HIBERNATE

Para poder trabajar en eclipse con hibernate debemos instalar las “Jbos tools”, son muchas tools con sólo marcar hibernate es suficiente.

PRIMER PROYECTO

Los pasos a seguir son:

1. Lo primero que vamos a hacer es crear un nuevo proyecto, una vez que lo tengamos creado lo vamos a “Maverizar”. (https://openwebinars.net/blog/que-es-apache-maven/) Para usar Hibernate necesitamos incluir un número muy elevado de librerías por lo que es mucho más fácil dejar que Maven se descargue de esas librerías por nosotros. Para crear un proyecto Maven debemos desplegar el menú del proyecto con el botón derecho y en **Configuración** seleccionar **Convert to Maven.** Esto lo que hará es crearnos un fichero pom.xml en el que deberemos poner las dependencias que vayamos a necesitar.
2. Nuestro fichero pom.xml deberá ser algo como

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
	<modelVersion>4.0.0</modelVersion>
	<groupId>HibernateInicial</groupId>
	<artifactId>HibernateInicial</artifactId>
	<version>0.0.1-SNAPSHOT</version>
	<build>
		<sourceDirectory>src</sourceDirectory>
		<plugins>
			<plugin>
				<artifactId>maven-compiler-plugin</artifactId>
				<version>3.8.0</version>
				<configuration>
					<release>11</release>
				</configuration>
			</plugin>
		</plugins>
	</build>
	<dependencies>
		<dependency>
			<groupId>org.hibernate</groupId>
			<artifactId>hibernate-core</artifactId>
			<version>5.4.14.Final</version>
		</dependency>
	</dependencies>
</project>
```

El control ojdbc Oracle no lo podemos añadir en el pom.xml porque Oracle no nos deja descargar su drive desde Maven (hay que registrarse en Oracle), por lo que tendremos que añadirlo de forma manual. Si trabajamos con Mysql si podemos añadirlo en nuestro pom.xml.

3. Debemos crear un Source Folder llamado *resources*

4. En esta ruta vamos a crear el fichero de configuración de Hibernate, para ello New→ Others → ...Hibernate→ **Hibernate configuration File**

5. Rellenamos los datos que en este caso serán:

```
hibernate.connection.driver_class: oracle.jdbc.driver.OracleDriver
hibernate.connection.password: dummy
hibernate.connection.url:
    jdbc:Oracle:thin:@//localhost:1521/ORCLCDB.localdomain
hibernate.connection.username: dummy
hibernate.dialect: org.hibernate.dialect.Oracle12cDialect
bernate.show_sql: true
hibernate.format_sql: true
hibernate.hbm2ddl.auto: create
```

   Cuidado con la propiedad última porque hará que hibernate nos cree las tablas que sean necesaria, pero si ya existe las borrará y nos creará nuevas tablas, por lo que es útil para ejecutarlo la primera vez si no tenemos creado la base de datos pero luego hay que eliminarlo.

6. Cuando terminemos de general el fichero de configuración de Hibernate tendremos que hacer una pequeña modificación manual. Para ello nos vamos al fichero y nos desplegará tres pestañas **Source** aparecerá:

   ```
   <hibernate-configuration name="">
   ```
   Debemos borrar la parte de name="" para evitar que nos salga un warning, si no, lo podemos dejar pero sabiendo que eso no afecta al funcionamiento de Hibernate.

7. Ahora debemos crear la clase o clases que queremos mapear con una entidad e indicarle a Hibernate que es una clase que debe manejar el ORM para ello utilizaremos las **anotaciones**.
   Existen muchas anotaciones vamos a empezar con las más básicas, pero hay mucho más que deberemos investigar si vamos a utilizar Hibernate a fondo.
   Para indicar que una clase tiene que ser utilizada por el ORM debemos utilizar la anotación **@Entity** y sólo tendríamos que indicarle obligatoriamente cuál es la clave primaria con la anotación **@Id.** También podemos añadir el atributo **@Column** para indicar las columnas de la tabla, si bien este atributo no es obligatorio y se puede utilizar para definir la longitud, si permite nulos o no, etc. En nuestro módulo no lo vamos a usar, simplemente para que sepáis que existe por si lo veis en alguna documentación.

8. Debemos **mapear** la clase que acabamos de crear con una entidad de la base de datos. Si no existe la tabla o entidad nos lo creará siempre y cuando hayamos puesto `hibernate.hbm2ddl.auto` aunque hay que tener cuidado porque si existe nos lo borrará. Desde el asistente de configuración de hibernate en el apartado de mapping añadimos la clase. Podemos ver que la clase también esta añadida en la pestaña del código fuente.

CRUD (Create, Read, Update, Delete)

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.

Ahora podríamos crear nuestra clase o Main que se encargará de crear los objetos que se mapearán en la base de datos. Para ello vamos a seguir los siguientes pasos:

1. Iniciamos un SessionFactory con la configuración definida anteriormente.
```
StandardServiceRegistry sr = new StandardServiceRegistryBuilder().configure().build();

