ORM-Hibernate

- NetBeans

Instalación Hibernate – para uso de las HibernateTools

Tools/plugins/Avaliable…

Hibernate, Java EE Base

(si da problemas org-netbeans-libs-freemarker.jar-> ponerlo en:

C:\Program Files\NetBeans-12.4\netbeans\ide\modules)

Comprobar que está instalado:

(BotónDrcho Proyecto)\New\Other\Hibernate

Hibernate Configuration Wizard (cfg.xml)

Hibernate ReverseEngineering Wizard (reveng.xml)

Hibernate Mapping Wizard (hbm.xml)

Hibernate Mapping Files and POJOs from Database (.java)

HibernateUtil.java

HibernateUtil.java

```
public static SessionFactory buildSessionFactory() {
    try {
        return new Configuration().configure().buildSessionFactory();
    } catch (Throwable ex) {
        System.err.println("Initial SessionFactory creation failed." + ex);
        throw new ExceptionInInitializerError(ex);
    }
}
```

Jar's

botonDrcho/Add Library/

Hibernate 4.3.x

Hibernate 4.3.x (JPA2.1)

MySQL - ..java-8.0.15.jar

JAXB –

- javax.activation-1.2.0.jar
http://search.maven.org/remotecontent?filepath=com/sun/activation/javax.activation/1.2.0/javax.activation-1.2.0.jar

- jaxb-api-2.3.0.jar
http://search.maven.org/remotecontent?filepath=javax/xml/bind/jaxb-api/2.3.0/jaxb-api-2.3.0.jar

- jaxb-core-2.3.0.jar
http://search.maven.org/remotecontent?filepath=com/sun/xml/bind/jaxb-core/2.3.0/jaxb-core-2.3.0.jar

- jaxb-impl-2.3.0.jar
http://search.maven.org/remotecontent?filepath=com/sun/xml/bind/jaxb-impl/2.3.0/jaxb-impl-2.3.0.jar

Jar's del LibroSintesis o bajados de la web de sourceforge.net

Ficheros configuración

- HibernateUtil.java

HHH000041: Configured SessionFactory: null Initial SessionFactory creation failed.java.lang.ExceptionInInitializerError Exception in thread "main" java.lang.ExceptionInInitializerError

- hibernate.cfg.xml

ERROR: HHH000142: Javassist Enhancement failed:

You should be use this property in hibernate.cfg.xml – NUNCA recomendado

```
<property name="hibernate.bytecode.use_reflection_optimizer">false</property>
```

at Jan 08 12:48:46 CET 2022 WARN: Establishing SSL connection without server's identity verification is not recommended. According to MySQL 5.5.45+, 5

jdbc:mysql://localhost:3306/ejemplo?useSSL=false

Error con TimeZone

En MySQL:

SET GLOBAL time_zone = '-3:00';

Mapeos

hbm.xml

.java

JAVA JDK 16

Initial SessionFactory creation failed.java.lang.ExceptionInInitializerError

Exception in thread "main" java.lang.ExceptionInInitializerError

at institutociclostalleres.NewHibernateUtil.buildSessionFactory(NewHibernateUtil.java:24)

at institutociclostalleres.InstitutoCiclosTalleres.main(InstitutoCiclosTalleres.java:25)

Caused by: java.lang.ExceptionInInitializerError

at com.sun.xml.bind.v2.runtime.reflect.opt.AccessorInjector.prepare(AccessorInjector.java:81)

at com.C:\Users\mrnov\AppData\Local\NetBeans\Cache\12.4\executor-snippets\run.xml:111: The following error occurred while executing this line:

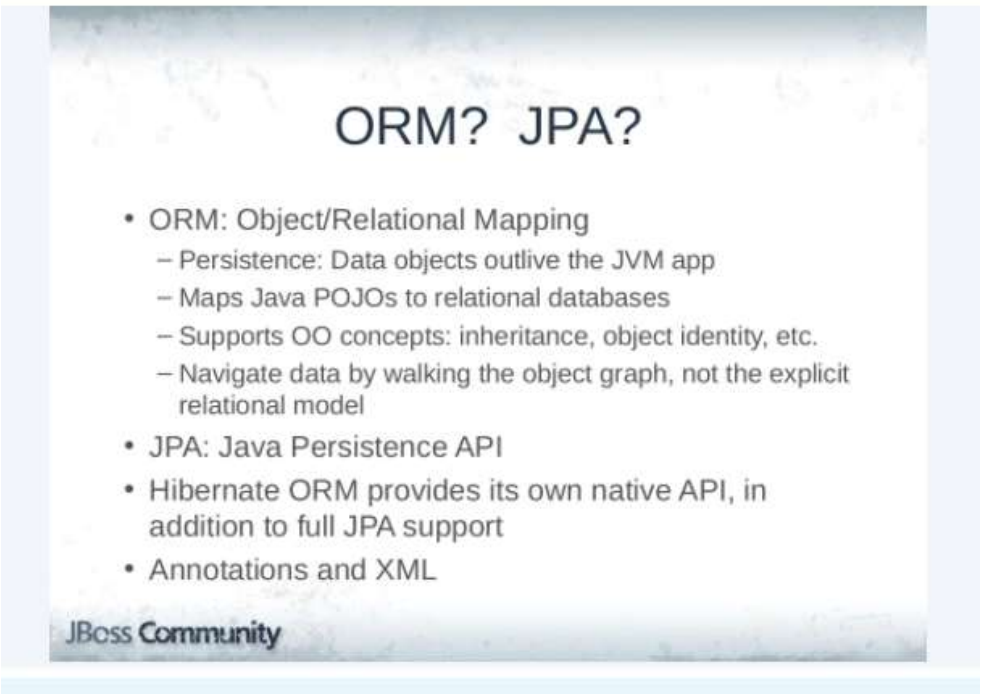
C:\Users\mrnov\AppData\Local\NetBeans\Cache\12.4\executor-snippets\run.xml:94: Java returned: 1

BUILD FAILED (total time: 6 seconds)

Solución

Uso Java JDK 1.8

- Eclipse
  - JPA+Hibernate



Instalación Hibernate

Help/Install New Software

Help/Eclipse Marketplace

Vista

Window/Show View/Other (Hibernate)

Window/Perspective/Open Perspective/Other/Hibernate

Window/Preferences/Data Management/Connectivity/Driver Definitions

File/New/Project/Java Project

(botonDercho)/Build Path/Add Library

(Añadir conector mysql)

Ahora es cuando cogemos el comprimido descargado de **Hibernate**, lo descomprimimos y copiamos (arrastrar y soltar, por ejemplo) los *.jar* contenidos en la carpeta */lib/required* a nuestra carpeta *lib*.

New/Other/Hibernate/Hibernate Configuration File (cfg.xml)

New/Other/Hibernate/Hibernate Console Configuration

New/Other/Hibernate/Hibernate Reverse Engineering File (reveng.xml)

Hibernate XML Mapping file (hbm.xml)

Jar's

C:\.._MapeoObjetoRelacional\InstalacionHibernateNB_eclipse\hibernate-release-5.6.1.Final\hibernate-release-5.6.1.Final\lib\required

Ficheros configuración

cfg.xml

```
<property name="hibernate.hbm2ddl.auto" value="" />
```

Para mapeo automático contra la BD

Mapeos

Anotaciones JPA ó fichero .xml

H2 , ApacheDerby (BDs en memoria)

Para desarrollo web – Java EE

Sino-Proyecto Ant o Maven

Ant

Maven

Pom.xml

```
<build>
<plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.8.1</version>
        <configuration>
          <source>1.8</source>
          <target>1.8</target>
        </configuration>
      </plugin>
    </plugins>
</build>
```

Instalación Hibernate

Se puede descargar https://hibernate.org/orm/releases/5.6/

O Si se usa Maven, dependencias

https://mvnrepository.com/artifact/org.hibernate/hibernate-core/5.6.1.Final

```
<!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-core  -->
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.6.1.Final</version>
</dependency>
```

Soporte para **jpa** (al usar java se)

hibernate-jpa-ejemploEclipse

```
<!-- https://mvnrepository.com/artifact/javax.persistence/javax.persistence-api -->
<dependency>
    <groupId>javax.persistence</groupId>
    <artifactId>javax.persistence-api</artifactId>
    <version>2.2</version>
</dependency>
```

Conversor JPA vs Hibernate (Entity Manager)

https://mvnrepository.com/artifact/org.hibernate/hibernate-entitymanager/5.6.1.Final

```
<!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-entitymanager -->
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-entitymanager</artifactId>
    <version>5.6.1.Final</version>
</dependency>
```

Inclusión Sistema log

(no es necesario)

(si se usa bd H2)

-formato jar, compatible jdbc, en memoria..ya hace él todo y al acabar ejecución se elimina todo

https://mvnrepository.com/artifact/com.h2database/h2/1.4.200

<!-- https://mvnrepository.com/artifact/com.h2database/h2 -->

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

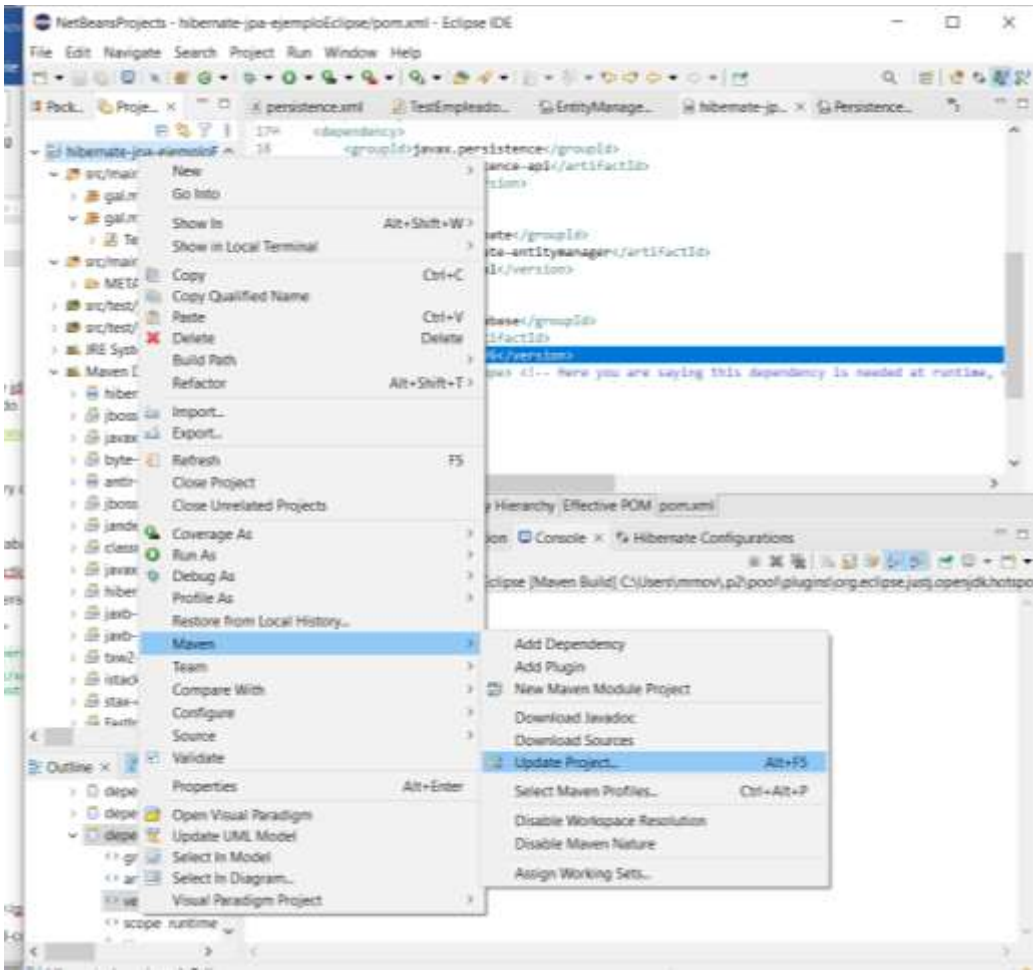
<version>1.4.200</version>

<scope>test</scope>

<version>1.4.196</version>

<scope>runtime</scope> <!-- Here you are saying this dependency is needed at runtime, not just at testing -->

</dependency>



(si se usa bd mysql)

```
<dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.15</version>
</dependency>
```

Archivos configuración

persistence.xml

The previous tutorials used the Hibernate-specific hibernate.cfg.xml configuration file. JPA, however, defines a different bootstrap process that uses its own configuration file named persistence.xml. This bootstrapping process is defined by the JPA specification. In Java™ SE environments the persistence provider (Hibernate in this case) is required to locate all JPA configuration files by classpath lookup of the META-INF/persistence.xml resource name.

Para declarar cómo acceder a la base de datos y qué driver JDBC utilizar, el archivo persistence.xml proporciona esto junto a la lista de clases que van a actuar como Entidades y otra configuración que depende de la propia implementación JPA que se esté utilizando, en este caso Hibernate.

Ejemplo de persistence.xml:

https://gist.github.com/danirod/7b23abcd5157bd47f422e2042f86e903

src/main/resources

New/folder/ **META-INF** (en main/resources)

(OJO. En Proyecto web estaría ubicado en carpeta WEB-INF)

Dentro de META-INF, crear archivo de configuración

New/XML,

al que se llamará persistence.xml (equivalente el contenido a hibernate.cfg.xml)

Uso de MenúEclipse-> Source/Format (**formatear** xml)

Uso de MenúEclipste->Window/Preferences (aumentar **tamaño letra**)

/General/Appearance/Color and Fonts

/Basic/TextFont (Edit)

Contenido del archivo usando BD H2

https://www.h2database.com/html/quickstart.html#embedding

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

```
<persistence xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
            http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
            version="2.0" xmlns="http://java.sun.com/xml/ns/persistence">

  <persistence-unit name="Persistencia">
  <!-- Representamos las clases-->
    <properties>
        <property name="javax.persistence.jdbc.driver" value="org.h2.Driver" />
        <property name="javax.persistence.jdbc.url" value="jdbc:h2:mem:test" />
        <property name="javax.persistence.jdbc.user" value="sa" />
        <property name="javax.persistence.jdbc.password" value="" />

        <property name="hibernate.dialect" value="org." />
        <property name="hibernate.hbm2ddl.auto" value="" />
    </properties>
  </persistence-unit>

</persistence>
```

https://docs.jboss.org/hibernate/orm/3.5/javadocs/org/hibernate/dialect/package-summary.html

```
<property name="hibernate.dialect" value="org.hibernate.dialect.H2Dialect" />
```

https://docs.jboss.org/hibernate/orm/5.0/manual/en-US/html/ch03.html

```
 <property name="hibernate.hbm2ddl.auto" value="" />
```

e.g. validate | update | create | create-drop

Entidades

Crear un modelo:

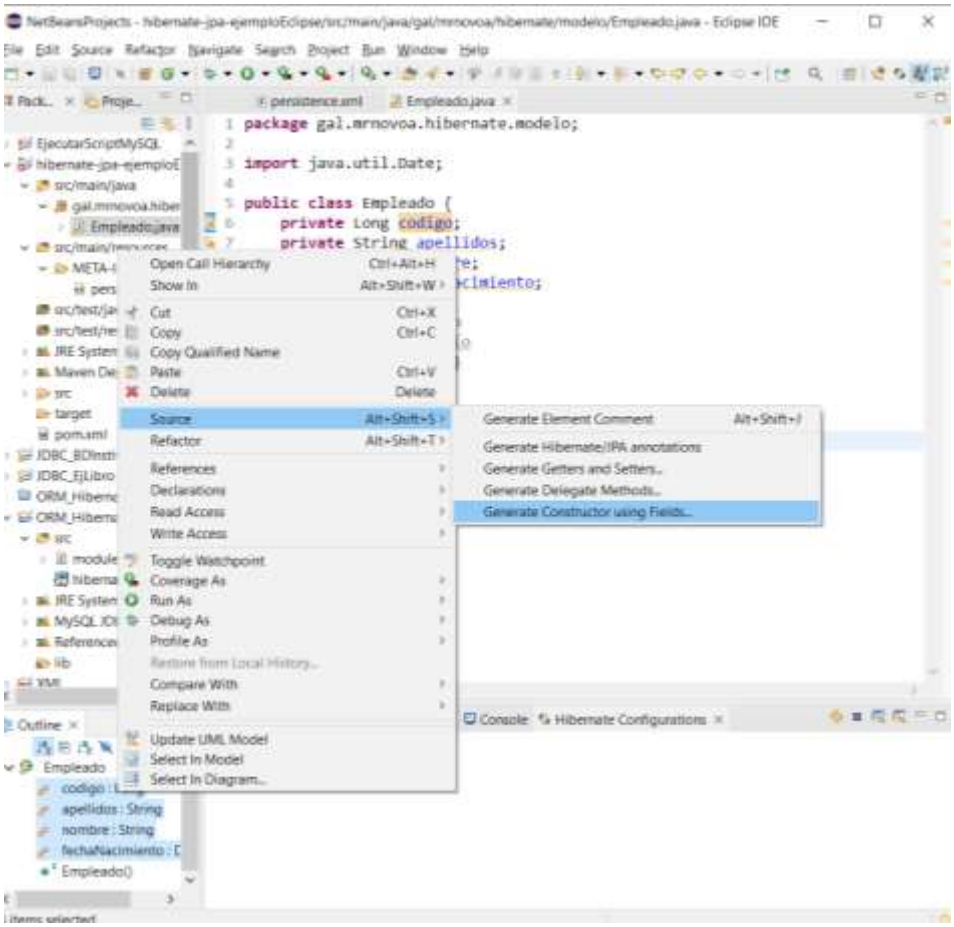
src/main/java

Crear la clase Empleado en un paquete determinado. Posteriormente se usará para jpa.

