Persistencia con Hibernate

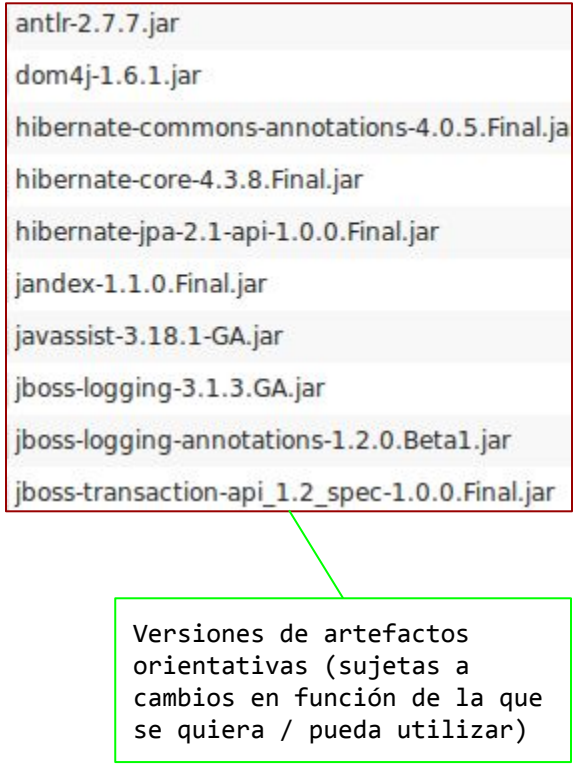
Realizado por A.Garay (dpto.de Informática)

Resumen de contenidos

- Instalación
- Configuración
- Concepto de POJO
- Acciones CRUD sobre POJO
- Composición de POJOs

Instalación (librerías)

- Descargar la última versión desde
  - https://hibernate.org/orm/releases/
- Abrir el “zip” y copiar los “jar” de la carpeta “lib/required” en “WEB-inf/lib”
- Opcionalmente, utilizar Maven y ajustar las dependencias en el pom.xml
  - http://hibernate.org/orm/



Instalación (Eclipse hibernate tools)

- Desde “Help->Eclipse marketplace” localizar “JBoss tools” y escoger las de tu versión de Eclipse: luna, mars, neon, etc.
- Instalar hibernate-tools

El fichero hibernate.cfg.xml (1/2)

- Es el principal fichero de configuración de Hibernate dentro de nuestro proyecto.
- Indica cómo conectarse a la Base de Datos (driver, nombreBD, dialecto SQL, etc.), así como las clases que queremos que sean persistentes.
- Se puede crear “a mano” o ayudándonos del Wizard de Eclipse “New->other->hibernate configuration file”
- Se suele ubicar bajo la carpeta “src” (en la raíz del classpath)

El fichero hibernate.cfg.xml (2/2)

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
        <property name="hibernate.connection.url">jdbc:mysql://localhost/test</property>
        <property name="hibernate.connection.username">root</property>
        <property name="hibernate.default_schema">test</property>
        <property name="hibernate.dialect">org.hibernate.dialect.MySQL5Dialect</property>
        <property name="current_session_context_class">thread</property>
        <property name="cache.provider_class"> org.hibernate.cache.NoCacheProvider</property>
        <property name="show_sql"> true</property>
        <property name="hbm2ddl.auto">update</property>

        <mapping class = "pruebas.hibernate.Pojo" />
    </session-factory>
</hibernate-configuration>
```

POJO persistente

- Es una clase JAVA (un objeto de negocio) que queremos que persista.
- POJO (Plain Old Java Object), sugiere que cualquier clase “normal y corriente” es susceptible de persistir con hibernate.
- Tan sólo hay que:
  - Asegurarse de que tenga siempre un constructor sin parámetros
  - Asegurarse de que todos sus atributos sean privados y tengan sus respectivos getters y setters estándar públicos.
  - Añadir un atributo llamado id de tipo Long (será el identificador de objeto) y añadir sus getters y setters
  - “anotar” la clase POJO con @Entity
  - “anotar” el atributo “id” (o el getter de id) con @Id y con @GeneratedValue
  - incluir una entrada <mapping class=”p1.p2….pn.Pojo”/> en el fichero ***hibernate.cfg.xml***

Creando una sesión hibernate

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

- Es el objeto “**session**”, es el objeto principal a través del cual nos comunicaremos con hibernate para hacer cualquier operación de persistencia.
- Podemos ubicar este código en un método estático “getSession()” de una clase “helper” a la que podemos denominar “HibernateUtil”, o bien en un atributo protegido de una clase de la que heredemos

Persistir un POJO (save)

```
Transaction t = session.beginTransaction();
    Pojo p = new Pojo();
    session.save(p);
t.commit();
```

- Es necesario que exista el objeto session, y crear una transacción para acotar la operación de tipo “CUD”
- “p” podría haber sido obtenido de otra manera (paso por parámetro, atributo, etc.), no hay por qué crearlo siempre (es tan solo un ejemplo)
- Se puede controlar la existencia de campos repetidos anotando con ...
  - `@Column (unique = true)`

...aquellos atributos para los que no queramos valores repetidos.

Recuperar un POJO conociendo su id