SessionFactory sf = new MetadataSources(sr).buildMetadata().buildSessionFactory();
```

2. Abrimos una sesión.
```
Session session = sf.openSession();
```

3. Creamos nuevas instancias, para crear, leer, modificar y borrar los datos.

4. Iniciamos una transacción.
```
session.getTransaction().begin();
```

5. Realizamos la operación que deseamos.

6. Commiteamos la transacción.
```
session.getTransaction().commit();
```

7. Cerramos los objetos.
```
session.close();
sf.close();
```

Guardar

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

```
Usuario user2 = new Usuario();
user2.setId(1);
user2.setUsername("juan");
user2.setUserMessage("Hola juan ");
session.getTransaction().begin();
session.save(user);
session.getTransaction().commit();
```

Leer

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

```
Usuario user1 = (Usuario) session.get(Usuario.class,1);
System.out.println("El usuario " + user1);
```

Actualizar

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

```
Usuario user1 = (Usuario) session.get(Usuario.class,1);
System.out.println("El usuario " + user1);
user1.setUsername("Inma Olias");

session.getTransaction().begin();
session.update(user1);
session.getTransaction().commit();
```

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.

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

```
Usuario user1 = (Usuario) session.get(Usuario.class,3);
System.out.println("El usuario " + user1);

session.getTransaction().begin();
session.delete(user1);
session.getTransaction().commit();
```

HIBERNATE RELATION

En Hibernate (y en general en las bases de datos) existen 4 tipos de relaciones:

- Uno a Uno (@OneToOne Relation)
- **Uno a Muchos (@OneToMany Relation)**
- **Muchos a Uno (@ManyToOne Relation)**
- Muchos a Muchos (@ManyToMany Relation)

Si le ponemos direcciones a estas relaciones (unidireccional, o bidireccional) tendremos 7 tipos de relaciones:

- Uno a uno unidireccional
- Uno a uno bidireccional
- Uno a muchos unidireccional
- **Uno a muchos bidireccional**
- Muchos a uno unidireccional
- muchos a muchos unidireccional
- **Muchos a muchos bidireccional**

La relación muchos a uno bidireccional es igual que la relación uno a muchos bidireccional.

Many-To-One Relation (Muchos a uno)

Se trata de la asociación más sencilla y común de todas. Este tipo de asociaciones a conocida en algunos contextos como una relación padre/hijo, donde el lado muchos es el hijo y el lado uno es el padre.

Son las típicas relaciones 1:N. En la clase que modele la entidad que tiene la FK debemos especificar @ManyToOne y @JoinColumn(name = "campoFK"). Por ejemplo si queremos añadir una tabla teléfono que se unirá a nuestra tabla usuario, seria:

```
@Entity
public class Telefonos {
     @Id
     private String Telefono;
     @ManyToOne
     private Usuario idUsuario;
 . . . . . )
```

En este caso tendríamos que añadirle a la clase teléfono el usuario correspondiente.

One-To-Many Relation (Uno a muchos) La que vamos a utilizar**

La asociación Uno-a-muchos nos permite enlazar, en una asociación padre/hijo, el lado padre con todos sus hijos. Para ello, en la clase debemos colocar una colección de elementos hijos. Las asociaciones 1 a N se pueden modelar con una relación @ManyToOne o con una relación @OneToMany que además podría ser bidireccional o unidireccional.

Si la asociación @OneToMany no tiene la correspondiente asociación @ManyToOne, decimos que es unidireccional. En caso de que sí exista, decimos que es bidireccional. Se recomiendan relaciones bidireccionales porque las búsquedas en ambos sentidos es más rápida. (http://www.juntadeandalucia.es/servicios/madeja/contenido/libro-pautas/46#Asignaciones_de_asociacion)

Por todo ello **no** vamos a usar las relaciones @ManyToOne como lo hemos visto en el apartado anterior, sino que usaremos la relación @OneToMany bidireccionales (que incluyen una @ManyToOne).

La asociación @OneToMany bidireccional necesita de una asociación @ManyToOne en el lado hijo.

Toda asociación bidireccional debe tener un lado propietario (lado hijo). El otro lado vendrá referenciado mediante el atributo mappedBy.

Para ello tendremos en la parte del hijo