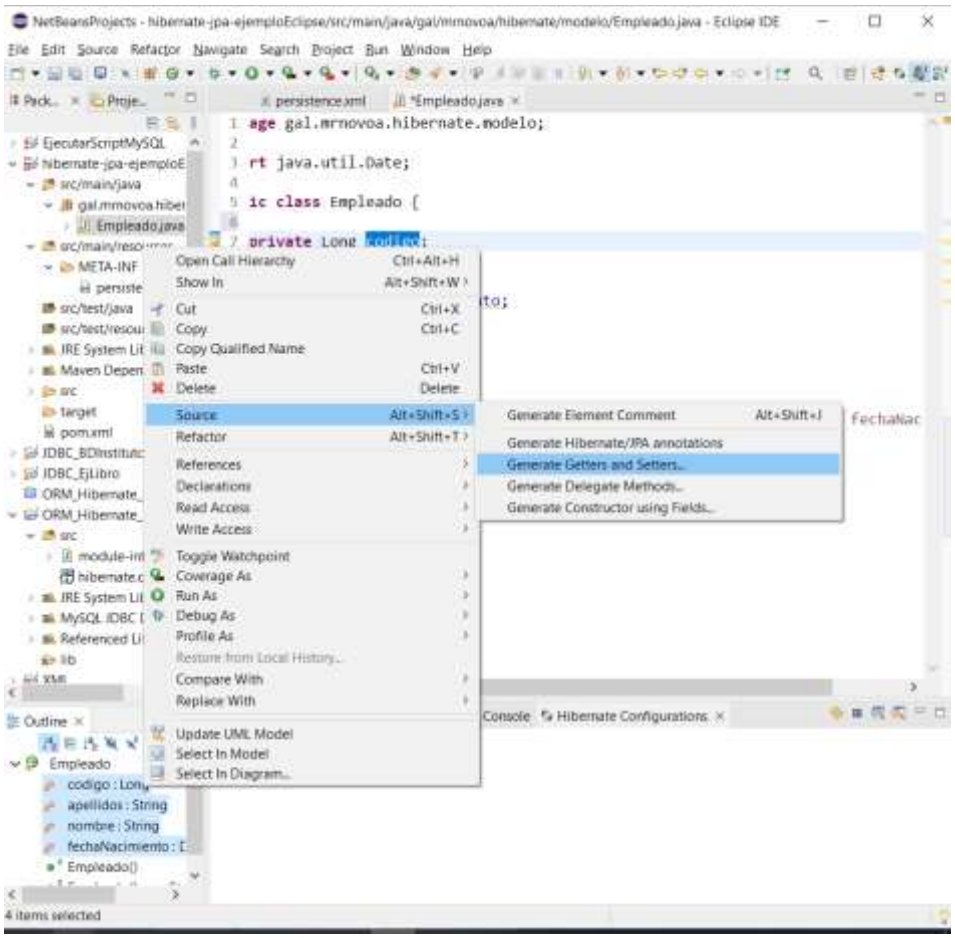
Convertir la clase en un **Bean**:

1. Constructor vacío (para crear campos…)

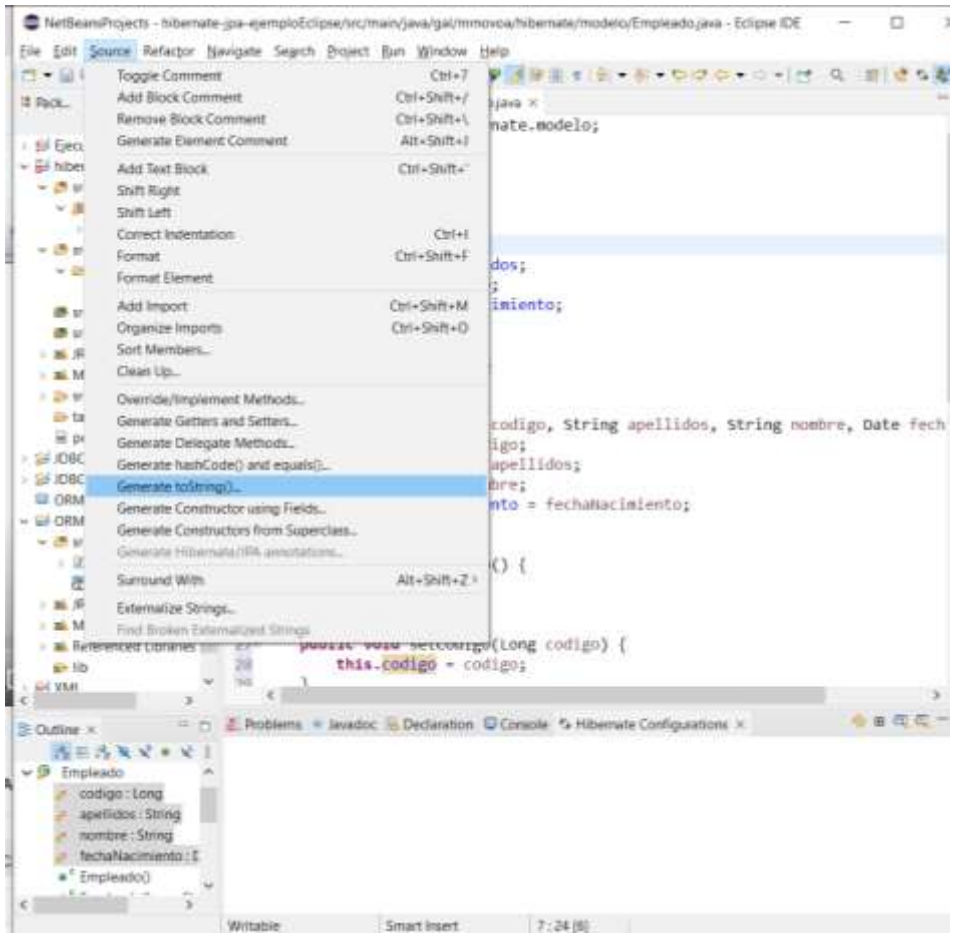
OJO. Se pueden añadir también los otros constructores necesarios:



2. Y también los getters y los setters:



3. Y el método toString()



4. Y por último que use Serializable

.. implements Serializable{

Reconocimiento **por jpa como Entity**:

@Entity– **javax.persistence**

```
import javax.persistence.Entity;
```

No usar el de Hibernate

Decirle a jpa con que se corresponde en la **BD**:

@Table

```
import javax.persistence.Table;
@Table (name = "EMPLEADO")
```

Para indicar la tabla específicamente

Y las columnas

```
@Column(name = "COD_EMPLEADO")
import javax.persistence.Column;
```

JPA obligará que tenga también un identificador:

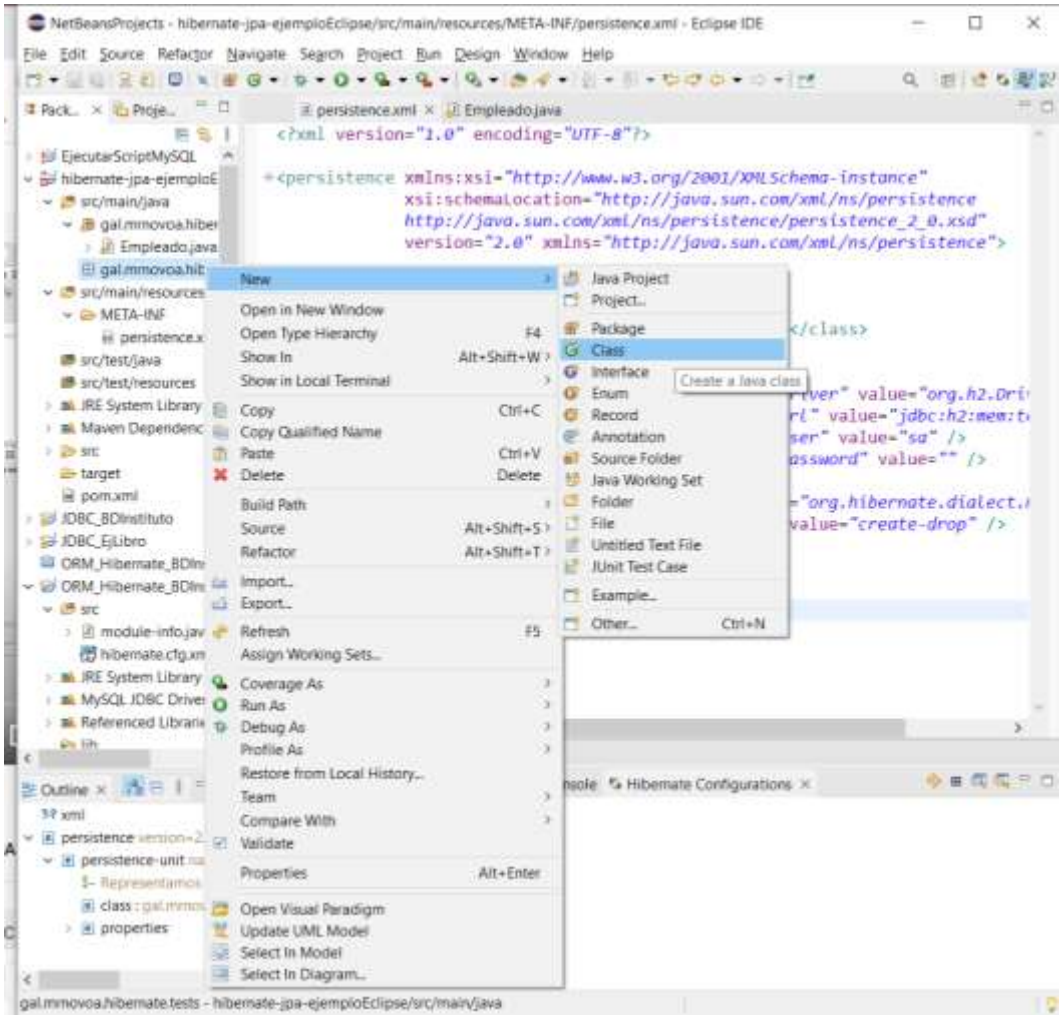
Persistencia

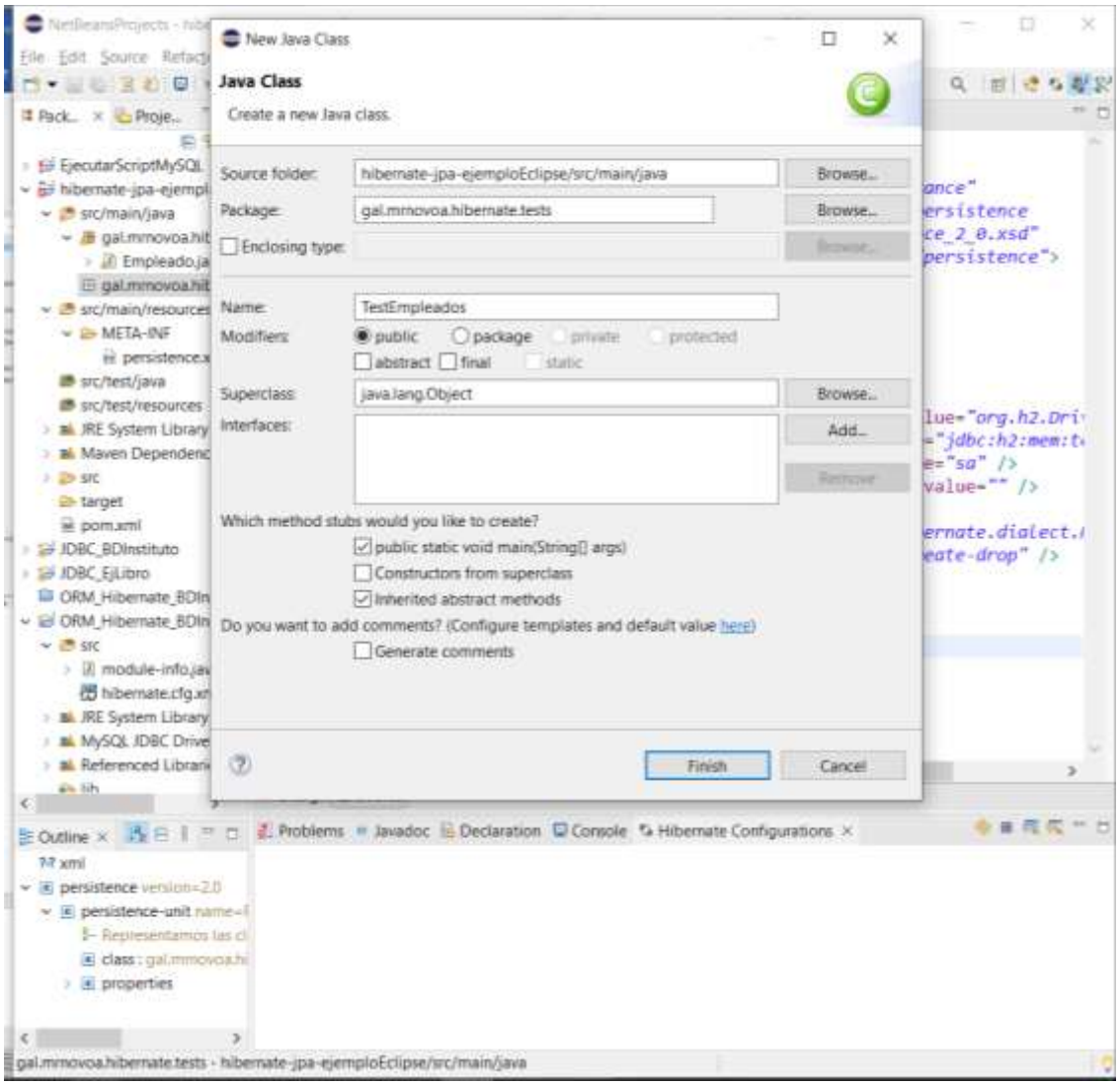
Entity Manager

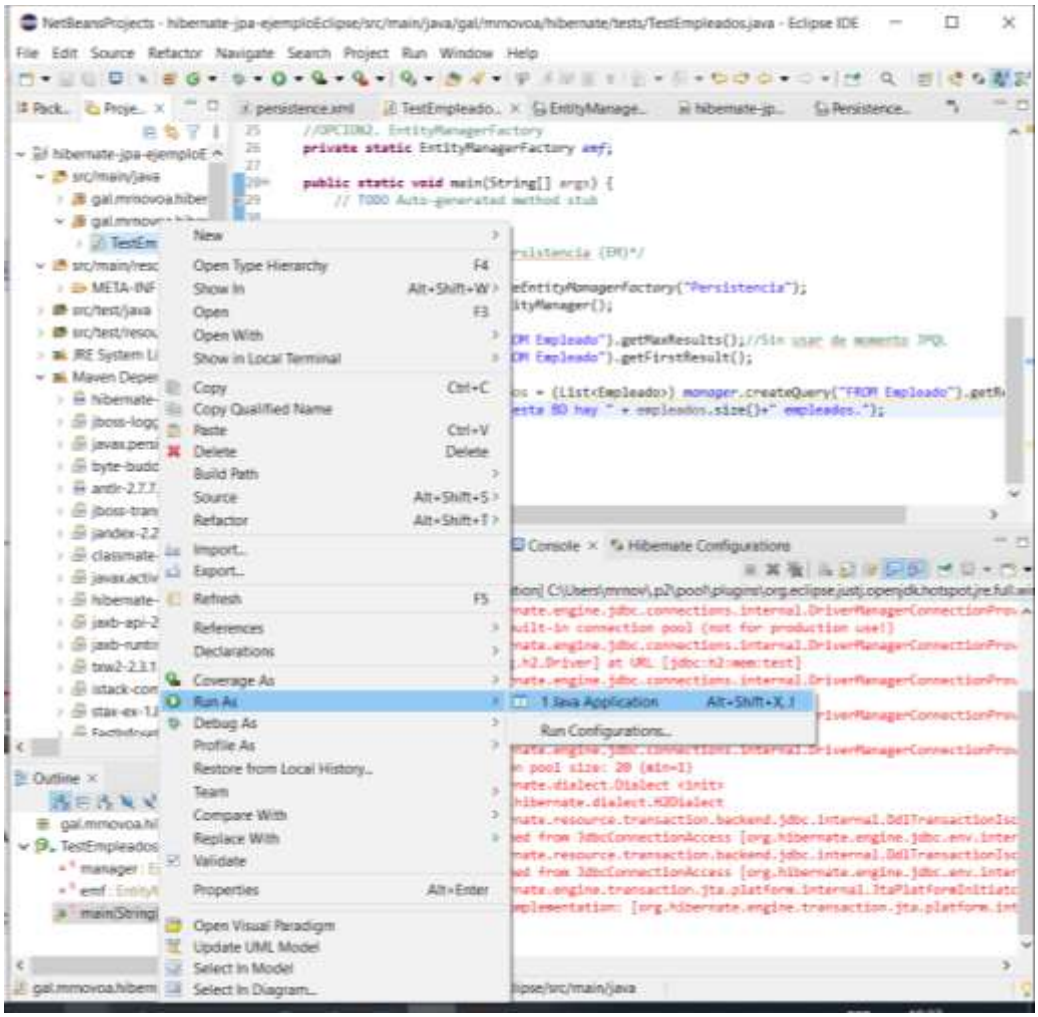
Registrar las Entities en el persistence.xml

<!-- Representamos las clases--
<class>gal.mrnovoa.hibernate.modelo.Empleado</class>

Generar un gestor de persistencia:




New Java Class
Java Class
Create a new Java class.
Source folder:
hibernate-jpa-ejemploEclipse/src/main/java
Browse...
Package:
gal.mnovoa.hibernate.tests
Browse...
Enclosing type:
Name:
TestEmpleados
Modifiers:
public
package
private
protected
abstract
final
static
Superclass:
java.lang.Object
Browse...
Interfaces:
Add...
Remove
Which method stubs would you like to create?
public static void main(String[] args)
Constructors from superclass
Inherited abstract methods
Do you want to add comments? (Configure templates and default value here)
Generate comments
Finish
Cancel
Outline
Problems
Javadoc
Declaration
Console
Hibernate Configurations
gal.mnovoa.hibernate.tests - hibernate-jpa-ejemploEclipse/src/main/java




```
emf = Persistence.createEntityManagerFactory("Persistencia");
manager = emf.createEntityManager();
```

USO de CRUD

Inserción

```
Empleado e = new Empleado(10L,"Perez","Pepito",new GregorianCalendar(1985,6,6).getTime());
                Empleado e2 = new Empleado(25L,"Rodriguez","Juan",new
GregorianCalendar(1986,10,1).getTime());

                //JPA más antiguo q Java8- Uso Calendar
                //Instalando ciertos elementos, Hibernate permite uso de Java8

                //INSERCION
                //TRANSACCIONES  - begin,..., commit
                //transaccion- posibilidad de hacer varias operaciones con la BD y no se envían/ejecutan
                //hasta hacer el commit(), y se puede hacer rollback() y dejar BD en estado anterior al
begin()
                //forma atómica- o se ejecuta todo o no se ejecuta nada

                manager.getTransaction().begin();

                manager.persist(e);
                manager.persist(e2);

                manager.getTransaction().commit();

                imprimirTodo();
```

```
	}


	@SuppressWarnings("unchecked")
	private static void imprimirTodo() {
		List <Empleado> emps = (List<Empleado>) manager.createQuery("FROM
Empleado").getResultList();
		System.out.println("En esta BD hay " + emps.size()+" empleados.");

		for(Empleado emp : emps) {
			System.out.println(emp.toString());
		}
	}
```

Managed Entities

Hacer una instancia administrada-managed.

