se abra la sesión y se mantendrá hasta que se cierre la sesión.
  - DELAYED_ACQUISITION_AND_HOLD
    - La conexión se adquirirá tan pronto como sea necesario y luego se mantendrá hasta que se cierre la sesión.
  - DELAYED_ACQUISITION_AND_RELEASE_AFTER_STATEMENT
    - La conexión se adquirirá tan pronto como sea necesario y se liberara después de que se ejecute cada instrucción.
  - DELAYED_ACQUISITION_AND_RELEASE_AFTER_TRANSACTION
    - La conexión se adquirirá tan pronto como sea necesario y se liberara después de que se complete cada transacción.

Coste de 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

Problemas de las 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

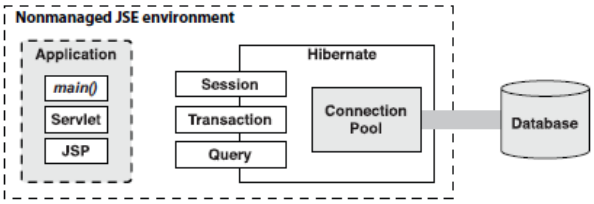
- En general, no es aconsejable crear una conexión cada vez que quiera interactuar con la base de datos.
- En su lugar, las aplicaciones Java deben utilizar un pool de conexiones que permita reutilizar conexiones abiertas:
  - Cada sub-proceso de aplicación que tiene que trabajar con la BBDD solicita una conexión al POOL.
  - Esta conexión es concedida desde el POOL: si ya hay una que está abierta se reutiliza, si no, se abre antes de utilizarla.
  - Cuando el sub-proceso acaba (se han ejecutado todas las operaciones SQL), la conexión es devuelta al POOL: si es el último que la utiliza se cierra (con un periodo de espera).
- Hay varias razones para utilizar Pool de Conexiones:
  - El Pool mantiene las conexiones y reduce al mínimo el coste de apertura y cierre de las conexiones.
  - Mediante Pool, podemos optimizar el uso de las conexiones, activando/desconectando conexiones si hay/no hay peticiones.
  - La creación de Sentencias Preparadas es caro para algunos DRIVERS, mediante el POOL podemos cachear las sentencias entre peticiones.

Estrategias de configuración

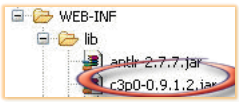
- Esquema centralizado:
  - Pool de conexiones compartido entre las aplicaciones web alojadas en el servidor JEE.
  - Un único pool con una única configuración
  - Un único tipo de pool de conexiones
  - Configurado a nivel de servidor (requiere permisos)
  - Usuario de base de datos único
  - Se publica 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)



Configuración del Pool de conexiones

- La configuración del POOL de conexiones se realiza dentro del fichero de configuración de Hibernate (src/hibernate.properties).

```
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
hibernate.c3p0.idle_test_period = 3000      // Tiempo máximo para la validación de un CX

hibernate.show_sql = true
hibernate.format_sql = true
```

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.

Problemas de simultaneidad

- Actualizaciones perdidas:
  - Este problema surge cuando dos o más transacciones seleccionan la misma fila y, a continuación, la actualizan de acuerdo con el valor seleccionado originalmente.
  - Ninguna transacción es consciente de las otras transacciones.
  - La última actualización sobrescribe las actualizaciones realizadas por las otras transacciones y, en consecuencia, se pierden datos.
- Dependencia no confirmada (lectura no actualizada o desfasada):
  - Este problema se produce cuando una transacción selecciona una fila que está siendo actualizada por otra transacción.
  - La transacción que llega en segundo lugar lee datos que todavía no han sido confirmados y que la transacción que actualiza la fila puede modificar.