```
@Entity
public class Telefonos {
	@Id
	private String Telefono;
	@ManyToOne
	private Usuario usuario;

	public Telefonos(String telefono, Usuario id) {
		Telefono = telefono;
		this.idUsuario = id;
	}

	public Telefonos() {}

	public String getTelefono() {return Telefono; }

	public Usuario getIdUsuario() { return usuario;}

	public void setIdUsuario(Usuario idUsuario) {
			this.usuario = idUsuario;
	}

	public void setTelefono(String telefono) {
		Telefono = telefono;
	}

}
```

Es decir ponemos la anotación @ManyToOne pero sin la anotación @JoinColumn

En la parte del padre debemos añadir una List (debe especificarse como List, Collection o Set) para el conjunto de teléfonos del usuario.

```
@Entity
public class Usuario {
	@Id
	private int id;
	private String username;
	private String userMessage;
	@OneToMany(mappedBy ="usuario", cascade = CascadeType.ALL,
orphanRemoval = true)
	private List<Telefonos> phones;

	public Usuario() { phones = new ArrayList<>();  }

	public int getId() { return id; }

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

	public String getUsername() { return username;  }

	public void setUsername(String username) {
		this.username = username;
	}

	public String getUserMessage() {
		return userMessage;
	}
}
```

Deberemos añadir los métodos para añadir los teléfonos y sólo nos haría falta salvar los usuarios.

```
	public void addPhone(Telefonos phone) {
	    phones.add(phone);
	    phone.setIdUsuario(this);
	}

	public void removePhone(Telefonos phone) {
	    phones.remove(phone);
	    phone.setIdUsuario(null);
	}
```

Many-To-Many Relation (Muchos a Muchos bidireccional)

Un caso que suele presentarse en muchas ocasiones con las asociaciones muchos a muchos es que necesitamos añadir un atributo que no es de ninguna de las dos entidades, sino de la asociación en sí. Incluso aunque no tenga atributos la relación nos recomiendan que no usemos la relación @ManyToMany sino romper la asociación

@ManyToMany en ambos extremos en dos asociaciones que den el mismo resultado: @ManyToOne + @OneToMany.

Los pasos a seguir son:

1. Generar una nueva entidad para la relación
2. Romper la asociación @ManyToMany en ambos extremos en dos asociaciones que den el mismo resultado: @ManyToOne + @OneToMany.
3. Manejar de forma conveniente la clave primaria de esta nueva entidad. Al ser una clave primaria compuesta, necesitaremos de una clase extra, EntidadRlaciónId, y de la anotación @IdClass, para poder manejarla.

La anotación @Id solo está permitida, en primera instancia, para claves primarias simples, por lo que una entidad, por norma, no puede tener dos atributos anotados con @Id. Para poder manejar una clave primaria compuesta, JPA nos obliga a utilizar alguna estrategia diferente, como @IdClass o @EmbeddId. Nosotros optamos por la primera. Para ello, creamos una clase que tendrá los campos que conforman la clave primaria y que cumplirá con las siguientes características:

1. Debe ser una clase pública
2. Debe tener un constructor sin argumentos
3. Debe implementar Serializable
4. No debe tener clave primaria propia
5. Debe implementar los métodos equals y hashCode.

HIBERNATE QUERY

Hasta ahora nos hemos dedicado a ver las diversas formas de persistencia que soporta Hibernate en función de nuestro modelo de objetos de negocio en Java. Pero una característica fundamental de cualquier ORM es la necesidad de leer dichos objetos de la base de datos.

Hibernate tiene el objeto Query que nos da acceso a todas las funcionalidades para poder leer objetos desde la base de datos. Veamos ahora un sencillo ejemplo de cómo funciona y aunque no vamos a ver todas las funcionalidades de la clase Query.

```
Query query = session.createQuery("SELECT p FROM Alumno p");
List<Alumno> lalumnos = query.getResultList();

for (Alumno alumno : lalumnos) {
      System.out.println(alumno.toString());
}
```

Con esto podemos obtener todos los alumnos. Para aquellas consultas que devuelven más de un resultado, tenemos a nuestra disposición el método getResultList(); si la consulta devuelve un solo resultado, tendremos entonces que llamar a getSingleResult().

Si queremos pasar un parámetro lo haremos con la opcion query.setParameter

```
query = session.createQuery("SELECT p FROM Alumno p where nombre
like :consulta ");
query.setParameter("consulta", "L%");
lalumnos = query.getResultList();

for (Alumno alumno : lalumnos) {
System.out.println(alumno.toString());
}
```

Bibliografía

- http://www.cursohibernate.es/doku.php?id=unidades:05_hibernate_query_language:01_query

- Curso Hibernate y JPA de OpenWebinars