//FORMA 1

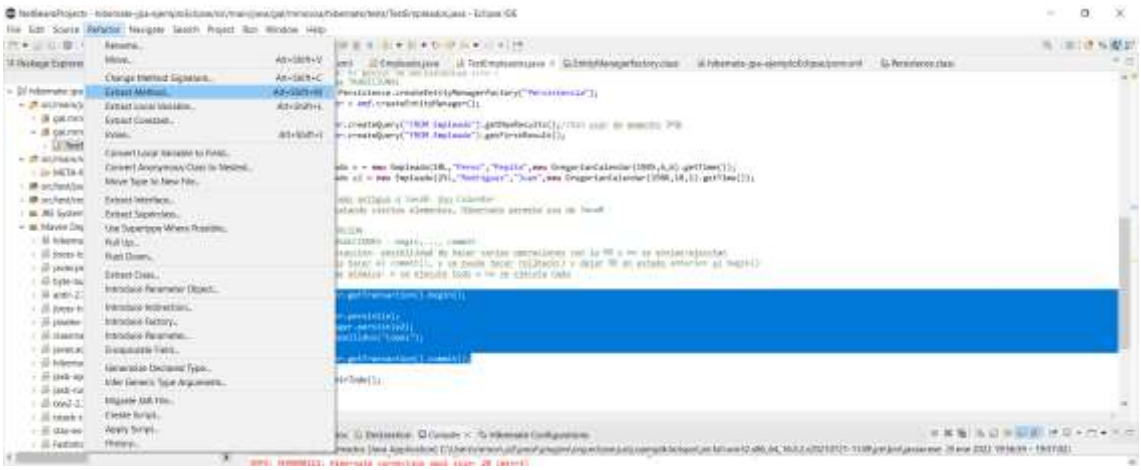
Que el gestor de Persistencia (EntityManager), ya tiene registrada la entidad y es interesante a la hora de realizar modificaciones

```
		manager.persist(e);
		//manager.persist(e2);
		e.setApellidos("Lopez");

		manager.getTransaction().commit();
```

El método .persist(e), convierte el objeto en administrado

Hay más formas

Crear nuevo método de inserción.



Uso método **find** **(en caso de que esté en estado managed)**

```
		manager.getTransaction().begin();
```

```
Empleado e1 = manager.find(Empleado.class, 10L);//también será managed por hacerlo bloqueTransaccion
e1.setNombre("Recaredo");
e1.setApellidos("Lopez");
manager.getTransaction().commit();
```

OJO. Optimizar recursos cerrando el manager cuando ya no se vaya a usar. Muy importante en peticiones web

```
private static EntityManagerFactory emf =
Persistence.createEntityManagerFactory("Persistencia");
```

```
imprimirTodo();
manager.close();
```

Si una Entidad está Detached, (pej si se ha cerrado sesión), se puede usar método **merge** – el objeto ahora pasará de **detached** a managed.(No es imprescindible su uso)

Merge y remove

Uso de sus propios EntityManager en cada operación:

```
public class TestEmpleados2{

....

private static void InsertInicial(){
		EntityManager man = emf.createEntityManager();

		Empleado e = new Empleado(10L,"Perez","Pepito",new
GregorianCalendar(1985,6,6).getTime());
		man.getTransaction().begin();

		man.persist(e);
		//manager.persist(e2);

		man.getTransaction().commit();

		man.close();
}
```

Merge

```
public static void main(String[] args){
		// TODO Auto-generated method stub

		EntityManager man = emf.createEntityManager();

		Empleado e = new Empleado(10L,"Perez","Pepito",new
GregorianCalendar(1986,10,1).getTime());

		man.getTransaction().begin();
		man.persist(e);
		man.getTransaction().commit();

		man.close();//la entidad e pasa de Managed a Detached (ya no está administrada)

		imprimirTodo();

		//Como ya no está managed -> al usar gestiones de persistencia distintos con begin() y
close()
		man = emf.createEntityManager();

		man.getTransaction().begin();

		//SOLUCIÓN FORMA2:
		e = man.merge(e);

		e.setNombre("Dani");//JPA no sabe que se refiere a la misma entidad anterior
		//SOLUCIÓN: uso de find() o merge()->mezcla entidad noManaged y pasa a Managed
		//SOLUCIÓN FORMA1:
		man.merge(e);

		man.getTransaction().commit();

		man.close();

		imprimirTodo();
	}
```

```
//SOLUCIÓN: uso de find() o merge()->mezcla entidad noManaged y pasa a Managed
        man.merge(e);
```

Con esta nueva línea pasó a ser Managed la entidad y el cambio de nombre se refleja.

Remove

```
man.remove(e);
```

Operaciones básicas JPA:

- persist(e)
- merge(e)
- remove(e)

Java 8

Ej uso de Fechas sin usar Date ni GregorianCalendar

Java.Time->Java 8

Mejores comprobaciones

JPA no los admitía en versión 2.2. Soluciones

- Migrar a JPA 3.0

  JPA 3.0 didn't add or change any functionality compared to JPA 2.2. It only changed the package from *javax.persistence* to *jakarta.persistence*. Due to that, you can simply replace all occurrences of "import javax.persistence" with "import jakarta.persistence".

```
<dependency>
    <groupId>org.eclipse.persistence</groupId>
    <artifactId>org.eclipse.persistence.jpa</artifactId>
    <version>3.0.1</version>
</dependency>
```

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core-jakarta</artifactId>
```

```
    <version>5.5.2</version>
</dependency>
```

```
import jakarta.persistence.Entity;
import jakarta.persistence.FetchType;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.ManyToOne;
import jakarta.persistence.Version;
```

persistence.xml configuration

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<persistence xmlns="https://jakarta.ee/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             version="3.0"
             xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence
https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd">
    <persistence-unit name="my-persistence-unit">
        ...
    </persistence-unit>
</persistence>
```

- Uso Hibernate -> https://mvnrepository.com/artifact/org.hibernate/hibernate-java8

```
        <!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-java8 -->
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-java8</artifactId>
    <version>5.6.1.Final</version>
</dependency>
```

Java8-specific Hibernate O/RM functionality has been merged into the hibernate-core module, making this hibernate-java8 module

obsolete. This module will be removed in Hibernate ORM 6.0. It is only kept here for various consumers that expect a
static set of artifact names across a number of Hibernate releases. See
https://hibernate.atlassian.net/browse/HHH-10883

Introducción Relaciones

Claves

TablesForáneas

Joins…

- JPAs Notaciones (la parte izqda.. siempre hace referencia a la Entity residente, la actual que se está codificando)

@OneToOne – Empleado->Dirección

@OneToMany - Empleado ->Facturas

- Pedido -> LíneasProducto

- Autor -> Libros

@ManyToOne

@ManyToMany – Relaciones con cardinalidad M:N

- Sistema de Acceso Relaciones

```
//Dependiendo de lo que tengamos q procesar->puede ser muy pesado
    //JPA->2 formas carga:
    //FORMA1 :EAGER - temprana (recuperar todos los datos a la vez desde principio)
    //FORMA2 :LAZY - perezosa (por defecto en JPA - sólo cuando se hace get)
```

- # Relación OneToOne

  Relación Objeto y sus elementos (ej. Empleado - Dirección)
  Relación de Objeto con su padre

```
package gal.mrnovoa.hibernate.modelo;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.OneToOne;
import javax.persistence.Table;

@Entity
@Table(name= "DIRECCION")
```

```java
public class Direccion {
	@Id
	@Column(name="ID_DIRECCION")
	private Long id;//Siempre para ser tratado como Entidad

	@Column(name="DIRECCION")
	private String direccion;

	@Column(name="LOCALIDAD")
	private String localidad;

	@Column(name="PROVINCIA")
	private String provincia;

	@Column(name="PAIS")
	private String pais;

	public Direccion() {}//Siempre para ser tratado como Entidad

	public Direccion(Long id, String direccion, String localidad, String provincia, String pais) {
		this.id = id;
		this.direccion = direccion;
		this.localidad = localidad;
		this.provincia = provincia;
		this.pais = pais;
	}

	public Long getId() {
		return id;
	}

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

	public String getDireccion() {
		return direccion;
	}

	public void setDireccion(String direccion) {
		this.direccion = direccion;
	}

	public String getLocalidad() {
		return localidad;
	}

	public void setLocalidad(String localidad) {
		this.localidad = localidad;
	}

	public String getProvincia() {
		return provincia;
	}

	public void setProvincia(String provincia) {
		this.provincia = provincia;
	}

	public String getPais() {
		return pais;
	}

	public void setPais(String pais) {
		this.pais = pais;
	}
```

```
	@Override
	public String toString() {
		return "Direccion [id=" + id + ", direccion=" + direccion + ", localidad=" + localidad + ",
provincia="
				+ provincia + ", pais=" + pais + "]";
	}
```

Empleado.java

```
	@OneToOne
	@JoinColumn(name = "ID_DIRECCION")//columna en la TablaEmpleado
	private Direccion direccion;
```

```
Exception in thread "main" javax.persistence.RollbackException: Error while committing the transaction
	at
org.hibernate.internal.ExceptionConverterImpl.convertCommitException(ExceptionConverterImpl.java:81)
	at org.hibernate.engine.transaction.internal.TransactionImpl.commit(TransactionImpl.java:104)
	at gal.mrnovoa.hibernate.tests.TestEmpleados2.main(TestEmpleados2.java:35)
Caused by: java.lang.IllegalStateException: org.hibernate.TransientPropertyValueException: object
references an unsaved transient instance - save the transient instance before flushing :
gal.mrnovoa.hibernate.modelo.Empleado.direccion -> gal.mrnovoa.hibernate.modelo.Direccion
	at org.hibernate.internal.ExceptionConverterImpl.convert(ExceptionConverterImpl.java:151)
	at org.hibernate.internal.ExceptionConverterImpl.convert(ExceptionConverterImpl.java:181)
	at org.hibernate.internal.ExceptionConverterImpl.convert(ExceptionConverterImpl.java:188)
	at org.hibernate.internal.SessionImpl.doFlush(SessionImpl.java:1420)
	at org.hibernate.internal.SessionImpl.managedFlush(SessionImpl.java:507)
	at org.hibernate.internal.SessionImpl.flushBeforeTransactionCompletion(SessionImpl.java:3299)
	at org.hibernate.internal.SessionImpl.beforeTransactionCompletion(SessionImpl.java:2434)
	at
org.hibernate.engine.jdbc.internal.JdbcCoordinatorImpl.beforeTransactionCompletion(JdbcCoordinatorImp
l.java:449)
	at
org.hibernate.resource.transaction.backend.jdbc.internal.JdbcResourceLocalTransactionCoordinatorImpl.b
eforeCompletionCallback(JdbcResourceLocalTransactionCoordinatorImpl.java:183)
	at
org.hibernate.resource.transaction.backend.jdbc.internal.JdbcResourceLocalTransactionCoordinatorImpl.a
ccess$300(JdbcResourceLocalTransactionCoordinatorImpl.java:40)
	at
org.hibernate.resource.transaction.backend.jdbc.internal.JdbcResourceLocalTransactionCoordinatorImpl$
TransactionDriverControlImpl.commit(JdbcResourceLocalTransactionCoordinatorImpl.java:281)
	at org.hibernate.engine.transaction.internal.TransactionImpl.commit(TransactionImpl.java:101)
	... 1 more
Caused by: org.hibernate.TransientPropertyValueException: object references an unsaved transient
instance - save the transient instance before flushing : gal.mrnovoa.hibernate.modelo.Empleado.direccion
-> gal.mrnovoa.hibernate.modelo.Direccion
	at org.hibernate.engine.spi.CascadingActions$8.noCascade(CascadingActions.java:379)
	at org.hibernate.engine.internal.Cascade.cascade(Cascade.java:169)
	at
org.hibernate.event.internal.AbstractFlushingEventListener.cascadeOnFlush(AbstractFlushingEventListen
er.java:159)
	at
org.hibernate.event.internal.AbstractFlushingEventListener.prepareEntityFlushes(AbstractFlushingEventLi
stener.java:149)
	at
org.hibernate.event.internal.AbstractFlushingEventListener.flushEverythingToExecutions(AbstractFlushing
EventListener.java:82)
	at
org.hibernate.event.internal.DefaultFlushEventListener.onFlush(DefaultFlushEventListener.java:39)
	at
org.hibernate.event.service.internal.EventListenerGroupImpl.fireEventOnEachListener(EventListenerGrou
pImpl.java:107)
	at org.hibernate.internal.SessionImpl.doFlush(SessionImpl.java:1416)
	... 9 more
```

JPA aún no tiene el valor:

- FORMA1

```
Direccion d = new Direccion(15L,"Calle Falsa,123","Springfield","Springfield","EEUU");
                Empleado e = new
Empleado(10L,"Perez","Pepito",LocalDate.of(1986,Month.OCTOBER,1));
                //Relacion 1ToMany

                e.setDireccion(d);

                man.getTransaction().begin();

                man.persist(d);
                man.persist(e);

                man.getTransaction().commit();
```

- FORMA2
  - JPA-mecanismo en cascada (inserción, borrado..)

```
Empleado e = new Empleado(10L,"Perez","Pepito",LocalDate.of(1986,Month.OCTOBER,1));

                e.setDireccion(new Direccion(15L,"Calle Falsa,123","Springfield","Springfield","EEUU"));
```

```
    @OneToOne(cascade = {CascadeType.ALL})//para que la actualización de dirección se propague en Insert/Delete..
```

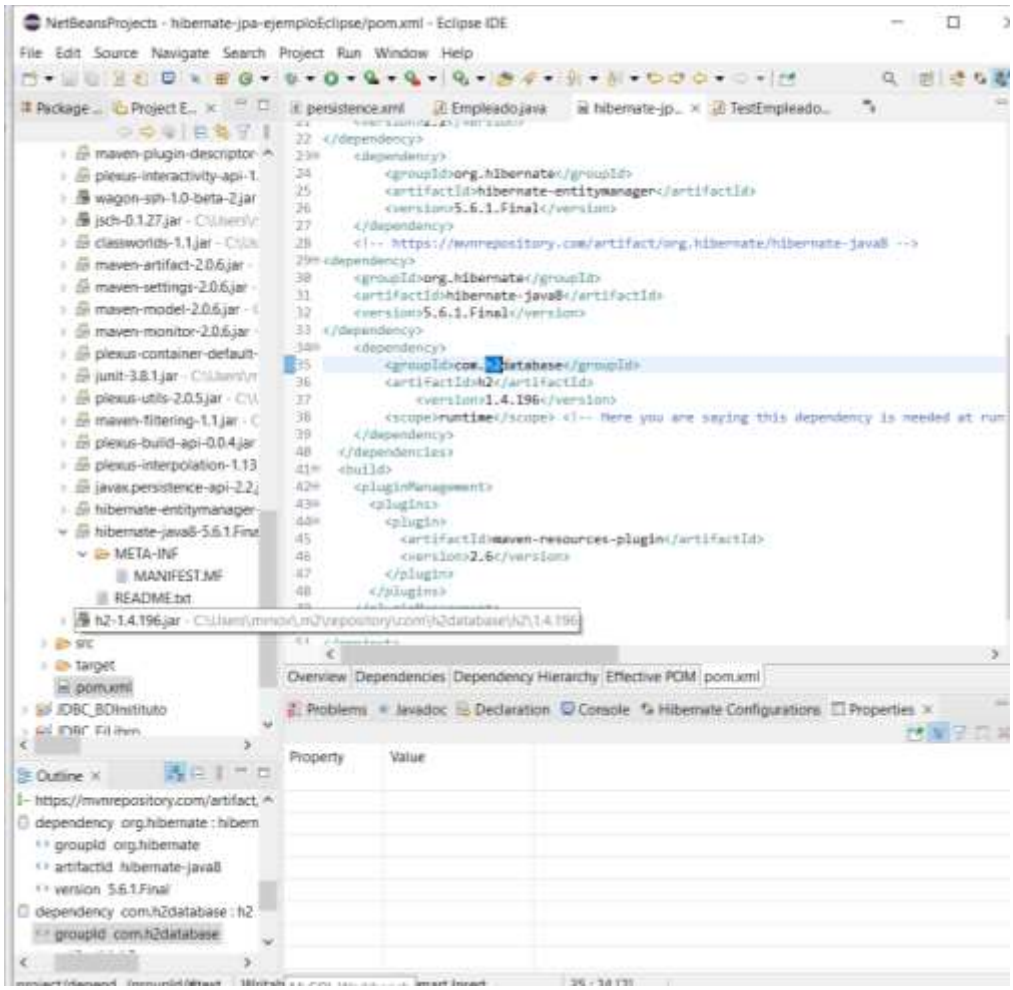
OJO. Asegurarse siempre que los objetos dependientes lleven la notación **cascade**