```
Pojo p = session.get(Pojo.class, 1L);
```

Si no existe un Pojo asociado a ese id devuelve **null**
Si se utiliza el método “load” (en lugar de “get”) se tratan de forma distinta los Pojo’s no encontrados

Introducir aquí el id conocido. Si es una constante, cualificarla con “L” para que sea long.
SI es una variable, que sea de tipo Long

Recuperar una lista de POJO's

```
List<Pojo> pojos = session.createQuery("from Pojo").list();
for ( Pojo pojo : pojos ) {
    pojo.metodoDelPojo();
}
```

- Utilizaremos un lenguaje de consultas propio de Hibernate, llamado **HQL** (hibernate query language)
- En su forma más sencilla, la consulta HQL "*from <Clase>*" me devuelve una lista de todos los objetos almacenados en la BD de una determinada clase.

Preparación / relleno de sentencias

```
List<Pojo> pojos =
        ss  .createQuery("from Pojo where campo = :valor")
            .setParameter("valor", valorConcreto)
            .list();
```

Más información acerca de binding de parámetros aquí

Recuperar una lista de resultados

```
List<Object[]> filas = session.createQuery("consulta HQL").list();
for ( Object[] fila : filas)
        {
              for (Object columna : fila ) {
                  System.out.print(columna + " // " );
               }
              System.out.println();
       }
```

- Como HQL puede ser complejo (incluir claúsula SELECT, JOIN entre objetos), podemos recoger esa lista como una lista de array de Object, en lugar de Pojo
- Accederemos a cada "columna" iterando sobre el array de Object, y apoyándonos en el método toString() para obtener una versión imprimible del dato.

Actualizar un POJO (merge)

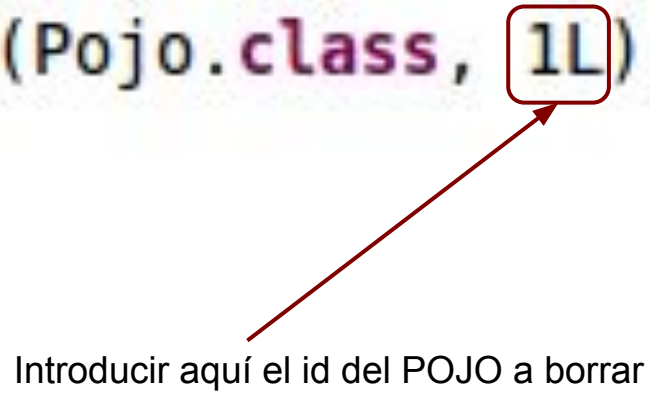
```
t = session.beginTransaction();
p = (Pojo)session.load(Pojo.class, 1L);
p.setNombre("Nuevo nombre");
session.merge(p);
t.commit();
```

Introducir aquí el id del POJO a actualizar

Aquí haremos los cambios que consideremos oportunos en el POJO

Borrar un POJO (delete)

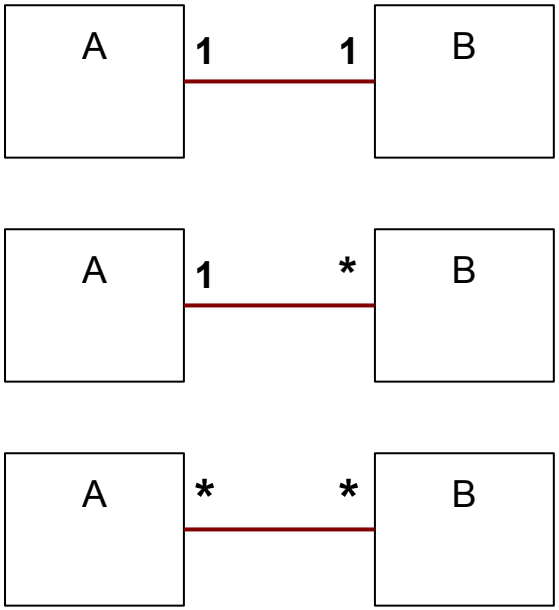
```
t = session.beginTransaction();
p = (Pojo)session.load(Pojo.class, 1L);
session.delete(p);
t.commit();
```



Composición de POJO's

- Trabajar con un único POJO, o con POJO's que no se relacionen con otros es relativamente sencillo, y no harían imprescindible un framework de persistencia.
- El "problema" viene cuando un POJO se relaciona con otros, y esa relación puede ser de tres tipos.
  - Uno a uno (1:1)
  - "Uno a muchos" ó "muchos a uno". (1:N ó N:1)
  - Muchos a muchos (N:M)
- Aunque no es imprescindible, "Hibernate" recomienda fehacientemente que las relaciones entre los POJO's sean "bidireccionales" (visibilidad mutua) y "coherentes" (si yo te veo a ti, tú me ves a mi) antes de persistirlos.

Implementando la composición



A	B

public class A {
@OneToOne(mappedBy="a" ....)
private B b;
public B getB() {...}
public void setB(B b) {...}
}
	public class B {
	@OneToOne(cascade=CascadeType.PERSIST, .)
	private A a;
	public A getA() {...}
	public void setA(A a) {...}
	}

public class A {
@OneToMany(mappedBy="a", ...)
private Collection<B> bs;
public Collection<B> getBs() {...}
public void setBs(Collection<B> bs) {...}
}
	public class B {
	@ManyToOne(....)
	private A a;
	public A getA() {...}
	public void setA(A a) {...}
	}

public class A {
@ManyToMany(mappedBy="as")
private Collection<B> bs;
public Collection<B> getBs() {...}
public void setBs(Collection<B> bs) {...}
}
	public class B {
	@ManyToMany(cascade=CascadeType.PERSIST...)
	private Collection<A> as;
	public Collection<A> getAs() {...}
	public void setAs(Collection<A> as) {...}
	}

Modos “cascade”

- Indican cómo se comportarán los POJO’s dependientes de otros POJO’s cuando se actúe sobre ellos.