Problemas de simultaneidad

- Análisis incoherente (lectura irrepetible):
  - Este problema se produce cuando una transacción obtiene acceso a la misma fila varias veces y en cada ocasión lee datos diferentes.
  - El análisis incoherente es similar a la dependencia no confirmada en tanto que una transacción está modificando los datos que está leyendo una segunda transacción.
  - Sin embargo, en el caso del análisis incoherente, los datos que lee la segunda transacción están confirmados por la transacción que realizó el cambio.
  - Además, el análisis incoherente comprende varias lecturas (dos o más) de la misma fila, y cada vez hay otra transacción que modifica la información; de ahí el término lectura irrepetible.

Problemas de simultaneidad

- Lecturas fantasmas o irrepetibles:
  - Este problema se produce cuando se realiza una acción de insertar o eliminar en una fila y ésta pertenece a un intervalo de filas que está leyendo una transacción.
  - La primera lectura que hizo la transacción en el intervalo de filas muestra una fila que ya no existe en la segunda lectura o en lecturas sucesivas porque otra transacción la ha eliminado.
  - De forma similar, como consecuencia de una inserción realizada por otra transacción, la segunda lectura o las lecturas sucesivas de la transacción muestran una fila que no existía en la primera lectura.

Niveles de aislamiento

- El estándar ANSI SQL define una serie de niveles de aislamiento de transacción que coinciden con los mismos que JTA.
- El Nivel de Aislamiento:
  - Define el grado en que se aísla una transacción de las modificaciones de datos realizadas por otras transacciones.
  - Indica los tipos y duración de los Bloqueos que utilizará el gestor de la base de datos.
  - Establece el equilibrio entre coherencia y concurrencia, los niveles superiores aseguran la coherencia pero degradan la concurrencia y viceversa para los niveles inferiores.

Niveles de aislamiento

- READ UNCOMMITTED

  Implementa las lecturas no confirmadas o el bloqueo de nivel de aislamiento 0, lo que significa que no hay bloqueos compartidos y que los bloqueos exclusivos no están garantizados. Cuando se establece esta opción, es posible leer datos no confirmados, los valores pueden cambiar y pueden aparecer y desaparecer filas en el conjunto de datos antes del final de la transacción. Se trata del menos restrictivo de los cuatro niveles de aislamiento.
- READ COMMITTED

  Especifica que se mantengan los bloqueos compartidos mientras se leen datos para evitar lecturas no actualizadas, pero se pueden modificar los datos antes del final de la transacción, lo que provoca lecturas no repetibles o datos fantasmas.

Niveles de aislamiento

- REPEATABLE READ

  Se establecen bloqueos para todos los datos utilizados en la consulta, lo que impide que otros usuarios los actualicen, aunque es posible insertar nuevas filas fantasmas en los datos que otro usuario establezca, de modo que se incluyan en lecturas posteriores de la misma transacción. Como la simultaneidad es inferior que el nivel de aislamiento predeterminado, sólo se debe usar esta opción cuando sea necesario.
- SERIALIZABLE

  Se establece un bloqueo de intervalo en el conjunto de datos, lo que impide que otros usuarios actualicen o inserten filas en el conjunto de datos hasta que finalice la transacción. Es el más restrictivo de los cuatro niveles de aislamiento. Al ser menor la simultaneidad, sólo se debe utilizar esta opción cuando sea necesario.

Niveles de aislamiento

Nivel de aislamiento	Actualización perdidas	Lectura no actualizada	Lectura no repetible	Lectura Fantasma
Lectura no confirmada (READ UNCOMMITTED)	Sí	Sí	Sí	Sí
Lectura confirmadas (READ COMMITTED)	Sí	No	Sí	Sí
Lectura repetible (REPEATABLE READ)	No	No	No	Sí
Serializable (SERIALIZABLE)	No	No	No	No

Niveles de aislamiento

- Si no indicamos nada, cada conexión JDBC utilizará el nivel de aislamiento definido por el SGBD.
- Hibernate configura el nivel de aislamiento en cada conexión JDBC obtenidas del Pool de conexiones antes de iniciar una transacción.
- 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

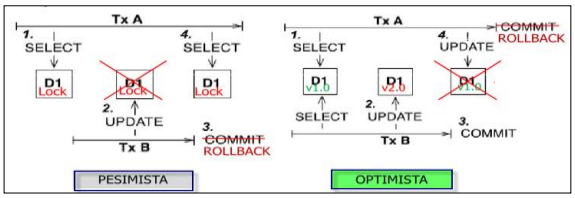
- Establecer el nivel de aislamiento es una opción global que afecta a todas las conexiones y transacciones.