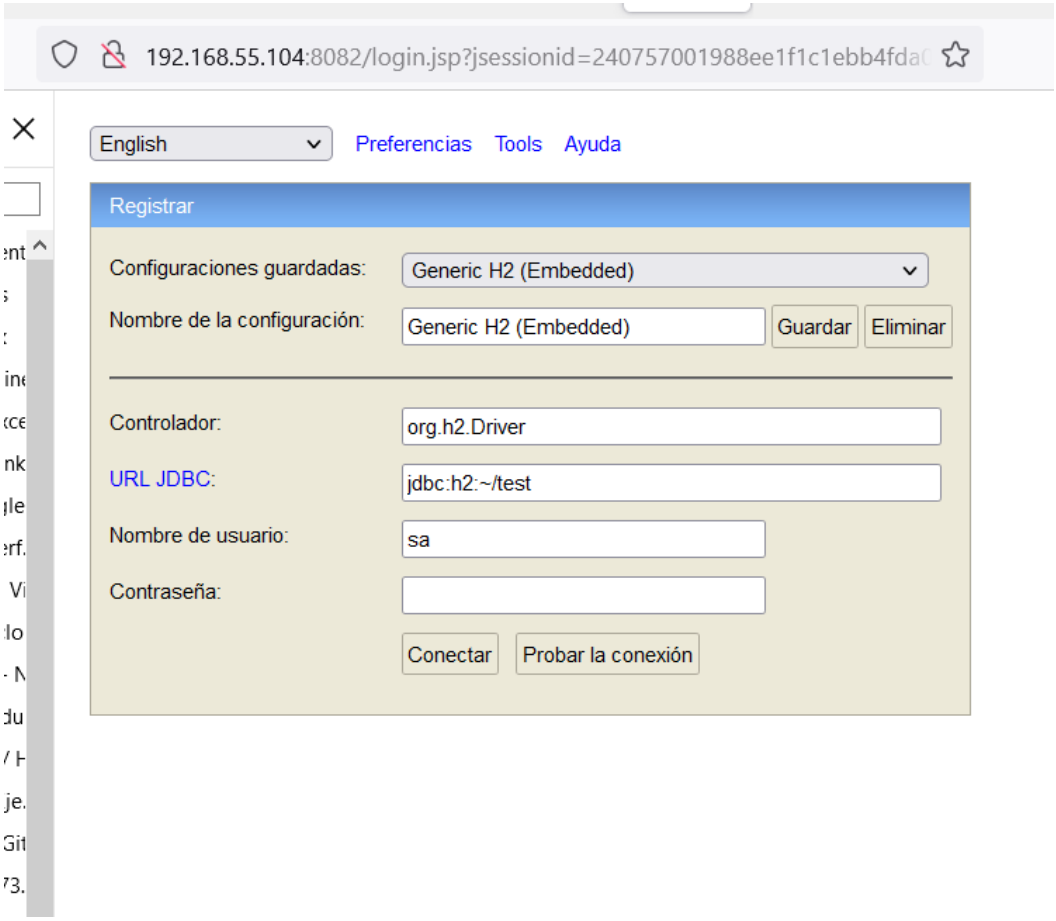
OneToOne inverso con mappedBy

Ej. Que la relación Empleado vs. Dirección sea **bidireccional**, es decir, que sabiendo la dirección se pueda obtener el empleado que la tiene asignada.

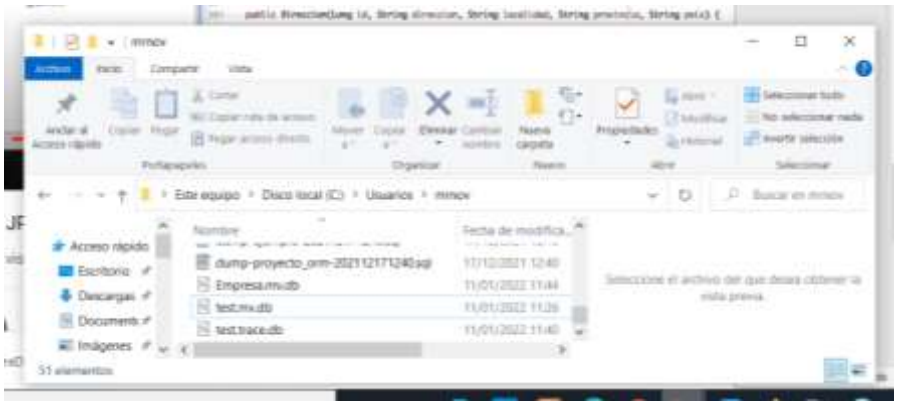
```
            <property name="javax.persistence.jdbc.driver" value="org.h2.Driver" />
              <!--Para que guarde en Memoria  -->
              <!--  <property name="javax.persistence.jdbc.url" value="jdbc:h2:mem:test" />-->

              <!--Para que guarde en un archivo llamado Empresa ubicado en el escritorio con extensión Empresa.mv.db-->
              <property name="javax.persistence.jdbc.url" value="jdbc:h2:~/Desktop/Empresa"  />
```

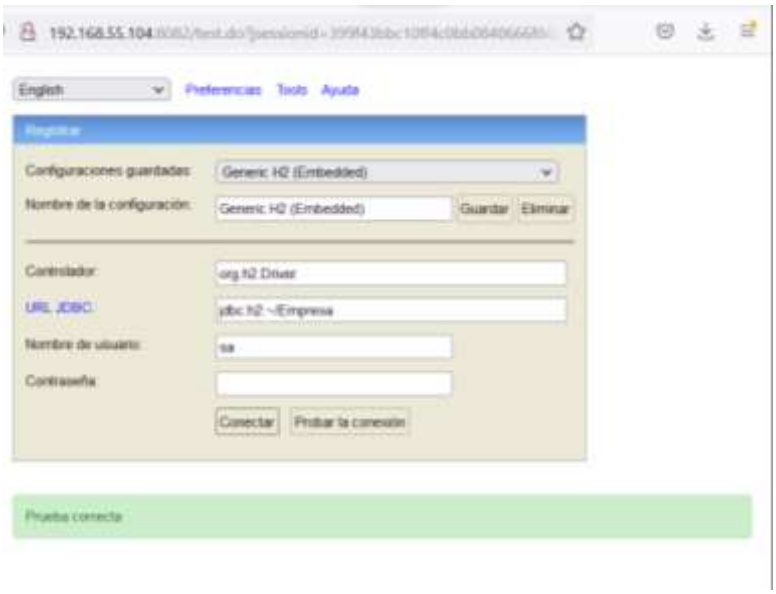
C:\Users\mrnov\.m2\repository\com\h2database\h2\1.4.200 o 1.4.196

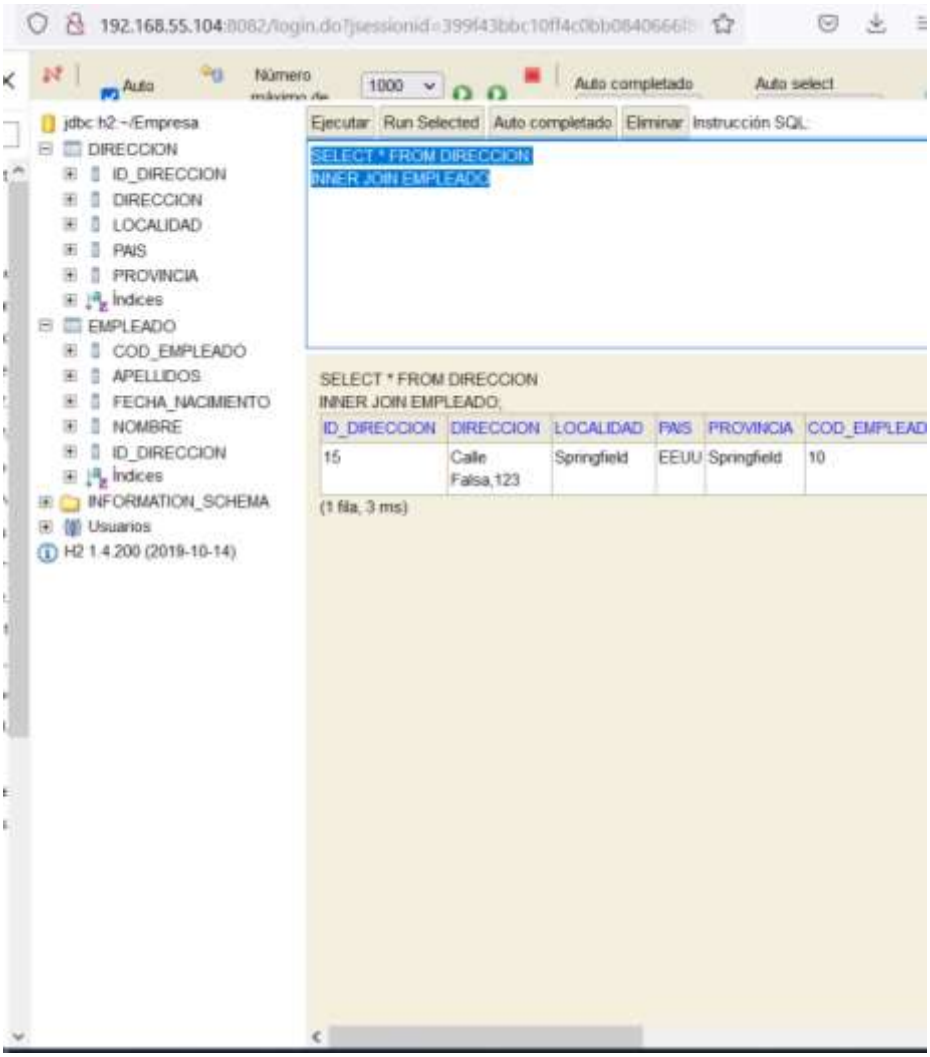

NetBeansProjects - hibernate-jpa-ejemploEclipse/pom.xml - Eclipse IDE
File Edit Source Navigate Search Project Run Window Help
Package ... Project E...
persistence.xml Empleado.java hibernate-jp... TestEmpleado...
maven-plugin-descriptor
plexus-interactivity-api-1.
wagon-ssh-1.0-beta-2.jar
jsch-0.1.27.jar
classworlds-1.1.jar
maven-artifact-2.0.6.jar
maven-settings-2.0.6.jar
maven-model-2.0.6.jar
maven-monitor-2.0.6.jar
plexus-container-default-
junit-3.8.1.jar
plexus-utils-2.0.5.jar
maven-filtering-1.1.jar
plexus-build-api-0.0.4.jar
plexus-interpolation-1.13
javax.persistence-api-2.2.
hibernate-entitymanager-
hibernate-java8-5.6.1.Fina
META-INF
MANIFEST.MF
README.txt
h2-1.4.196.jar
src
target
pom.xml
JDBC_BDInstituto
Outline
https://mvnrepository.com/artifact,
dependency org.hibernate : hibern
groupId org.hibernate
artifactId hibernate-java8
version 5.6.1.Final
dependency com.h2database : h2
groupId com.h2database
<dependency>
<groupId>org.hibernate</groupId>
<artifactId>hibernate-entitymanager</artifactId>
<version>5.6.1.Final</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-java8 -->
<dependency>
<groupId>org.hibernate</groupId>
<artifactId>hibernate-java8</artifactId>
<version>5.6.1.Final</version>
</dependency>
<dependency>
<groupId>com.h2database</groupId>
<artifactId>h2</artifactId>
<version>1.4.196</version>
<scope>runtime</scope> <!-- Here you are saying this dependency is needed at run
</dependency>
</dependencies>
<build>
<pluginManagement>
<plugins>
<plugin>
<artifactId>maven-resources-plugin</artifactId>
<version>2.6</version>
</plugin>
</plugins>
Overview Dependencies Dependency Hierarchy Effective POM pom.xml
Problems Javadoc Declaration Console Hibernate Configurations Properties
Property Value



Si estuviese directamente en el home del usuario:







SELECT * FROM DIRECCION

INNER JOIN EMPLEADO

Direccion.java

```
@OneToOne(mappedBy = "direccion",fetch = FetchType.LAZY)//nombre del campo en la clase Empleado
	private Empleado empleado;
```

```
En esta BD hay 1 empleados.
Empleado [codigo=10, apellidos=Perez, nombre=Pepito, fechaNacimiento=1986-10-01, direccion=Direccion [id=15, direccion=Calle Falsa,123, localidad=Springfield, provincia=Springfield, pais=EEUU, empleado=10]]
```

OneToMany: Planteamiento

Relaciones cardinalidad 1:N

Ej. Cliente 0..N Facturas

Libro-Autor

Crear las correspondientes Entities.

Libro tendrá referencia al Autor que lo escribió

Autor tendrá una colección de libros que ha escrito dicho autor

OneToMany: Anotaciones

@OneToMany

```
package gal.mrnovoa.hibernate.modelo;

import java.util.ArrayList;
import java.util.List;

import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import javax.persistence.Table;

@Entity
@Table(name="AUTOR")
public class Autor {

	@Id
	@Column(name="AUTOR_ID")
	private Long id;

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

	@Column(name="NACIONALIDAD")
	private String nacionalidad;
```

```
	//1Autor - NLibros
	@OneToMany(mappedBy = "autor", cascade = CascadeType.ALL)//mappedBy - Nombre del campoUnion en la otra Entity

	                    //cascadeType.ALL - pej. para que se guarden los libros automáticamente cuando se creal el autor

	                   //fetch - no será necesario, ya que por defecto es Lazy en @OneToOne y @OneToMany
	private List<Libro> libros = new ArrayList<Libro>();

	public Autor() {}

	public Autor(Long id, String nombre, String nacionalidad) {
		this.id = id;
		this.nombre = nombre;
		this.nacionalidad = nacionalidad;
	}
	public Long getId() {
		return id;
	}
	public void setId(Long id) {
		this.id = id;
	}
	public String getNombre() {
		return nombre;
	}
	public void setNombre(String nombre) {
		this.nombre = nombre;
	}
	public String getNacionalidad() {
		return nacionalidad;
	}
	public void setNacionalidad(String nacionalidad) {
		this.nacionalidad = nacionalidad;
	}

	public List<Libro> getLibros() {
		return libros;
	}

	public void setLibros(List<Libro> libros) {
		this.libros = libros;
	}

	@Override
	public String toString() {
		return "Autor [id=" + id + ", nombre=" + nombre + ", nacionalidad=" + nacionalidad + "]";
	}


}



package gal.mrnovoa.hibernate.modelo;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.Table;
```

```java
@Entity
@Table(name="LIBROS")
public class Libro {
	@Id
	@Column(name="LIBRO_ID")
	private Long id;

	@Column(name="TITULO")
	private String titulo;

	//NLibros - 1 Autor
	@ManyToOne(fetch = FetchType.LAZY)//Hay que indicarlo. Para que un autor no se carguen sus datos hasta que se ejecute un get
	@JoinColumn(name = "AUTOR_ID")//Nexo de unión entre las 2 Entities
	private Autor autor;

	public Libro() {}

	public Libro(Long id, String titulo, Autor autor) {
		super();
		this.id = id;
		this.titulo = titulo;
		this.autor = autor;
	}

	public Long getId() {
		return id;
	}

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

	public String getTitulo() {
		return titulo;
	}

	public void setTitulo(String titulo) {
		this.titulo = titulo;
	}

	public Autor getAutor() {
		return autor;
	}

	public void setAutor(Autor autor) {
		this.autor = autor;
	}

	@Override
	public int hashCode() {
		final int prime = 31;
		int result = 1;
		result = prime * result + ((autor == null) ? 0 : autor.hashCode());
		result = prime * result + ((id == null) ? 0 : id.hashCode());
		result = prime * result + ((titulo == null) ? 0 : titulo.hashCode());
		return result;
	}

	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
			return false;
```

```
		if (getClass() != obj.getClass())
			return false;
		Libro other = (Libro) obj;
		if (autor == null) {
			if (other.autor != null)
				return false;
		} else if (!autor.equals(other.autor))
			return false;
		if (id == null) {
			if (other.id != null)
				return false;
		} else if (!id.equals(other.id))
			return false;
		if (titulo == null) {
			if (other.titulo != null)
				return false;
		} else if (!titulo.equals(other.titulo))
			return false;
		return true;
	}

	@Override
	public String toString() {
		return "Libro [id=" + id + ", titulo=" + titulo + ", autor=" + autor +
"]";
	}


}
```

OneToMany:EntityManager

```
package gal.mrnovoa.hibernate.tests;

import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

import gal.mrnovoa.hibernate.modelo.Autor;
import gal.mrnovoa.hibernate.modelo.Libro;

public class TestAutores {

	private static EntityManagerFactory emf =
Persistence.createEntityManagerFactory("Persistencia");

	public static void main(String[] args) {
		// TODO Auto-generated method stub

		crearDatos();
		imprimirDatos();
	}


	private static void crearDatos() {
		// TODO Auto-generated method stub
		EntityManager em = emf.createEntityManager();
```

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

		Autor autor1 = new Autor (1L, "Pablo Pérez", "Española");
		Autor autor2 = new Autor (2L, "Elena Gómez", "Mexicana");
		Autor autor3 = new Autor (3L, "Miguel López", "Chilena");

		em.persist(autor1);
		em.persist(autor2);
		em.persist(autor3);

		em.persist( new Libro (1L, "Programar en Java es fácil", autor2));
		em.persist( new Libro (2L, "Cómo vestirse con estilo", autor3));
		em.persist( new Libro (3L, "Cómo cocinar sin quemar la cocina",
autor1));
		em.persist( new Libro (4L, "Programar en Cobol es divertido", autor2));
		em.persist( new Libro (5L, "Programar en Cobol no es divertido",
autor2));

		em.getTransaction().commit();

		em.close();

	}

	private static void imprimirDatos() {
		// TODO Auto-generated method stub
		EntityManager em = emf.createEntityManager();

		Autor autor = em.find(Autor.class, 3L);
		List<Libro> libros = autor.getLibros();
		for (Libro libro:libros) {
			System.out.println("*"+libro.toString());
		}

		System.out.println(autor);

		em.close();
	}

}
```

ERRORES. En relaciones OneToMany o ManyToOne

- Error Lazy Initialization (OneToMany)

```
	private static void imprimirDatos() {
		// TODO Auto-generated method stub
		EntityManager em = emf.createEntityManager();

		Autor autor = em.find(Autor.class, 2L);
		List<Libro> libros = autor.getLibros();
		em.close();

		for (Libro libro:libros) {
			System.out.println("*"+libro.toString());
```

```
            }

            System.out.println(autor);