  - CascadeType.**PERSIST** : save() or persist() persistirán los POJO’s dependientes
  - CascadeType.**MERGE** : merge() provocará que los POJO’s relacionados se actualicen.
  - CascadeType.**REFRESH** : refresh() operation, provocará que los POJO’s relacionados se refresquen.
  - CascadeType.**REMOVE** : delete() provocará que se borren los POJO’s relacionados
  - CascadeType.**DETACH** : desasocia los POJO’s relacionados si se desasocian “a mano”.
  - CascadeType.**ALL** : Un alias para indicar que aplican todos de tipos de cascada.
- Estos modos se indican con el atributo cascade=CascadeType.??? en la anotación @OneToOne, @ManyToOne, @OneToMany ó @ManyToMany correspondiente.
- Se pueden seleccionar varios modos cascade incluyéndolos entre llaves y separándolos por comas.

Borrado de “huérfanos”

- Sirve para que al desasociarse de un objeto en el mundo “objetual” se borre dicho objeto en la BD en el caso de que se haya quedado “huérfano”, es decir que ninguna otra entidad “apunte” a él.
- Se consigue indicando en la anotación @OneToOne, @ManyToOne o @OneToMany, el atributo orphanRemoval=true
- Más información acerca de tipos “cascada” y “huérfanos” aquí

Modos de materialización

- En cada POJO se puede indicar el modo de materialización, que indica cuándo “subirán” a memoria los POJO’s dependientes cuando se realice una consulta
  - Ansiosa: fetch=FetchType.EAGER
    - Los POJO’s dependientes subirán inmediatamente a memoria.
  - Perezosa: fetch=FetchType.LAZY
    - Los POJO’s dependientes subirán a memoria cuando se acceda a ellos.

Relaciones uno a uno

- Sean dos clases: “Base” y “Uno”. Para establecer una relación persistente entre ambos deberemos:
  - Añadir a “Base” un atributo “Uno uno”.
  - Anotar getUno() por ejemplo, con
    - @OneToOne(cascade = {CascadeType.ALL}, fetch=FetchType.LAZY)
  - Añadir a “Uno” un atributo “Base base”
  - Anotar getBase() por ejemplo, con
    - @OneToOne(cascade = {CascadeType.ALL}, fetch=FetchType.LAZY)
  - Crear un objeto “uno” de clase “Uno” y un objeto “base” de clase “Base”
  - Utilizar base.setUno(uno) para que “base” tenga visibilidad sobre “uno”.
  - Utilizar uno.setBase(base) para que “uno” tenga visibilidad sobre “base”
  - Persistirlo con “**persist**(base)”.
- El atributo “cascade” de las anotaciones gestiona el nivel de dependencia de un objeto respecto a otro.
  - En este ejemplo, si cualquiera de los dos se crea o se borra, se creará o desaparecerá su “compañero”.
  - Si quisiéramos que los POJO’s fueran independientes, no indicaremos ningún atributo cascade, y tendremos que establecer las relaciones atómicamente.

Relaciones uno a muchos

- Similares a las “uno a uno”.
- La única diferencia sustancial es que en el lado “uno”, hibernate debe saber, cuál es el nombre del atributo relacionado del lado muchos, usando el atributo “mappedBy”, así:
- El resto de anotaciones y su significado son las mismas que las estudiadas en las relaciones OneToOne

```
public class MaU {
        private UaM uam;
        ...
        @ManyToOne
        public UaM getUam() {
        }
        ...
}
```

```
public class UaM {
        private Collection<MaU> maus;
        ...
        @OneToMany(mappedBy=”uam”)
        public Collection<MaU> getMaus() {
        }
        ...
}
```

Relaciones muchos a muchos

- Similares a las muchos a uno
- Un lado debe ser el “propietario” y el otro el “inverso”.
- El que hace el papel de inverso, debe llevar un atributo “mappedBy” apuntando al nombre del atributo “colección” del lado del “propietario”

Transient & detached, persist vs save

- Objetos detached, transient y persistent
- persist o save. ¿Cúal usar?