Estrategias

- Optimista

  La estrategia optimista asume que no va a haber problemas por lo que se pueden completar varias transacciones sin que se afecten entre sí y, por lo tanto, las transacciones pueden continuar sin bloquear los recursos de datos que afectan. Antes de confirmarla, cada transacción verifica que ninguna otra transacción haya modificado sus datos. Si la verificación revela modificaciones conflictivas, la transacción se revierte.
- Pesimista

  La estrategia pesimista se basa en que puede fallar y que las transacciones concurrentes entrarán en conflicto entre sí, por lo que requiere que los recursos se bloqueen después de que se lean y solo se desbloqueen después de que la aplicación haya terminado de usar los datos.



Estrategia optimista

- No bloquea los datos que se van a actualizar y asume que los datos que están siendo actualizados no van a cambiar desde que se han leído. Un bloqueo optimista siempre asume que todo va a estar bien y que las modificaciones de datos en conflicto son raros.
- Para asegura que los datos que están siendo escritos son consistentes con los leídos en primera instancia, es decir, que ninguna otra transacción ha actualizado los datos después de la lectura, se utiliza el VERSIONADO.
- Cuando la aplicación usa transacciones largas o conversaciones que abarcan varias transacciones de base de datos, se puede almacenar información de la versión de los datos de modo que si la misma entidad se actualiza mediante dos conversaciones, la última en confirmar los cambios encontrará discrepancias con la versión leída, será informada mediante una excepción del conflicto y evite sobre escribir el trabajo de la otra conversación. Este enfoque garantiza cierto aislamiento, pero escala bien y funciona particularmente bien en situaciones de lecturas frecuentes pero escasas escrituras.
- Hibernate proporciona dos mecanismos diferentes para almacenar información de versión: un número de versión auto incrementable o una marca temporal.

Estrategia optimista

- El inconveniente del mecanismo de versionado es que requiere modificar la estructura de las tablas en la base de datos. La mayoría de los motores de bases de datos dispones de sus propios mecanismos de versionado.
- Mediante la anotación @Version se mapea la columna responsable del control del versionado (según JPA, los tipos válidos para estos atributos se limitan a: int o Integer, short o Short, long o Long y java.sql.Timestamp):

```
@Version
private long version;
```

- Si el valor de versión lo genera la base de datos, se debe complementar con la anotación @Generated(GenerationTime.ALWAYS)

```
@Version
@Generated(GenerationTime.ALWAYS)
private Timestamp version;
```


Estrategia optimista

- De forma predeterminada, cada modificación de un atributo de la entidad activará el incremento de versión. Para evitar que una determinada propiedad de la entidad active el versionado es necesario anotarla con @OptimisticLock:

```
@OptimisticLock( excluded = true )
private long count;
```

- La aplicación tiene prohibido alterar el valor de versión establecido por Hibernate. Para aumentar artificialmente el número de versión se utiliza LockModeType.OPTIMISTIC_FORCE_INCREMENT o LockModeType.PESSIMISTIC_FORCE_INCREMENT.
- Para interceptar los errores de concurrencia optimista, si no son de la misma versión, Hibernate lanzará la excepción OptimisticLockException (EntityManager) o el nativo StaleObjectStateException (Session).