        }
```

```
Exception in thread "main" org.hibernate.LazyInitializationException: failed to
lazily initialize a collection of role: gal.mrnovoa.hibernate.modelo.Autor.libros,
could not initialize proxy - no Session
        at
org.hibernate.collection.internal.AbstractPersistentCollection.throwLazyInitializatio
nException(AbstractPersistentCollection.java:612)
        at
org.hibernate.collection.internal.AbstractPersistentCollection.withTemporarySessionIf
Needed(AbstractPersistentCollection.java:218)
        at
org.hibernate.collection.internal.AbstractPersistentCollection.initialize(AbstractPer
sistentCollection.java:591)
        at
org.hibernate.collection.internal.AbstractPersistentCollection.read(AbstractPersisten
tCollection.java:149)
        at
org.hibernate.collection.internal.PersistentBag.iterator(PersistentBag.java:387)
        at gal.mrnovoa.hibernate.tests.TestAutores.imprimirDatos(TestAutores.java:58)
        at gal.mrnovoa.hibernate.tests.TestAutores.main(TestAutores.java:20)
```

Solución 1

```
Autor autor = em.find(Autor.class, 2L);
            List<Libro> libros = autor.getLibros();

            libros.size();//activar de nuevo, para que el Lazy se espabile

Solución típica al estar haciendo CRUD en Spring

            em.close();
```

Solución 2

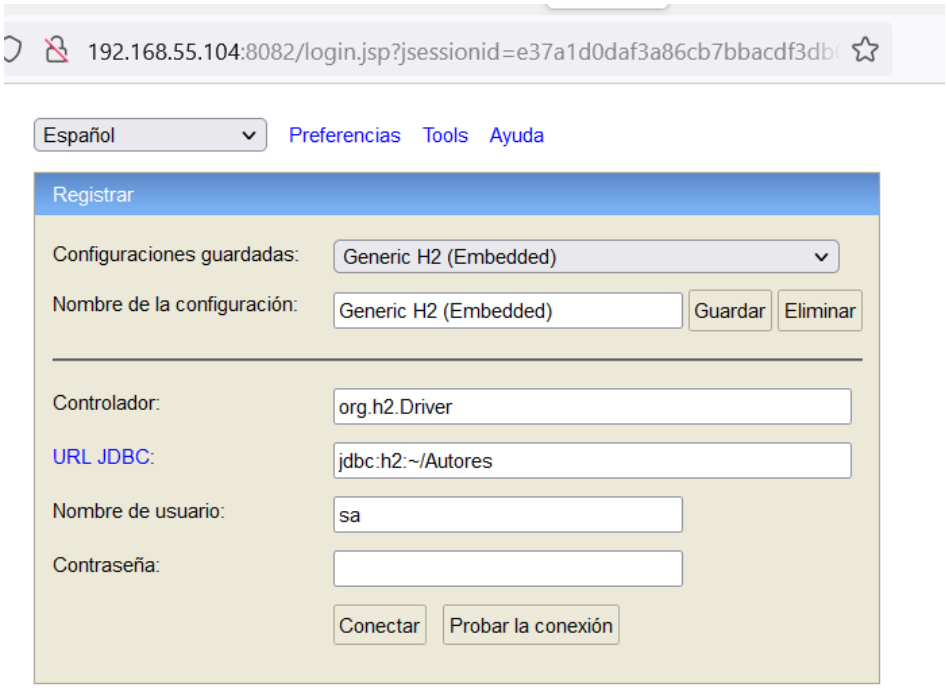
```
@OneToMany(mappedBy = "autor", cascade = CascadeType.ALL,
fetch=FetchType.EAGER)//OJO.No recomendable...para solucionar que el lazy no cargue
datos necesarios

        private List<Libro> libros = new ArrayList<Libro>();
```

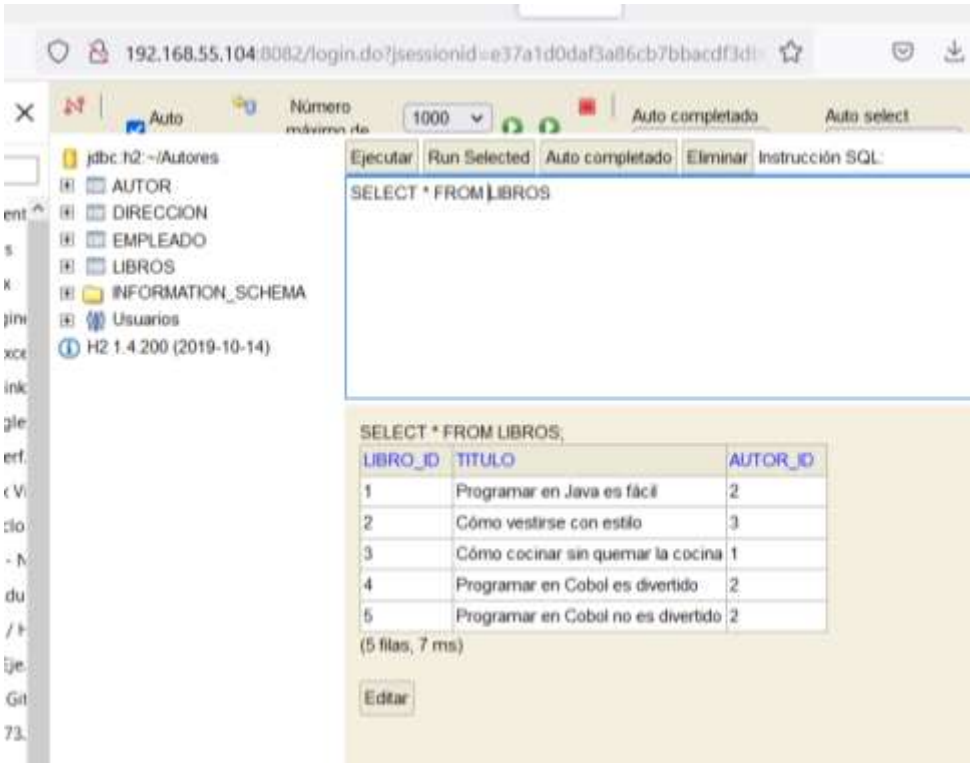
No recomendado, ya que puede generar problemas de rendimiento.

- No se fijan las relaciones (OneToMany)

Uso guardado de BD en un fichero Autores

```
<property name="javax.persistence.jdbc.url" value="jdbc:h2:~/Autores" />
```






```
Libro l1 = new Libro();
              l1.setId(1L);
              l1.setTitulo("JPA e Hibernate");
              em.persist(l1);

              Autor a1 = new Autor(1L, "Dani", "Española");
              a1.setLibros(Arrays.asList(l1));
```

```
		System.out.println("Libros escritos (pre-save): "
+a1.getLibros().size());
		em.persist(a1);

		em.getTransaction().commit();

	em.close();
```

```
Libros escritos (pre-save): 1
Libros escritos (post-sabe)0
```

OJO. JPA espera que se diga a cada Libro quién es su autor..como en la implementación del paso anterior

Java espera que todas las relaciones sean implementadas expresamente por el programador.

OPCION1

```
//Autor.java
//Solución a asignar a cada libro su autor
	public void setLibros(List<Libro> libros) {
		this.libros = libros;

		for(Libro l:libros) {
			l.setAutor(this);
		}
	}
```

OPCION2

```
//Autor.java
//Solución2 a asignar a cada libro su autor
	public void addLibro(Libro l) {
		if(!libros.contains(l)) {
			libros.add(l);
			l.setAutor(this);
		}
	}


//TestAutores.java


Autor a1 = new Autor(1L, "Dani", "Española");
		//a1.setLibros(Arrays.asList(l1));
	a1.addLibro(l1);
```

BORRADOS

- Borrar hijos en un OneToMany

Ej. Publicacion vs Comentario

```
package gal.mrnovoa.hibernate.modelo;


import java.util.ArrayList;

import java.util.List;


import javax.persistence.CascadeType;

import javax.persistence.Column;

import javax.persistence.Entity;

import javax.persistence.Id;

import javax.persistence.OneToMany;

import javax.persistence.Table;


@Entity

@Table(name="PUBLICACION")

public class Publicacion {

	@Id

	@Column(name="PUBLICACION_ID")

	private Long id;


	@Column (name= "TITULO")

	private String titulo;


	@OneToMany(mappedBy = "publicacion",cascade=CascadeType.ALL,orphanRemoval=true)

	//orphanRemoval- en eliminación.indica que no puede existir un comentario que no pertenezca a una publicación

	//el cascade actúa en la publicación, no en el comentario que quedaría colgando.

	private List<Comentario> comentarios = new ArrayList<Comentario>();


	public Publicacion() {

		super();

	}
```

```
public Publicacion(Long id, String titulo) {
	super();
	this.id = id;
	this.titulo = titulo;
}

public Long getId() {
	return id;
}

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

public String getTitulo() {
	return titulo;
}

public void setTitulo(String titulo) {
	this.titulo = titulo;
}
public void insertarComentario(Comentario c1) {
	if(!comentarios.contains(c1)) {
		comentarios.add(c1);
		c1.setPublicacion(this);
	}
}

//borrado - desasignar a cada comentario la publicacion
public void eliminarComentario(Comentario c1) {
	if (comentarios.contains(c1)) {
		comentarios.remove(c1);
		c1.setPublicacion(null);
	}
}
```

```
	public List<Comentario> getComentarios() {
		return comentarios;
	}

	public void setComentarios(List<Comentario> comentarios) {
		this.comentarios = comentarios;
	}

	@Override
	public String toString() {
		return "Publicacion [id=" + id + ", titulo=" + titulo + "]";
	}
}
```

```java
package gal.mrnovoa.hibernate.modelo;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.Table;

@Entity
@Table(name="COMENTARIO")
public class Comentario {

	@Id
	@Column(name="COMENTARIO_ID")
	private Long id;

	@Column(name="MENSAJE")
	private String mensaje;

	@ManyToOne(fetch = FetchType.LAZY)
	//@JoinColumn //no tiene sentido si se quiere permitir comentarios descolgados-violacion 1FN
				//más coherente usar JoinTable
	@JoinColumn(name = "PUBLICACION_ID")//Nexo de unión entre las 2 Entities
	private Publicacion publicacion;

	public Comentario() {
		super();
	}

	public Comentario(Long id, String mensaje) {
		this.id = id;
		this.mensaje = mensaje;
	}

	@Override
	public String toString() {
		return "Comentario [id=" + id + ", mensaje=" + mensaje + ", publicacion=" + publicacion + "]";
	}

	public Long getId() {
		return id;
	}

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

	public String getMensaje() {
		return mensaje;
	}

	public void setMensaje(String mensaje) {
		this.mensaje = mensaje;
	}

	public Publicacion getPublicacion() {
		return publicacion;
	}

	public void setPublicacion(Publicacion publicacion) {
		this.publicacion = publicacion;
	}	}
```

```
package gal.mrnovoa.hibernate.tests;


import java.util.List;


import javax.persistence.EntityManager;

import javax.persistence.EntityManagerFactory;

import javax.persistence.Persistence;


import gal.mrnovoa.hibernate.modelo.Autor;

import gal.mrnovoa.hibernate.modelo.Comentario;

import gal.mrnovoa.hibernate.modelo.Libro;

import gal.mrnovoa.hibernate.modelo.Publicacion;


public class TestEjBlog_borrado {

	static EntityManagerFactory emf =
Persistence.createEntityManagerFactory("Persistencia");

	public static void main(String[] args) {

		// TODO Auto-generated method stub

		insertar();

		imprimir();

		borrar(2L);

		imprimir();

		emf.close();

	}

	private static void borrar(long comentario) {

		// TODO Auto-generated method stub

		EntityManager em = emf.createEntityManager();

		em.getTransaction().begin();;


		Comentario c = em.find(Comentario.class, comentario);

		//opcion1

		//Publicacion pub = c.getPublicacion();

		//pub.eliminarComentario(c);

		//opcion2

		//em.remove(c);
```

```
		//OJO. El cascade funciona al elimnar padre-Publicacion, eliminando
todos los comentarios asociados
		em.remove(c.getPublicacion());

		em.getTransaction().commit();
		em.close();
	}

	private static void insertar() {
		// TODO Auto-generated method stub
		Publicacion p = new Publicacion(1L, "Hoy hace sol");
		Comentario c1 = new Comentario (1L, "Aqui llueve");

		Comentario c2 = new Comentario(2L, "eres kk");
		p.insertarComentario(c1);
		p.insertarComentario(c2);

		EntityManager em = emf.createEntityManager();
		em.getTransaction().begin();

		em.persist(p);

		em.getTransaction().commit();
		em.close();

	}

	private static void imprimir() {
		// TODO Auto-generated method stub
		EntityManager em = emf.createEntityManager();
		System.out.println("ENTRADAS DE BLOG");

		List<Publicacion> pubs = em.createQuery("FROM
Publicacion").getResultList();
```

```
		if(pubs.isEmpty())
			System.out.println("Sin entradas");
		else System.out.println(pubs.size()+" entradas");

		for(Publicacion publi:pubs) {

			System.out.println("\nPublicacion: "+ publi.toString());
			if(publi.getComentarios().isEmpty())
				System.out.println("Sin comentarios");
			else {
				List<Comentario> comentarios = publi.getComentarios();

				System.out.println("Comentarios escritos (post-
sabe)"+comentarios.size());
				for (Comentario c:comentarios) {
					System.out.println("*Comentario:"+c.toString());
				}
			}
		}