Estrategia optimista

- Cuando no se pueda o quiera modificar la estructura de la base de datos, Hibernate admite una forma de bloqueo optimista que no requiere un "atributo de versión" dedicado.
- La idea es hacer que Hibernate realice "comprobaciones de versión" utilizando todos los atributos de la entidad o solo los atributos que han cambiado. Esto se logra mediante el uso de la anotación @OptimisticLocking. Las estrategias disponibles son (OptimisticLockType):
  - NONE: el bloqueo optimista está deshabilitado incluso si hay una anotación @Version presente.
  - VERSION (el valor por defecto): realiza un bloqueo optimista basado el versionado (@Version)
  - ALL: realiza un bloqueo optimista basado en todos los campos como una extensión de la cláusula WHERE comparando con los valores originales en las instrucciones UPDATE y DELETE de SQL
  - DIRTY: realiza un bloqueo optimista basado en los campos sucios (camviado) como una extensión de la cláusula WHERE comparando con los valores originales en las instrucciones UPDATE y DELETE de SQL.

```
@Entity
@OptimisticLocking(type = OptimisticLockType.DIRTY)
@DynamicUpdate
@SelectBeforeUpdate
public static class Profesor {
```

- La anotación @DynamicUpdate para que la instrucción UPDATE tenga en cuenta todas las propiedad sucias y la anotación @SelectBeforeUpdate para que las entidades separadas sean manejadas correctamente cuando se agregan con Session.update(entity)

Estrategia pesimista

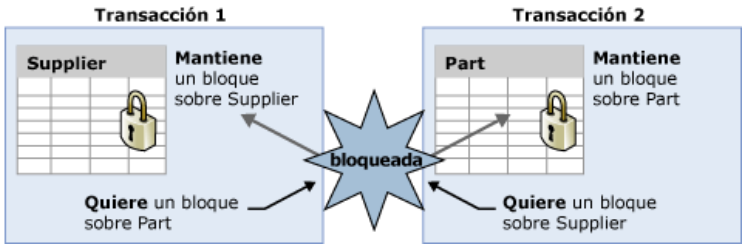
- Los datos son bloqueados previamente a su modificación para evitar que nadie los modifique. Hibernate siempre usa el mecanismo de bloqueo de la base de datos y nunca bloquea objetos en la memoria.
- Una vez que los datos a actualizar han sido bloqueados la aplicación puede acometer los cambios y, con commit o rollback, 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.
- Normalmente, solo se necesita especificar un nivel alto de aislamiento para las conexiones JDBC y dejar que la base de datos maneje los problemas de bloqueo. Los bloqueos pesimistas pueden ser obtenidos mediante:
  - Session.lock (Elemento , LockMode.XXXXX);
  - Session.load(), especificando un LockMode
  - Query.setLockMode(LockMode.XXXXX)
- Ninguna transacción simultánea puede obtener un bloqueo de los mismos datos.
- Los bloqueos pesimistas producen menos errores pero consume mas tiempo y recursos en su implementación.

Modos de bloqueo

LockMode (Hibernate)	LockModeType (JPA)	Descripción
NONE	NONE	La ausencia de bloqueos. Todos los objetos cambian a este modo de bloqueo al final de una transacción. Los objetos asociados con la sesión a través de update() o saveOrUpdate() también comienzan en este modo de bloqueo.
READ	READ y OPTIMISTIC	La versión de la entidad se verifica al final de la transacción en ejecución.
WRITE	WRITE OPTIMISTIC_FORCE_INCREMENT	La versión de la entidad se incrementa automáticamente incluso si la entidad no ha cambiado.
PESSIMISTIC_FORCE_INCREMENT	PESSIMISTIC_FORCE_INCREMENT	La entidad se bloquea de forma pesimista y su versión se incrementa automáticamente incluso si la entidad no ha cambiado.
PESSIMISTIC_READ	PESSIMISTIC_READ	La entidad está bloqueada de manera pesimista mediante un bloqueo compartido si la base de datos admite dicha característica. De lo contrario, se utiliza un bloqueo explícito.
PESSIMISTIC_WRITE, UPGRADE	PESSIMISTIC_WRITE	La entidad está bloqueada mediante un bloqueo explícito.
UPGRADE_NOWAIT	PESSIMISTIC_WRITE Con timeout = 0	La solicitud de adquisición de bloqueo falla inmediatamente si la fila ya está bloqueada.
UPGRADE_SKIPLOCKED	PESSIMISTIC_WRITE Con timeout = -2	La solicitud de adquisición de bloqueo omite las filas ya bloqueadas. Utiliza un SELECT … FOR UPDATE SKIP LOCKED (Oracle y PostgreSQL 9.5), o SELECT … with (rowlock, updlock, readpast) con SQL Server.