		em.close();
	}

}
```

- ON DELETE SET NULL

@JoinTable

Añadir usuario que hizo publicación

Que se pueda eliminar el usuario pero mantener las publicaciones

@ManyToMany

Hibernate Search

- Full-text search on the DB
  - Bad performance
  - CPU/IO overhead
- Offload full-text queries to Hibernate Search engine
  - Fully indexed
  - Horizontally scalable
- Based on Apache Lucene

Hibernate Search (cont'd)

- Annotate entities with @Indexed
- Annotate properties with @Field
  - Index the text: index=Index.YES
  - "Analyze" the text: analyze=Analyze.YES
    - Lucene analyzer
    - Chunks sentences into words
    - Lowercase all of them
    - Exclude common words ("a", "the")
- Combo of indexing and analysis == performant full-text searches!

En general

https://docs.jboss.org/hibernate/orm/

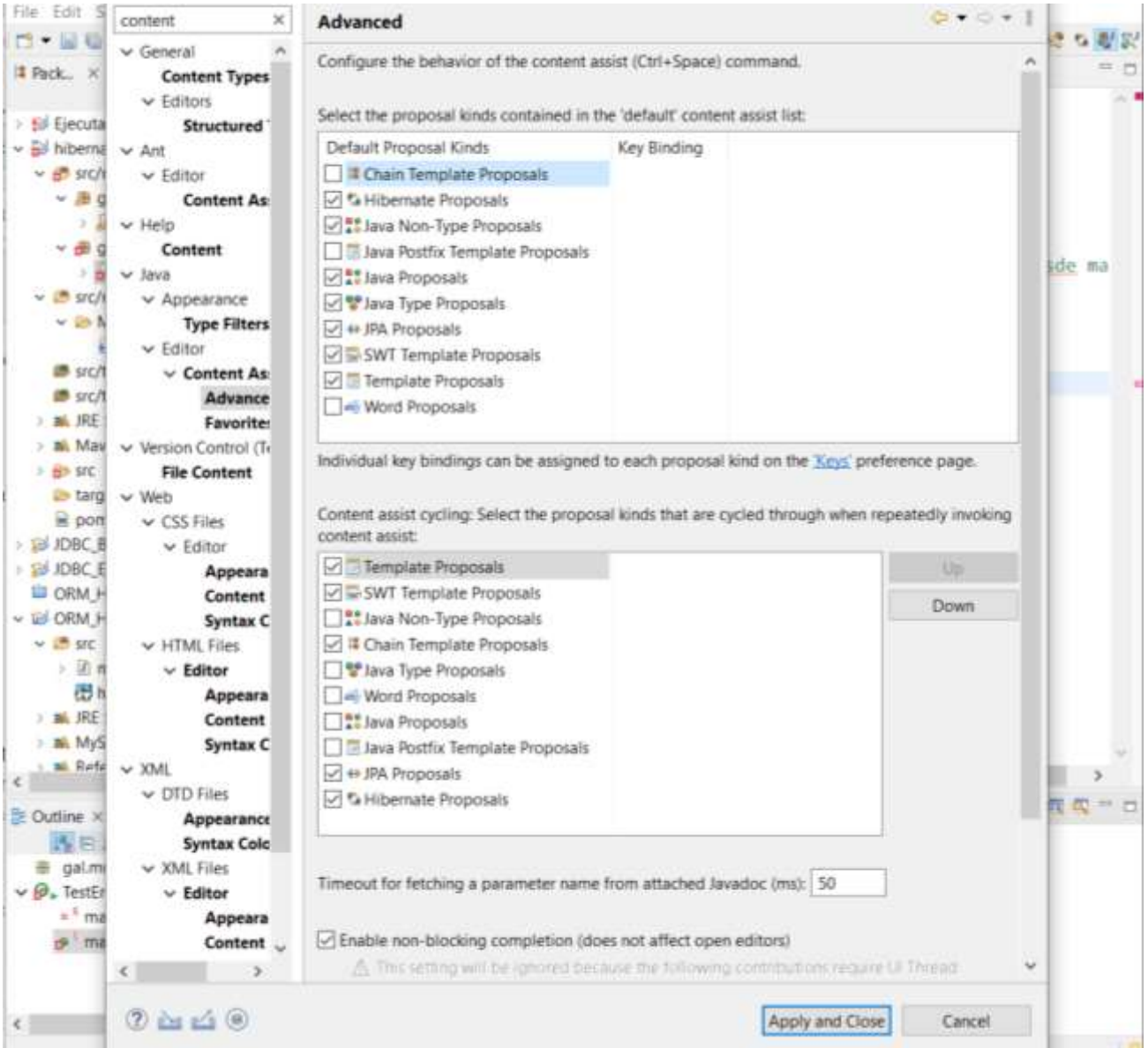
https://docs.jboss.org/hibernate/orm/5.6/quickstart/html_single/

https://docs.jboss.org/hibernate/orm/5.6/userguide/html_single/Hibernate_User_Guide.html

Las librería a añadir en el proyecto estarán en:
https://sourceforge.net/projects/hibernate/files/hibernate-orm/

hibernate-release-5.X.0.Final.zip\hibernate-release-5.X.0.Final\lib\required

Autocomplete of eclipse is not working



Lenguaje Consulta-HQL (Hibernate Query Language)

El Hibernate Query Languaje (HQL) es el lenguaje de consultas que usa Hibernate para obtener los objetos desde la base de datos. Su principal particularidad es que las consultas se realizan sobre los objetos java que forman nuestro modelo de negocio, es decir, las entidades que se persisten en Hibernate. Ésto hace que HQL tenga las siguientes características:

- Los tipos de datos son los de Java.
- Las consultas son independientes del lenguaje de SQL específico de la base de datos
- Las consultas son independientes del modelo de tablas de la base de datos.
- Es posible tratar con las colecciones de Java.
- Es posible navegar entre los distintos objetos en la propia consulta.

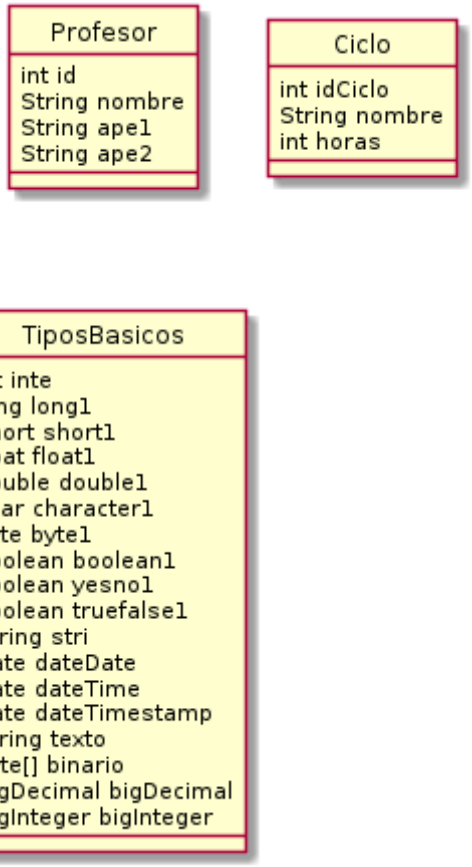
Vuelvo a insistir sobre al apartado anterior. En Hibernate las consultas HQL se lanzan (o se ejecutan) sobre el modelo de entidades que hemos definido en Hibernate, esto es, sobre nuestras clases de negocio.
De forma poco ortodoxa se podría ver cómo que nuestro *modelo de tablas* en HQL son las clases Java y **NO** las tablas de la base de datos. Es decir que cuando hagamos "SELECT columna FROM nombreTabla" , el "nombreTabla" será una clase Java y "columna" será una propiedad Java de dicha clase y **nunca** una tabla de la base de datos ni una columna de una tabla.

Modelo

En los ejemplos que vamos a realizar van a usarse las siguientes clases Java y tablas, que sólo mostraremos en formato UML. No vamos a poner el código fuente ni los ficheros de hibernate de mapeo ya que aún no han sido explicadas todas la características que usan en lecciones anteriores.

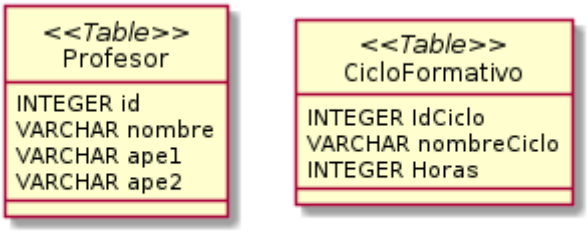
Modelo de Java

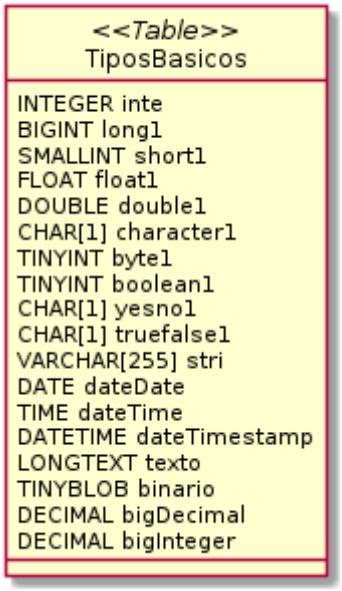
El modelo de clases Java es el siguiente:



Modelo de Tablas

El modelo de tablas es el siguiente:





Podemos ver cómo hay un par de diferencias entre el modelo en Java y el modelo de tablas.

- Para la clase `Ciclo` su tabla se llama `CicloFormativo`
- En la clase `Ciclo` su propiedad `nombre` se almacena en la columna llamada `nombreCiclo`

Veamos ahora un sencillo ejemplo de consulta en HQL

```
SELECT c FROM Ciclo c ORDER BY nombre
```

¿Qué diferencias podemos ver entre HQL y SQL?

- `Ciclo` hace referencia a la clase Java`ejemplo02.Ciclo` y **NO** a la tabla `CicloFormativo`.Nótese que la clase Java y la tabla tienen distinto nombre .
- Es necesario definir el alias `c` de la clase Java `Ciclo`.
- Tras la palabra `SELECT` se usa el alias en vez del "`*`".
- Al ordenar los objetos se usa la propiedad `nombre` de la clase `Ciclo` en vez de la columna `nombreCiclo` de la tabla `CicloFormativo`.

Recuerda incluir el alias en la consulta HQL. Si no se hace y se deja la consulta de la siguiente forma:

```
SELECT Ciclo FROM Ciclo
```

se producirá la siguiente excepción:

```
java.lang.NullPointerException
```

Hibernate soporta **no** incluir la parte del `SELECT` en la consulta HQL, quedando entonces la consulta de la siguiente forma:

```
FROM Ciclo
```

pero en la propia documentación se recomienda no hacerlo [1] ya que de esa forma se mejora la portabilidad en caso de usar el lenguaje de consultas de JPA [2].
Se ha hecho mención de esta característica ya que en muchos tutoriales que se encuentran por Internet se hace uso de ella.

Mayúsculas

Respecto a la sensibilidad de las mayúsculas y minúsculas , el lenguaje HQL sí que lo es, pero con matices.

- Las palabras clave del lenguaje **NO** son sensibles a las mayúsculas o minúsculas.
  - Las siguientes 2 consultas son equivalentes.

```
select count(*) from Ciclo
SELECT COUNT(*) FROM Ciclo
```

- El nombre de las clases Java y sus propiedades **SI** son sensibles a las mayúsculas o minúsculas.
  - La siguiente consulta HQL es correcta

```
SELECT c.nombre FROM Ciclo c WHERE nombre='Desarrollo de aplicaciones Web'
```

  - La siguiente consulta HQL es **errónea** ya que la propiedad `nombre` está escrita con la "N" en mayúsculas.

```
SELECT c.Nombre FROM Ciclo c WHERE Nombre='Desarrollo de aplicaciones Web'
```

  - La siguiente consulta HQL es **errónea** ya que el nombre de la clase Java `Ciclo` está escrita con la "c" en minúsculas.

```
SELECT c.nombre FROM ciclo c WHERE nombre='Desarrollo de aplicaciones Web'
```

- Al realizar comparaciones con los valores de las propiedades, éstas **NO** son sensibles a las mayúsculas o minúsculas.
  - Las siguientes 2 consultas retornan los mismos objetos

```
SELECT c.nombre FROM Ciclo c WHERE nombre='Desarrollo de aplicaciones Web'
SELECT c.nombre FROM Ciclo c WHERE nombre='DESARROLLO DE APLICACIONES WEB'
```

Filtrando

Al igual que en SQL en HQL también podemos filtrar los resultados mediante la cláusula `WHERE`. La forma de usarla es muy parecida a SQL.

```
SELECT p FROM Profesor p WHERE nombre='ISABEL' AND ape1<>'ORELLANA'
```

Al igual que con el nombre de la clase, el nombre de los campos del `WHERE` siempre hace referencia a las propiedades Java y nunca a los nombres de las columnas.De esa forma seguimos independizando nuestro código Java de la estructura de la base de datos.

Literales

Texto

El carácter para indicar un literal de texto es la comilla simple no pudiéndose usar la doble comilla.

```
SELECT p FROM Profesor p WHERE nombre='juan'
```

Si se quiere usar la comilla dentro de un literal deberemos duplicarla.

```
SELECT p FROM Profesor p WHERE ape1='perez l''andreu'
```

Integer

Para incluir un número del tipo `integer` simplemente se escribe dicho número.

```
SELECT tb FROM TiposBasicos tb WHERE inte=4
```

Long

Para incluir un número del tipo `long` se escribe dicho número y se añade una `L` mayúscula al final.

```
SELECT tb FROM TiposBasicos tb WHERE long1=4L
```

double

Para representar un `double` se escribe el número separarando la parte decimal con un punto o se puede usar la notación científica.

```
SELECT tb FROM TiposBasicos tb WHERE double1=1.45
SELECT tb FROM TiposBasicos tb WHERE double1=1.7976931348623157E308
```

float

Para representar un `float` se escribe el número separarando la parte decimal con un punto o se puede usar la notación científica pero se le añade el carácter `F` en mayúscula al final.

```
SELECT tb FROM TiposBasicos tb WHERE float1=1.45F
SELECT tb FROM TiposBasicos tb WHERE float1=3.4028235E38F
```

Fecha

Para indicar una fecha la incluiremos entre comillas simples con el formato `yyyy-mm-dd`

```
SELECT tb FROM TiposBasicos tb WHERE dateDate='2012-07-25'
```

Hora

Para indicar una hora la incluiremos entre comillas simples con el formato `hh:mm:ss`

```
SELECT tb FROM TiposBasicos tb WHERE dateTime='02:05:10'
```

Fecha y hora

Para indicar una fecha y hora la incluiremos entre comillas simples con el formato `yyyy-mm-dd hh:mm:ss.millis`,siendo optativos el último punto y los milisegundos .

```
SELECT tb FROM TiposBasicos tb WHERE dateTime='2012-07-25 02:05:10'
```

Operadores de comparación

Para comparar los datos en una expresión se pueden usar las siguientes Operadores:

- Signo igual "=": La expresión será verdadera si los dos datos son iguales. En caso de comparar texto, la comparación **no** es sensible a mayúsculas o minúsculas.
- Signo mayor que ">": La expresión será verdadera si el dato de la izquierda es mayor que el de la derecha.
- Signo mayor que ">=": La expresión será verdadera si el dato de la izquierda es mayor o igual que el de la derecha.
- Signo mayor que "<": La expresión será verdadera si el dato de la izquierda es menor que el de la derecha.

- Signo mayor que "<=": La expresión será verdadera si el dato de la izquierda es menor o igual que el de la derecha.
- Signo desigual "<>": La expresión será verdadera si el dato de la izquierda es distinto al de la derecha.
- Signo desigual "!=": La expresión será verdadera si el dato de la izquierda es distinto al de la derecha.
- Operador "`between`": La expresión será verdadera si el dato de la izquierda está dentro del rango de la derecha.

```
SELECT tb FROM TiposBasicos tb WHERE inte BETWEEN 1 AND 10
```

- Operador "`in`": La expresión será verdadera si el dato de la izquierda está dentro de la lista de valores de la derecha.

```
SELECT tb FROM TiposBasicos tb WHERE inte IN (1,3,5,7)
```

- Operador "`like`": La expresión será verdadera si el dato de la izquierda coincide con el patrón de la derecha.Se utilizan los mismos signos que en SQL "`%`" y "_".

```
SELECT tb FROM TiposBasicos tb WHERE stri LIKE 'H_la%'
```

- Operador "`not`": Niega el resultado de una expresión.
- expresión "`is null`": Comprueba si el dato de la izquierda es `null`.

```
SELECT tb FROM TiposBasicos tb WHERE dataDate IS NULL
```

Operadores Lógicos

Se puede hacer uso de los típicos operadores lógicos como en SQL:

- AND
- OR
- NOT

```
SELECT p FROM Profesor p WHERE nombre='ANTONIO' AND (ape1='LARA' OR ape2='RUBIO')
```

Operadores Aritméticos

Se puede hacer uso de los típicos operadores aritméticos:

- suma +
- resta -
- multiplicación *
- division /

```
SELECT tb FROM TiposBasicos tb WHERE (((inte+1)*4)-10)/2=1
```

Funciones de agregación

Las funciones de agregación que soporta HQL son:

- `AVG()`: Calcula el valor medio de todos los datos.
- `SUM()`: Calcula la suma de todos los datos.
- `MIN()`: Calcula el valor mínimo de todos los datos.
- `MAX()`: Calcula el valor máximo de todos los datos.
- `COUNT()`: Cuanta el nº de datos.

```
SELECT
AVG(c.horas),SUM(c.horas),MIN(c.horas),MAX(c.horas),COUNT(*)
FROM Ciclo c
```

Funciones sobre escalares

Algunas de las funciones que soporta HQL sobre datos escalares son:

- `UPPER(s)`: Transforma un texto a mayúsculas.
- `LOWER(s)`: Transforma un texto a minúsculas.
- `CONCAT(s1, s2)`: Concatena dos textos
- `TRIM(s)`:Elimina los espacio iniciales y finales de un texto.
- `SUBSTRING(s, offset, length)`: Retorna un substring de un texto.El `offset` empieza a contar desde 1 y no desde 0.
- `LENGTH(s)`: Calcula la longitud de un texto.
- `ABS(n)`: Calcula el valor absoluto de un número.
- `SQRT(n)`: Calcula la raíz cuadrada del número
- Operador "`||`" : Permite concatenar texto.

```
SELECT p.nombre || ' ' || p.ape1 || ' ' || p.ape2 FROM
Profesor p WHERE Id=1001
```

Ordenación

Como en SQL también es posible ordenar los resultados usando `ORDER BY`. Su funcionamiento es como en SQL.

```
SELECT p FROM Profesor p ORDER BY nombre ASC,ape1 DESC
```

Las palabras `ASC` y `DESC` son opcionales al igual que en SQL.

El uso de funciones escalares y funciones de agrupamiento en la cláusula `ORDER BY` sólo es soportado por Hibernate si es soportado por el lenguaje de SQL de la base de datos sobre la que se está ejecutando.

No se permite el uso de expresiones aritméticas en la cláusula `ORDER BY`.

Agrupaciones

Al igual que en SQL se pueden realizar agrupaciones mediante las palabras claves `GROUP BY` y `HAVING`

```
SELECT nombre,count(nombre) FROM Profesor p GROUP BY nombre HAVING
count(nombre)>1 ORDER BY count(nombre)
```

Los nombres de profesores que se repiten mas de una vez

El uso de funciones escalares y funciones de agrupamiento en la cláusula `HAVING` sólo es soportado por Hibernate si es soportado por el lenguaje de SQL de la base de datos sobre la que se está ejecutando.

No se permite el uso de expresiones aritméticas en la cláusula `GROUP BY`.

Subconsultas

HQL también soporta subconsultas como en SQL.