Interbloqueos

- Cuando un proceso encuentra los datos bloqueados pasa automáticamente a esperar a que los datos estén disponibles.
- Un Deadlock (también llamado bloqueo mutuo, interbloqueo, etc.) sucede cuando una transacción que bloquea un conjunto de datos intenta acceder a datos bloqueados por lo que queda a la espera de una transacción que acaba bloqueada al intentar acceder a los datos bloqueados por la primera transacción (dependencia cíclica entre dos o más subprocesos para algún conjunto de resultados).



Interbloqueos

- El interbloqueo tiene mala solución, hay que cancelar una de las transacciones.
- 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
- Para evitar los interbloqueos:
  1. No abrir una transacción mientras examina los datos si es posible.
  2. Las transacciones deben ser atómicas.
  3. Hacer la transacción lo más corta posible.
  4. La transacción no debería depender de interfaz de usuario.
  5. Hacer un uso inteligente de los niveles más bajos de aislamiento de las transacciones.
  6. Manejar siempre que sea posible los conjuntos de datos en el mismo orden.

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 devueltos 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.

ANEXOS

DTO

- Un objeto de transferencia de datos (DTO) es un objeto que define cómo se enviarán los datos a través de la red.
- Su finalidad es:
  - Desacoplar del nivel de servicio de la capa de base de datos.
  - Quitar las referencias circulares.
  - Ocultar determinadas propiedades que los clientes no deberían ver.
  - Omitir algunas de las propiedades con el fin de reducir el tamaño de la carga.
  - Eliminar el formato de grafos de objetos que contienen objetos anidados, para que sean más conveniente para los clientes.
  - Evitar el "exceso" y las vulnerabilidades por publicación.

Lombok

https://projectlombok.org/

- En las clases Java hay mucho código que se repite una y otra vez: constructores, equals, getters y setters. Métodos que quedan definidos una vez que dicha clase ha concretado sus propiedades, y que salvo ajustes menores, serán siempre sota, caballo y rey.
- Project Lombok es una biblioteca de java que se conecta automáticamente al editor y crea herramientas que automatizan la escritura de java.
- Mediante simples anotaciones ya nunca mas vuelves a escribir otro método get o equals.

```
@Data @AllArgsConstructor @NoArgsConstructor public class MyDTO {
    private long id;
    private String name;
}
```

- La anotación @Value (no confundir con la de Spring) crea la versión de solo lectura.
- Es necesario agregar las bibliotecas al proyecto y configurar el entorno.

ModelMapper

http://modelmapper.org/

- Las aplicaciones a menudo contienen modelos de objetos similares pero diferentes, donde los datos en dos modelos pueden ser similares pero la estructura y las responsabilidades de los modelos son diferentes. El mapeo de objetos facilita la conversión de un modelo a otro, permitiendo que los modelos separados permanezcan segregados.
- ModelMapper facilita el mapeo de objetos, al determinar automáticamente cómo se mapea un modelo de objeto a otro, de acuerdo con las convenciones, de la misma forma que lo haría un ser humano, al tiempo que proporciona una API simple y segura de refactorización para manejar casos de uso específicos.

```
ModelMapper modelMapper = new ModelMapper();
OrderDTO orderDTO = modelMapper.map(order, OrderDTO.class);
```