```
SELECT c.nombre,c.horas FROM Ciclo c WHERE c.horas > (SELECT
AVG(c2.horas) FROM Ciclo c2)
```

Ciclos que duran más horas que la media de duración de todos los ciclos

Uso VisualParadigm?

¿JPA vs Hibernate?

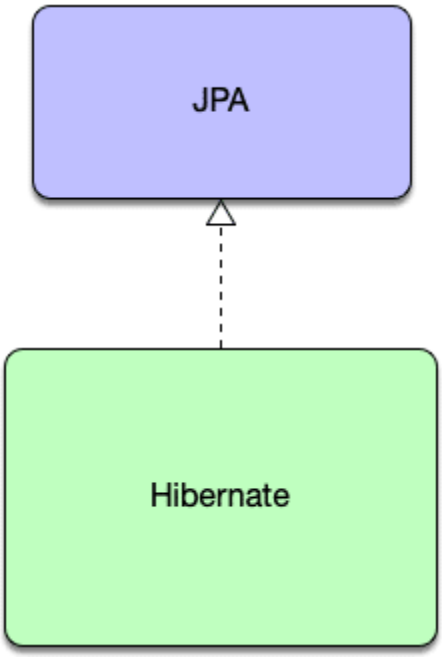
Tabla de Contenidos

JPA vs Hibernate . ¿Qué diferencia hay entre ellos y cual se debe elegir? . Esta es una de las preguntas más habituales cuando uno empieza a trabajar con un framework de Persistencia.

¿Que es JPA?

JPA o Java Persistence API es una **especificación** concretamente la JSR 338 . Una especificación no es más que un documento en el que se define como se debe gestionar una funcionalidad X. En este caso una capa de persistencia con objetos Java. Por ejemplo que anotaciones han de usarse, como han de persistirse los objetos como han de buscarse, cuál es su ciclo de vida etc .

Al tratarse de un documento evidentemente no **implementa nada**. **Para poder trabajar** con ella se **necesitará** tener un Framework que implemente la especificación, uno de los **más conocidos es Hibernate**



¿Existen más implementaciones de JPA que al fin y al cabo es únicamente una especificación Si aquí podemos ver algunas:

EclipseLink : Otra implementación muy utilizada ya que está apoyada por la fundación Eclipse y da soporte a JPA.

DataNucleus : Otro framework de persistencia que soporta JPA pero incluye muchas más opciones y tipos de persistencia.

TopLink : La implementación de Oracle clásica para sus productos hoy por hoy esta basada en EclipseLink

ObjectDB Otra implementación de JPA que promete un rendimiento alto comparado con sus competidores.

Por lo tanto, no solo existe Hibernate

Las siglas ORM significan “Object-Relational mapping” [1)] y en castellano es “Mapeo Objeto-Relacional”. El ORM es simplemente el código que escribimos para guardar el valor de nuestras clases en una base de datos relacional. Simplemente éso.

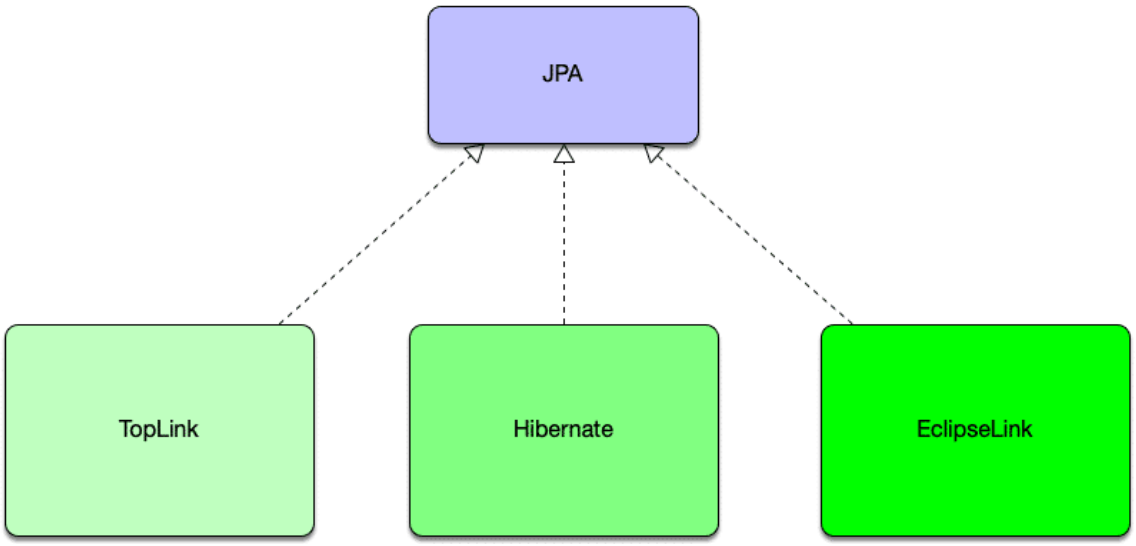
Es decir, es un framework de persistencia de nuestros datos a una base de datos.

Anteriormente:

Especificaciones de ORM

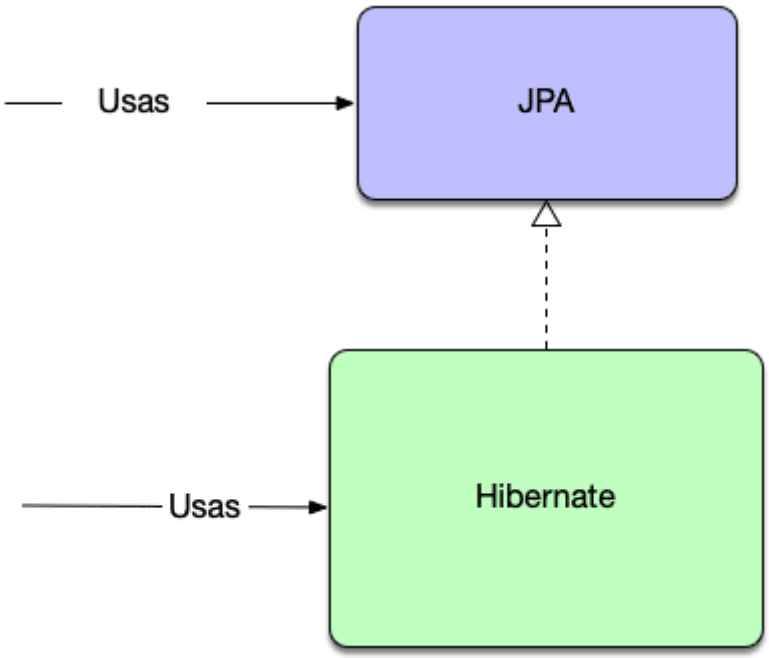
- EJB
- JDO
- JPA (última en llegar)

JPA es **el estándar** de persistencia en Java y para aupar más aún a JPA ,Hibernate implementa la especificación de JPA



¿JPA vs Hibernate?

Normalmente la pregunta de mucha gente es que usar si JPA o Hibernate, es decir, si usar directamente la especificación o usar directamente el framework.



¿Usar Hibernate?

Hay respuestas bastante razonables en las que la gente prefiere usar el framework (Hibernate) de forma directa . ¿Porque? Porque el framework tiene capacidades adicionales que JPA como standard y especificación no soporta.

Hasta aquí parece todo muy razonable. Sin embargo es interesante contemplar usar JPA solamente . ¿Por qué? . Porque en la mayoría de las situaciones el usar directamente el framework aunque aporte alguna cosa adicional esos aportes no son críticos y las pocas ventajas que aporta no son suficientes comparadas con las ventajas de usar directamente la especificación.

¿Usar la especificación?

Vamos a ver un poco sus ventajas:

1. La especificación está muy trabajada y a nivel cultural muy extendida entre todos los desarrolladores . El uso concreto de un framework no lo está tanto.
2. La especificación nos permitirá cambiar de implementación de forma transparente si tenemos la necesidad. Es raro tener que hacerlo pero existe la posibilidad y permitirá portar de forma más sencilla nuestra aplicación entre

diferentes servidores Java EE que usen diferentes especificaciones.
3. La especificación es mucho más atemporal ya que es un documento y todo el mundo se ciñe a él. Por lo tanto lo aprendido dura más en el tiempo y es más sólido al paso de las décadas.
4. Al ser algo tan inmutable es habitual que aparezcan nuevos frameworks o nuevas especificaciones que lo complementen. Por ejemplo, un caso claro de JPA es el uso de Spring Data con JPA que permite un trabajo mucho más cómodo y rápido con Repositorios.

JPA vs Hibernate Conclusión

Mi recomendación es usar siempre JPA como especificación y programar contra la especificación usando EntityManager ,EntityManagerFactory etc y no usar clases concretas de una implementación como puedan ser en Hibernate Session y SessionFactory. ¿Cuál de todas las implementaciones usar? . Pues para mí esta claro que Hibernate es la implementación más conocida y por lo tanto la que debemos instalar . Eso sí recordemos que debemos usar solo anotaciones del standard de JPA y evitar en lo posible hacer uso directo de las funciones de Hibernate esto nos permitirá mantener la portabilidad del código y las ventajas de que soluciones que salgan a futuro y se apoyen en JPA , las podamos integrar de forma natural.

Hibernate con JPA

Para hacer la persistencia de nuestros objetos Java a la base de datos hay que indicar cómo se debe realizar dicha persistencia. Para ello hay dos métodos:

- Ficheros XML
- Anotaciones en el código.

Las anotaciones que tiene Hibernate son actualmente las del estándar de JPA. De esa forma al ver las anotaciones de hibernate estaremos viendo las de JPA.Cuando veamos los ficheros XML usaremos los ficheros propietarios de Hibernate, así también veremos cómo funciona realmente Hibernate

Mapeo de una Entidad

Una entidad va a ser una simple clase Java que deseamos persistir en la base de datos.

Este tutorial está dividido en 3 partes:

- La clase Java
- Fichero de mapeo ".hbm.xml"
- Anotaciones

La clase Java

La clases Java deberán tener las siguientes características:

- Deben tener un constructor público sin ningún tipo de argumentos [1].
- Para cada propiedad que queramos persistir debe haber un método get/set asociado.
- Implementar el interfaz Serializable [2]

```
1:  public class Profesor implements Serializable {
2:      private int id;
3:      private String nombre;
4:      private String ape1;
5:      private String ape2;
6:
7:      public Profesor(){
8:      }
9:
10:     public Profesor(int id, String nombre, String ape1, String ape2) {
11:         this.id = id;
12:         this.nombre = nombre;
13:         this.ape1 = ape1;
14:         this.ape2 = ape2;
15:     }
16:
17:     public int getId() {
18:         return id;
19:     }
20:
21:     public void setId(int id) {
22:         this.id = id;
23:     }
24:
25:     public String getNombre() {
26:         return nombre;
27:     }
28:
29:     public void setNombre(String nombre) {
30:         this.nombre = nombre;
31:     }
32:
33:     public String getApe1() {
34:         return ape1;
35:     }
36:
37:     public void setApe1(String ape1) {
38:         this.ape1 = ape1;
39:     }
40:
41:     public String getApe2() {
42:         return ape2;
```

```
43:     }
44:
45:     public void setApe2(String ape2) {
46:         this.ape2 = ape2;
47:     }
48: }
```

Profesor.java

Vemos en el código fuente cómo la clase Profesor tiene un constructor sin ningún tipo de argumentos (Linea 7). Además para las propiedades id,nombre,ape1 y ape2 están el par de métodos get y set y por último implementa el interfaz Serializable (Línea 1).

¿Ya podemos persistir la clase Profesor usando hibernate? Pues **NO**, debemos indicarle a hibernate toda la metainformación relativa a esta clase. Hay que *explicarle* como se mapeará el objeto en una base de datos relacional [3)] , indicando para ello en que tabla de base de datos se debe guardar cuál es la clave primaria de la tabla, las columnas que tiene, etc.

Fichero de mapeo ".hbm.xml"

Para cada clase que queremos persistir se creará un fichero 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.
Nada impide que el fichero .hbm.xml esté en otro paquete distinto al de la clase Java. En este sentido suele haber 2 posibilidades:

1. Almacenar el fichero .hbm.xml en el mismo paquete que la clase Java a la que hace referencia.
2. Crear un *árbol* alternativo de paquetes donde almacenar los ficheros .hbm.xml. Por ejemplo, si tenemos el paquete raíz com.miempresa.proyecto.dominio donde se guardan todas las clases Java a persistir, crear otro paquete llamado com.miempresa.proyecto.persistencia donde almacenar los ficheros .hbm.xml.

La ventaja de la segunda opción es que en caso de que no queramos usar Hibernate, simplemente hay que borrar toda la carpeta com.miempresa.proyecto.persistencia y ya está, mientras que la ventaja de la primera opción es que la clase Java y su correspondiente fichero de mapeo están mas juntos facilitando en caso de algún cambio en la clase Java el cambio en el fichero de mapeo.
Mi opinión personal es que es mejor usar la segunda opción ya que quedan más independizadas las clases de negocio del método de persistencia.

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

Profesor.hbm.xml
Podemos ver cómo el fichero Profesor.hbm.xml es un típico fichero xml.

- En la linea 1 vemos la declaración de que es un fichero XML.
- En la linea 2 se aprecia la declaración del DOCTYPE junto con la referencia al documento http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd DTD que permite validarlo. Es decir que si nos descargamos el fichero hibernate-mapping-3.0.dtd podremos saber todos los elementos que hay en un fichero de mapeo.
- El nodo raiz del documento xml se llama <hibernate-mapping> y se encuentra en la línea 3.
- La parte interesante de este fichero empieza en la línea 4. Vemos el tag <class> que nos indica que vamos a mapear una clase.
    - En el atributo name deberemos poner el FQCN [4] de la clase que queremos mapear.Es decir el nombre de la clase incluyendo el paquete en el que se encuentra.

        - El atributo table nos indica el nombre de la tabla en la que vamos a mapear la clase. Este atributo es opcional si el nombre de la clase Java y el de la tabla coinciden.
    - El tag <id> de la línea 5 se usa para indicar la propiedad de la clase que es la clave primaria.
        - El atributo name es el nombre de la propiedad Java que contiene la clave primaria.
        - El atributo column contiene el nombre de la columna de la base de datos asociado a la propiedad. Este atributo es opcional si el nombre de la propiedad Java y el nombre de la columna coinciden.
        - El atributo type indica el tipo de la propiedad Java. Este atributo no es necesario puesto que Hibernate por defecto ya usa el tipo de la propiedad Java. Mas información en Tipos básicos.
    - El tag <property> de las líneas 6 a la 8 se usa para declarar más propiedades Java para ser mapeadas en la base de datos. Si no declaramos las propiedades Java mediante este tag no se leerán o guardarán en la base de datos.
        - El atributo name es el nombre de la propiedad Java que queremos mapear a la base de datos.
        - El atributo column contiene el nombre de la columna de la base de datos asociado a la propiedad. Este atributo es opcional si el nombre de la propiedad Java y el nombre de la columna coinciden.

Recuerda que usando el atributo column puedes especificar un nombre de columna en la tabla distinto del nombre de la propiedad en la clase Java.

```
<?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" table="Profesor" >
    <id column="Id" name="id" type="integer"/>
    <property name="nombre"  />
    <property name="ape1" column="primer_apellido" />
    <property name="ape2" column="segundo_apellido" />
  </class>
</hibernate-mapping>
```

Como podemos apreciar en el DTD http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd, hay muchos más tags y atributos en los ficheros .hbm.xml pero por ahora simplemente hemos visto lo mas básico. Durante el resto del curso iremos viendo muchas mas opciones de este fichero.

Si buscamos documentación sobre hibernate podemos encontrar que el DOCTYPE antes de la versión 3.6 era [5] :

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

Así que no confundirlo al hacer algún *copy-empastre* desde algún tutorial de Internet de versiones anteriores.
Hemos comentado que necesitamos los métodos get/set para que hibernate acceda a los campos. Sin embargo nos puede interesar que no estén alguno de esos métodos para que el usuario no

pueda cambiar o leer los valores. En ese caso le deberemos decir a Hibernate que acceda directamente a las propiedades privadas, ya que por suerte Hibernate sabe hacerlo.

Para ello modificaremos el fichero .hbm.xml añadiendo el atributo access="field" a la propiedad sobre la que queremos que acceda directamente.
Por ejemplo si no quisieramos tener un getNombre() o setNombre() de la clase Profesor deberíamos cambiar el fichero Profesor.hbm.xml añadiendo en la definición de la columna profesor el texto access="field", quedando en ese caso de la siguiente forma:

```
<property name="nombre" access="field" />
```

La propiedad access="field" también puede aplicarse al tag <id> como a los diversos tag que definen una propiedad en Hibernate.
Realmente como norma general siempre deberíamos utilizar al tributo access="field" ya que así, podremos decidir *tranquilamente* si poner o no los métodos get'/set y además dichos métodos get'/set podrían tener reglas o calculos que hicieran que se generaran errores en nuestra aplicación al ser cargados desde Hibernate.Más información sobre este tema en Avoiding Anemic Domain Models with Hibernate
Sin embargo, durante el resto del curso no haremos uso de esta característica para simplificar las explicaciones/ejemplos.

Anotaciones

En el apartado anterior hemos visto cómo mediante un fichero .hbm.xml podemos especificar cómo mapear la clases Java en tablas de base de datos.
Desde hace algunos años en Java se ha creado el concepto llamado El infierno XML.Este infierno ha consistido en que en demasiados frameworks [6] se hacía un uso intensivo de XML , siendo el XML un formato de ficheros demasiado largo de escribir , muy repetitivo , verboso, etc. Ésto ha llevado a crear una solución para evitar los ficheros XML de persistencia en hibernate [7]: El uso de Anotaciones Java en el propio código. Estas anotaciones permiten especificar de una forma más compacta y sencilla la información de mapeo de las clases Java.
Inicialmente Hibernate creó sus propia anotaciones en el paquete org.hibernate.annotations pero a partir de la versión 4 de Hibernate la mayoría de dichas anotaciones han sido 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 pero en ese caso Hibernate 4 no las ha marcado como java.lang.Deprecated
Veamos ahora el ejemplo de la clase Profesor pero mapeada con anotaciones.

```
1: @Entity
2: @Table(name="Profesor")
3: public class Profesor implements Serializable {
4:
5:     @Id
6:     @Column(name="Id")
7:     private int id;
8:
9:     @Column(name="nombre")
10:    private String nombre;
11:
12:    @Column(name="ape1")
13:    private String ape1;
14:
15:    @Column(name="ape2")
16:    private String ape2;
17:
18:
19:    public Profesor(){
20:    }
21:
22:    public Profesor(int id, String nombre, String ape1, String ape2) {
23:        this.id = id;
24:        this.nombre = nombre;
25:        this.ape1 = ape1;
26:        this.ape2 = ape2;
27:    }
28:
29:    public int getId() {
30:        return id;
31:    }
32:
33:    public void setId(int id) {
34:        this.id = id;
35:    }
36:
37:    public String getNombre() {
38:        return nombre;
39:    }
40:
41:    public void setNombre(String nombre) {
42:        this.nombre = nombre;
43:    }
44:
45:    public String getApe1() {
46:        return ape1;
47:    }
48:
```

```
49:     public void setApe1(String ape1) {
50:         this.ape1 = ape1;
51:     }
52:
53:     public String getApe2() {
54:         return ape2;
55:     }
56:
57:     public void setApe2(String ape2) {
58:         this.ape2 = ape2;
59:     }
60: }
```

Profesor.java anotado

Las anotaciones que se han usado son las siguientes:

- @**Entity**: Se aplica a la clase e indica que esta clase Java es una entidad a persistir. Es una anotación estándar de JPA. En nuestro ejemplo estamos indicando que la clase Profesor es una entidad que se puede persistir.
- **@Table(name="Profesor")**: 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. Es una anotación estándar de JPA. En nuestro ejemplo estamos indicando que la clase Profesor se persistirá en la tabla Profesor de la base de datos.
- @**Id**: Se aplica a una propiedad Java e indica que este atributo es la clave primaria. Es una anotación estándar de JPA. En nuestro ejemplo estamos indicando que la propiedad Java id 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. Es una anotación estándar de JPA. En nuestro ejemplo estamos indicando que la propiedad Java id se persistirá en una columna llamada Id.
- **@Column(name="nombre")**: 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. Es una anotación estándar de JPA. En nuestro ejemplo estamos indicando que la propiedad Java nombre se persistirá en una columna llamada nombre.
- **@Column(name="ape1")**: Es igual al caso anterior pero para la propiedad ape1.
- **@Column(name="ape2")**: Es igual al caso anterior pero para la propiedad ape2.

Una diferencia importante entre usar el fichero de mapeo .hbm.xml y las anotaciones es que en el fichero es **obligatorio** indicar todas las propiedades que queremos que se persistan en la base de datos, mientras que usando las anotaciones éso no es necesario. Usando anotaciones se persisten todas las propiedades que tengan los métodos get/set.

Ya hemos comentado en el apartado anterior sobre como Hibernate accede a los datos al usar el fichero .hbm.xml, si mediante el uso de los métodos get/set o mediante el acceso a las propiedades.Veamos como se especifica ésto mediante notaciones:

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

Personalmente siempre coloco las anotaciones sobre las propiedades ya que así tengo todas ellas más agrupadas visualmente y queda el código más legible.

Configurando

Como con cualquier otro framework es necesario configurarlo para su correcto funcionamiento. Ya hemos visto que se puede configurar usando los ficheros de mapeo o las anotaciones Mapeo de una Entidad. Sin embargo, aún quedan aspectos que debemos configurar como:

- Datos de conexión a la base de datos
- Ubicación de las clases a persistir.
- Ubicación de los ficheros .hbm.xml.
- Nivel de Log
- Pool de conexiones a usar.
- Etc, etc.

hibernate.cfg.xml

La forma de configurar hibernate es usando el fichero XML de configuración llamado hibernate.cfg.xml. Este fichero deberemos guardarlo en el paquete raíz de nuestras clases Java, es decir fuera de cualquier paquete.Si estamos usando NetBeans deberá ser en la carpeta src de nuestro proyecto.
La información que contiene es la siguiente:

- Propiedades de configuración.
- Las clases que se quieren mapear.

El fichero tiene la siguiente estructura:

```
1: <?xml version="1.0" encoding="UTF-8"?>
2: <!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
3: <hibernate-configuration>
4:   <session-factory>
5:     <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
6:     <property name="connection.url">jdbc:mysql://localhost/hibernate1</property>
7:     <property name="connection.username">hibernate1</property>
8:     <property name="connection.password">hibernate1</property>
9:     <property name="dialect">org.hibernate.dialect.MySQL5Dialect</property>
10:     <property name="hibernate.show_sql">true</property>
11:
12:     <mapping resource="ejemplo01/Profesor.hbm.xml"/>
13:     <mapping class="ejemplo01.Profesor"/>
14:
15:   </session-factory>
16: </hibernate-configuration>
```

Fichero hibernate.cfg.xml

Podemos ver que el fichero hibernate.cfg.xml es un típico fichero xml.

- En la línea 1 vemos la declaración de que es un fichero XML.
- En la línea 2 se aprecia la declaración del DOCTYPE junto con la referencia al documento http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd DTD que permite validarlo. Es decir que si nos descargamos el fichero hibernate-configuration-3.0.dtd podremos saber todos los elementos que hay en un fichero de mapeo.
- El tag raiz del documento xml se llama <hibernate-configuration> y se encuentra en la línea 3.
- Dentro del tag <hibernate-configuration> encontramos,en la línea 4, el tag <session-factory> que contendrá la configuración de hibernate.

- Las siguientes 5 líneas (desde la 5 a la 10) contienen propiedades de configuración mediante el tag <property>
    - El atributo name contiene el nombre de la propiedad de configuración.
    - El contenido del tag <property> define el valor de la propiedad de configuración.
- Las líneas 12 y 13 contienen el tag <mapping> que se usa para indicarle a hibernate las clases que queremos usar desde hibernate.
    - El atributo resource contiene el nombre de un fichero .hbm.xml asociada a la clase que queremos persistir. En nuestro caso del fichero Profesor.hbm.xml.
    - El atributo class contiene la FQCN [1)] de la clase que queremos persistir. En nuestro ejemplo será la clase ejemplo02.Usuario.

Para una misma clase solo es necesario indicar una única vez el tag <mapping> con el atributo resource o class.

En el fichero se ha incluido dos veces para indicar las dos posibilidades **excluyentes**.

Si buscamos documentación sobre hibernate podemos encontrar que el DOCTYPE del fichero hibernate.cfg.xml antes de la versión 3.6 era [2)] :

```
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD//EN" "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
```

Así que no confundirlo al hacer algún *copy-empastre* desde algún tutorial de Internet de versiones anteriores.

Pasemos ahora a explicar en detalle los tag <property> y <mapping>.

Tag <property>

El tag <property> se usa para definir cada una de las propiedades de configuración [3)] de hibernate. Como ya hemos indicado consta del atributo name con el nombre de la propiedad de configuración. Dentro del tag <property> incluiremos el valor de dicha propiedad de configuración.

Volvamos ahora a ver las propiedades de configuración del fichero

```
1: <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
2: <property name="connection.url">jdbc:mysql://localhost/hibernate1</property>
3: <property name="connection.username">hibernate1</property>
4: <property name="connection.password">hibernate1</property>
5: <property name="dialect">org.hibernate.dialect.MySQL5Dialect</property>
6: <property name="hibernate.show_sql">true</property>
```

Propiedades de configuración hibernate.cfg.xml

Estas propiedades se usan para poder conectarse mediante JDBC a la base de datos.

Propiedad de configuración	Explicación
connection.driver_class	Contiene la FQCN del driver de la base de datos a usar.
connection.url	La URL de conexión a la base de datos tal y como se usa en JDBC
connection.username	El usuario de la base de datos
connection.password	La contraseña de la base de datos

dialect	El lenguaje de SQL que usará Hibernate contra la base de datos. Este parámetro es opcional ya que hibernate lo puede intentar deducir a partir de los datos de la conexión [4]. Los posibles valores de esta propiedad de configuración son la FQCN de una clase Java que extienda de la clase org.hibernate.dialect.Dialect. La lista de dialectos que soporta hibernate se encuentra en el paquete org.hibernate.dialect.
hibernate.show_sql	Propiedad opcional que indica si se mostrará por la consola la SQL que lanza 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 funcionanado Hibernate
connection.datasource	Indica el nombre del DataSource con el que se conectará Hibernate a la base de datos. En caso de estar esta propiedad no debería estar ninguna de las 4 primeras propiedades sobre la conexión o viceversa. 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.

Al utilizar la propiedad connection.datasource recuerda que antes del nombre que se le haya dado al datasource hay que añadir:

```
java:/comp/env/
```

Es decir que si nuestro datasource se llama jdbc/hibernate1 el valor de connection.datasource deberá ser:

```
java:/comp/env/jdbc/hibernate1
```

Los dialectos de hibenate son importantes ya que indican en qué base de datos podemos usarlo. En caso de que una base de datos no esté soportada , se puede definir una nueva clase que extienda de org.hibernate.dialect.Dialect para poder soportar dicha base de datos.

Tag <mapping>

El tag <mapping> se usa para indicarle a Hibernate las clases queremos mapear. Este tag se usa de 2 formas distintas:

- Indicando el nombre del fichero de mapeo ".hbm.xml"
- Indicando la FQCN de la clase Java que hemos anotado

Fichero de mapeo .hbm.xml

Si hemos creado un fichero de mapeo .hbm.xml para poder persistir una clase Java a la base de datos , deberemos utilizar el atributo resource en el tag <mapping>.

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

El atributo resource contiene el *path* dentro de los paquetes Java al fichero .hbm.xml.
Nótese que para separar el *path* se usa la barra y no el punto.

Estamos indicando un path dentro del sistema de paquetes de Java y **NO** una ruta en el sistema de ficheros del disco duro.

FQCN de la clase Java

Si la clase Java la hemos anotado para poder persistirla en vez de usar el fichero .hbm.xml deberemos utilizar el atributo class en el tag <mapping>.

```
<mapping class="ejemplo01.Profesor"/>
```

El atributo class contiene la FQCN de la clase Java que deseamos que se pueda persistir.
Nótese que para separar los paquetes ahora se usa el punto en vez de la barra.

Usando Hibernate

Hasta ahora hemos visto cómo configurar hibernate usando los ficheros XML de configuración o usando notaciones pero no hemos visto nada de código Java para usarlo realmente. En esta lección veremos finalmente cómo usar Java para persistir una clase.
La clase que más usaremos en Hibernate es org.hibernate.Session. Esta clase contiene métodos para leer, guardar o borrar entidades sobre la base de datos. Pero antes de poder usarla hace falta crear el objeto SessionFactory que mediante el método SessionFactory.openSession() nos dará acceso a Session.

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

Veamos ahora cómo crear el objeto SessionFactory.

SesionFactory

La forma de crear el objeto SessionFactory es mediante un objeto org.hibernate.cfg.Configuration que leerá el fichero de configuración de hibernate hibernate.cfg.xml que se encuentra en el directorio raíz de las clases Java.

```
1: import org.hibernate.SessionFactory;
2: import org.hibernate.cfg.Configuration;
3: import org.hibernate.service.ServiceRegistry;
4: import org.hibernate.service.ServiceRegistryBuilder;
5:
6: SessionFactory sessionFactory;
7:
8: Configuration configuration = new Configuration();
9: configuration.configure();
10: ServiceRegistry serviceRegistry = new ServiceRegistryBuilder().applySettings(configuration.getProperties()).buildServiceRegistry();
11: sessionFactory = configuration.buildSessionFactory(serviceRegistry);
```

Listado 1.Creación del objeto SessionFactory

- En la línea 8 vemos cómo se crea el objeto Configuration
- En la línea 9 mediante el método configure() se va a leer el fichero de configuración hibernate.cfg.xml.
- La línea 10 es nueva en hibernate 4 y contiene la lista de los distintos servicios que usará [1] hibernate.Para ello se crea un objeto de la clase org.hibernate.service.ServiceRegistry.
- La línea 11 es la que finalmente creará el objeto SessionFactory en función de la configuración y de los servicios.

Todo el código anterior deberá realizarse una única vez en la aplicación y deberemos guardar la referencia a SessionFactory para su posterior uso durante toda la vida de la aplicación.
Si vemos la documentación de hibernate 4.1 se indica que la forma de crear el objeto SessionFactory es la siguiente:

```
1: Configuration configuration = new Configuration();
2: configuration.configure();
3: sessionFactory = configuration.buildSessionFactory();
```

Forma antigua de crear el objeto SessionFactory

Sin embargo al usar este código se puede observar que el método buildSessionFactory() está depreciado.En la documentación del propio método se indica que ahora se use buildSessionFactory(ServiceRegistry serviceRegistry) es decir que la documentación contiene un error y debemos usar el código del Listado 1.
Una discusión sobre el error en la documentación de Hibernate 4 la podéis encontrar en:Is buildSessionFactory() deprecated in hibernate 4?
Al finalizar la aplicación deberemos llamar al método close():

```
sessionFactory.close();
```

Session

Ahora que ya tenemos el objeto SessionFactory podemos obtener la Session para trabajar con Hibernate. Como ya hemos visto crear la Session es tan sencillo como llamar al método openSession():

```
Session session = sessionFactory.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 trabajar con una base de datos usamos las transacciones. En hibernate es tan sencillo como:

- Crear una nueva transacción llamando al método beginTransaction() de la sesión:

```
session.beginTransaction();
```

- Hacer un commit de la transacción actual llamando al método commit() de la transacción actual de la sesión:

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

- Hacer un rollback de la transacción actual llamando al método rollback() de la transacción actual de la sesión:

```
session.getTransaction().rollback();
```

CRUD

Ya hemos llegado al punto en que tenemos todo preparado para poder trabajar con Hibernate en las operaciones fundamentales de una base de datos, las operaciones CRUD.

- **C**reate: Guardar un nuevo objeto en la base de datos.
- **R**ead:Leer los datos de un objeto de la base de datos.
- **U**pdate:Actualizar los datos de un objeto de la base de datos.
- **D**elete:Borrar los datos de un objeto de la base de datos.

Estas 4 operaciones será tan sencillas de usar desde hibernate como llamar a un único método para cada uno de ellos.

Guardar

Usaremos el método save(Object object) de la sesión pasándole como argumento el objeto a guardar.

```
Profesor profesor=new Profesor(101, "Juan", "Perez", "García");  //Creamos el objeto

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

session.save(profesor); //<|--- Aqui guardamos el objeto en la base de datos.

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

Como vemos , guardar una clase Java en la base de datos solo implica usar una única línea.

Leer

El método que debemos usar es get(Class,Serializable), al que le deberemos pasar la clase que queremos leer y su clave primaria.

```
Profesor profesor=(Profesor)session.get(profesor.class,101);
```

El método get(Class,Serializable) permite leer un único objeto de la base de datos a partir de su clave primaria.
El uso de este método tiene 2 peculiaridades:

- Uso del cast:Es necesario hacer un cast añadiendo (Message) en el retorno de la función. Ésto es así ya que el método get() se usa para cualquier tipo de entidad así que Java desconoce que tipo de datos va a retornar , por lo que debemos decírselo nosotros mediante el cast para "asegurarle" el tipo que retorna.
- El uso de la propiedad .class:Ésta es la forma que se ha definido en en lenguaje Java para pasar un objeto de la clase java.lang.Class. Véase Class Literals.

Actualizar

El método a usar es update(Object object), al que le deberemos pasar el objeto a actualizar en la base de datos

```
session.beginTransaction();

session.update(profesor);

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

El método update(Object object) simplemente actualiza el objeto de la base de datos.

Borrar

Ahora pasemos a borrar un objeto desde la base de datos.El método que debemos usar es delete(Object object), al que le deberemos pasar el objeto a borrar de la base de datos

```
session.beginTransaction();

session.delete(profesor);

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

El método delete(Object object) simplemente borra el objeto de la base de datos.

Guardar y actualizar

Muchas veces resulta cómodo al programar no tener que estar pendiente de si un objeto va a insertarse o actualizarse. Para ello Hibernate dispone del método saveOrUpdate(Object object) que inserta o actualiza en la base de datos en función de si ya existe o no dicha fila.

```
1: Profesor profesor=new Profesor(101, "Juan", "Perez", "García");
2:
3: session.beginTransaction();
4:
5: session.saveOrUpdate(profesor);
6:
7: session.getTransaction().commit();
```

En este caso dependiendo de si existe o no la fila en la base de datos con Id=101 se realizará un UPDATE o un INSERT contra la base de datos.

JPQL

Uno de los aspectos fundamentales de JPA es la posibilidad de realizar consultas sobre las entidades, muy similares a las consultas SQL. El lenguaje en el que se realizan las consultas se denomina Java Persistence Query Language (JPQL).

SpringBoot con Hibernate

- Ej. Relación OneToMany

- Ej. Relación ManyToOne

Referencias sobre Spring

- **Spring initializr (bootstrapping** - generar esqueleto programa [dependencias]/ Inicio Rápido**)**

https://start.spring.io/

  - Hibernate
  - Web
  - JPA
  - H2, MySQL…

- **Spring Boot con Hibernate**
  - **Dependencias Maven pom.xml**
    - ✓ JPA

```
<dependency>
	<groupId>org.springframework.boot</groupId>
	<artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

JPA API, JPA Implementation, JDBC y otras librerías.

Desde que Hibernate es JPA Implementation por defecto, esta dependencia indicada ya la trae por defecto también.

- ✓ H2

```
<dependency>
	<groupId>com.h2database</groupId>
	<artifactId>h2</artifactId>
	<scope>runtime</scope>
</dependency>
```

OJO. En application.yml

```
spring.h2.console.enabled=true
```

```
server.port: 8090
spring:
  application:
    name: one-to-many
  datasource:
    url: jdbc:h2:mem:testdb
    driverClassName: org.h2.Driver
    username: sa
    password: password
    initialize: true
    initialization-mode: always
    hikari:
      connection-timeout: 6000
      initialization-fail-timeout: 0
  jpa:
    database-platform: org.hibernate.dialect.H2Dialect
    hibernate:
      ddl-auto: update
      naming:
        physical-strategy:
org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl
    properties:
      hibernate:
        show_sql: true
        format_sql: true
        enable_lazy_load_no_trans: true
```

```
  h2:
    console:
      enabled: true

logging:
  level:
    root: INFO
```

Para comprobar que todo ha ido correcto:

HHH000412: Hibernate Core {#Version}
HHH000206: hibernate.properties not found
HCANN000001: Hibernate Commons Annotations {#Version}
HHH000400: Using dialect: org.hibernate.dialect.H2Dialect

H2 console on localhost: http://localhost:8090/h2-console/.

http://localhost:8090/h2-console/login.jsp?jsessionid=376764215f3bddb07db7509e97106a0c

- ## JPA Entity (pej. en carpeta models)

Para comprobar el correcto funcionamiento con H2

```
@Entity
public class Book {
@Id @GeneratedValue
private Long id;
```

```
private String name; // standard constructors // standard getters
and setters

}
```

H2 creará una tabla a partir de esta entidad.

To add some initial data to our application, we need to create a new SQL file, with some insert statements and put it in our resources folder. We can use **import.sql** (Hibernate support) or **data.sql** (Spring JDBC support) files to load data.

```
insert into book values(1, 'The Tartar Steppe');

insert into book values(2, 'Poem Strip');

insert into book values(3, 'Restless Nights: Selected Stories of
Dino Buzzati');
```

Crear Repository y Service

We'll continue creating the basic components in order to test our application. First, let's add the JPA Repository in a new *repositories* folder:

```
@Repository
public interface BookRepository extends JpaRepository<Book, Long>
{
}
```

We can use the *JpaRepository* interface from Spring framework, which provides a default implementation for the basic *CRUD* operations.

Next, let's add the *BookService* in a new *services* folder:

```
@Service
public class BookService {

   @Autowired
   private BookRepository bookRepository;

   public List<Book> list() {
      return bookRepository.findAll();
   }
}
```

To test our application, we need to check that the data created can be fetched from the *list()* method of the service.

We'll write the following *SpringBootTest*:

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class BookServiceUnitTest {

    @Autowired
    private BookService bookService;

    @Test
    public void whenApplicationStarts_thenHibernateCreatesInitialRecords() {
        List<Book> books = bookService.list();

        Assert.assertEquals(books.size(), 3);
    }
}
```

By running this test, we can check that Hibernate creates the *Book* data which are then fetched successfully by our service. That was it, Hibernate is running with Spring Boot.

○ Uppercase Table Name

Sometimes we may need to have the table names in our database written in uppercase letters. As we already know, **by default Hibernate will generate the names of the tables in lowercase letters**.

We could also try to explicitly set the table name, like this:

```
@Entity(name="BOOK")
public class Book {
    // members, standard getters and setters
}
```

However, that wouldn't work. What works is setting this property in *application.properties*:

```
spring.jpa.hibernate.naming.physical-strategy=org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl
```

As a result, we can check in our database that the tables are created successfully with uppercase letters.