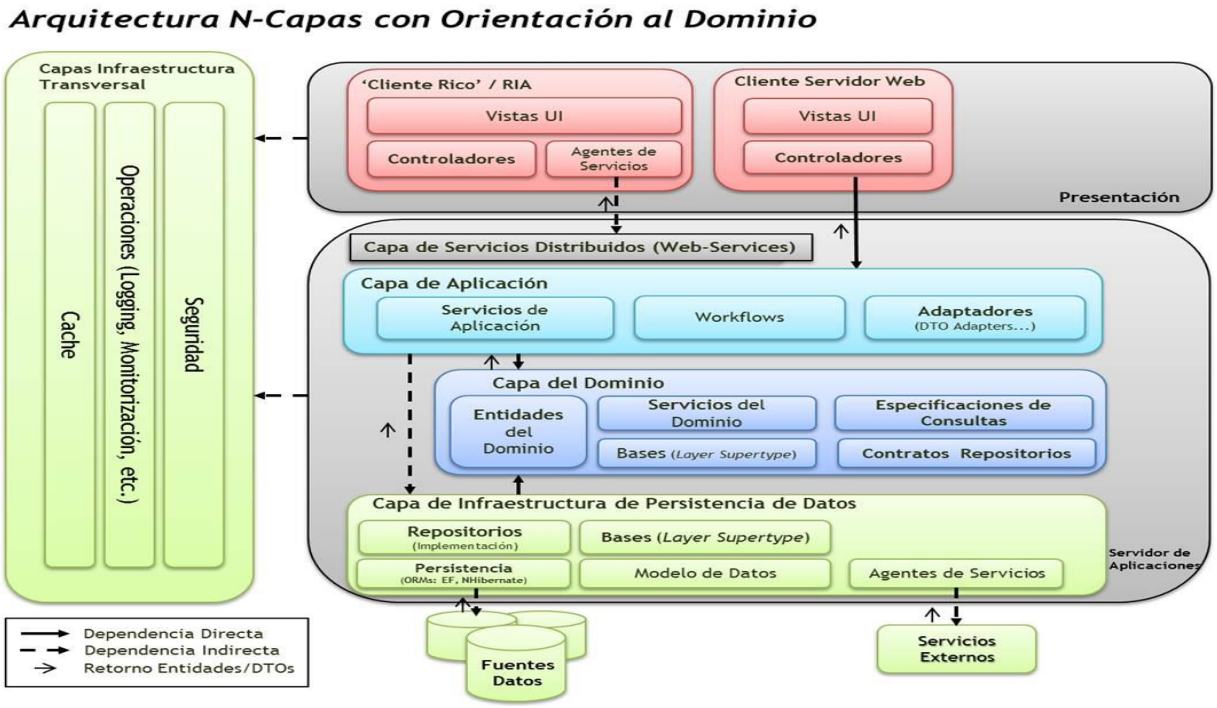

GenerateData

- GenerateData es una herramienta para la generación automatizada de juegos de datos.
- Ofrece ya una serie de datos precargados en BBDD y un conjunto de tipos de datos bastante amplio, así como la posibilidad de generar tipos genéricos.
- Podemos elegir en que formato se desea la salida de entre los siguientes:
  - CSV
  - Excel
  - HTML
  - JSON
  - LDIF
  - Lenguajes de programación (JavaScript, Perl, PHP, Ruby, C#)
  - SQL (MySQL, Postgres, SQLite, Oracle, SQL Server)
  - XML
- Online: http://www.generatedata.com/?lang=es
- Instalación (PHP): http://benkeen.github.io/generatedata/

Mockaroo

https://www.mockaroo.com/

- Mockaroo permite descargar rápida y fácilmente grandes cantidades de datos realistas de prueba generados aleatoriamente en función de sus propias especificaciones que luego puede cargar directamente en su entorno de prueba utilizando formatos CSV, JSON, XML, Excel, SQL, … No se requiere programación y es gratuita (para generar datos de 1.000 en 1.000).
- Los datos realistas son variados y contendrán caracteres que pueden no funcionar bien con nuestro código, como apóstrofes o caracteres unicode de otros idiomas. Las pruebas con datos realistas harán que la aplicación sea más robusta porque detectará errores en escenarios de producción.
- El proceso es sencillo ya que solo hay que ir añadiendo nombres de campos y escoger su tipo. Por defecto nos ofrece mas de 140 tipos de datos diferentes que van desde nombre y apellidos (pudiendo escoger estilo, género, etc...) hasta ISBNs, ubicaciones geográficas o datos encriptados simulados.
- Además es posible hacer que los datos sigan una distribución Normal o de Poisson, secuencias, que cumplan una expresión regular o incluso que fuercen cadenas complicadas con caracteres extraños y cosas así. Tenemos la posibilidad de crear fórmulas propias para generarlos, teniendo en cuenta otros campos, condicionales, etc... Es altamente flexible.



Capas de DDD

- Interface de usuario (User Interface)
  - Responsable de presentar la información al usuario, interpretar sus acciones y enviarlas a la aplicación.
- Aplicación (Application)
  - Responsable de coordinar todos los elementos de la aplicación. No contiene lógica de negocio ni mantiene el estado de los objetos de negocio. Es responsable de mantener el estado de la aplicación y del flujo de esta.
- Dominio (Domain)
  - Contiene la información sobre el Dominio. Es el núcleo de la parte de la aplicación que contiene las reglas de negocio. Es responsable de mantener el estado de los objetos de negocio. La persistencia de estos objetos se delega en la capa de infraestructura.
- Infraestructura (Infraestructure)
  - Esta capa es la capa de soporte para el resto de capas. Provee la comunicación entre las otras capas, implementa la persistencia de los objetos de negocio y las librerías de soporte para las otras capas (Interface, Comunicación, Almacenamiento, etc..)

- Dado que son capas conceptuales, su implementación puede ser muy variada y en una misma aplicación, tendremos partes o componentes que formen parte de cada una de estas capas.

Servicio

- Los servicios representan operaciones, acciones o actividades que no pertenecen conceptualmente a ningún objeto de dominio concreto. Los servicios no tienen ni estado propio ni un significado más allá que la acción que los definen. Se anotan con @Service.
- Podemos dividir los servicios en tres tipos diferentes:
  - Domain services
    - Son responsables del comportamiento más específico del dominio, es decir, realizan acciones que no dependen de la aplicación concreta que estemos desarrollando, sino que pertenecen a la parte más interna del dominio y que podrían tener sentido en otras aplicaciones pertenecientes al mismo dominio.
  - Application services
    - Son responsables del flujo principal de la aplicación, es decir, son los casos de uso de nuestra aplicación. Son la parte visible al exterior del dominio de nuestro sistema, por lo que son el punto de entrada-salida para interactuar con la funcionalidad interna del dominio. Su función es coordinar entidades, value objects, domain services e infrastructure services para llevar a cabo una acción.
  - Infrastructure services
    - Declaran comportamiento que no pertenece realmente al dominio de la aplicación pero que debemos ser capaces de realizar como parte de este.

Patrón: Eventos de dominio

- Motivación:
  - Un servicio a menudo necesita publicar eventos cuando actualiza sus datos. Estos eventos pueden ser necesarios, por ejemplo, para actualizar una vista CQRS. Alternativamente, el servicio podría participar en una Saga basada en coreografía, que utiliza eventos para la coordinación.
- Intención:
  - ¿Cómo publica un servicio un evento cuando actualiza sus datos?
- Solución:
  - Organizar la lógica de negocios de un servicio como una colección de agregados DDD que emiten eventos de dominio cuando se crean o actualizan. El servicio publica estos eventos de dominio para que puedan ser consumidos por otros servicios.
- Patrones relacionados:
  - Los patrones Saga y CQRS crean la necesidad de este patrón.
  - El patrón Agregado se utiliza para estructurar la lógica empresarial.
  - El patrón de Bandeja de salida transaccional se utiliza para publicar eventos como parte de una transacción de base de datos
  - El aprovisionamiento de eventos se utiliza a veces para publicar eventos de dominio.