CONEXIÓN A BASES DE DATOS RELACIONALES (HIBERNATE)

Acceso a Datos

3 DE NOVIEMBRE DE 2024

ERIK AMO TOQUERO

Miniintroducción

Hibernate es un plugin creado para manejar distintos tipos de bases de datos relacionales de forma más sencilla. Necesita crear un archivo de configuración al que se accederá más tarde, además de necesitar ambos plugins, tanto el suyo como el del gestor de bases d datos que vaya a usar.

Plugin:

```
<!--Dependencia de Hibernate-->
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>5.6.9.Final</version>
</dependency>
<!--Dependencia de MySQL Connector-->
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>8.0.33</version>
</dependency>
```

Archivo de configuración:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
       "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
       "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
    <property
name="connection.driver_class">com.mysql.jdbc.Driver</property> <!--El
driver de acceso a la base de datos-->
    <property
name="connection.url">jdbc:mysql://localhost/cochesmultashibernate</pr
operty> <!--La base de datos a la que me voy a conectar-->
    <property name="connection.username">root</property> <!--Usuario--
>
    <property name="connection.password">toor</property> <!--
Contraseña-->
    <property
name="dialect">org.hibernate.dialect.MySQL5Dialect</property> <!--
Dialecto de las sentencias-->
    <property name="hibernate.show_sql">true</property> <!---->
    <property name="hibernate.jdbc.batch_size">50</property>
    </session-factory>
</hibernate-configuration>
```

Acceso a la Base de Datos

Además del objeto CRUD que hay que hacer con los métodos, hay que especificar en el propio objeto una serie de valores de Hibernate:

```
@Entity
@Table(name = "coches")
public class Coche implements Serializable {
    /*
  TABLE coches (
  id integer NOT NULL AUTO_INCREMENT,
  matricula varchar(50) DEFAULT NULL,
  marca varchar(50) DEFAULT NULL,
  modelo varchar(50) DEFAULT NULL,
  tipo varchar(50) DEFAULT NULL,
  PRIMARY KEY (id)
     */
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private int id;

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

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

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

    @Column(name = "tipo")
    private String tipo;
}
```

- **Id:** Identificador, clave primaria de la tabla
- **GeneratedValue:** Se utiliza cuando la tabla tiene un autoincremento
- **Column:** Columnas de la tabla

Importante tener creados getter, setter y constructor vacío.

Luego, se tiene que crear una **Session**, que es lo que utiliza Hibernate para acceder a la base de datos

```
public void generarSesion() {
    Configuration configuration = new Configuration();

configuration.configure(R.getHibernateConfig("hibernate.cfg.xml"));
    configuration.addAnnotatedClass(Multa.class);
    configuration.addAnnotatedClass(Coche.class);

    SessionFactory factory = configuration.buildSessionFactory();

    Session session = factory.openSession();

}
```

Las AnnotatedClass son las distintas clases a las que accede Hibernate, las cuales deben tener anotaciones como las anteriores.

Métodos CRUD

Los CRUD son más sencillos si está todo bien hecho, ya que son iguales

1. Empezar transacción
2. Realizar acción
3. Commit
    a. Si salta excepción, rollback
4. Limpiar sesión (session.clear(), libera memoria y evita errores de objetos)

Inserción

Inserta contenido en la base de datos

```
public void insertarCoche(Coche c) {
    try{
        session.beginTransaction();
        session.save(c); //Inserta
        session.getTransaction().commit();
    }catch (Exception e){
        System.out.println("Error de BD Hibernate");
        session.getTransaction().rollback(); //Rollback
    }
    session.clear();
}
```

Edición

Cambia contenido en la base de datos

```
public void editarCoche(Coche c) {
    try{
        session.beginTransaction();
        session.update(c); //Inserta
        session.getTransaction().commit();
    }catch (Exception e){
        System.out.println("Error de BD Hibernate");
        session.getTransaction().rollback(); //Rollback
    }
    session.clear();
}
```

Eliminación

Borra contenido de la base de datos

```
public void borrarCoche(Coche c) {
    try{
        session.beginTransaction();
        session.remove(c); //Inserta
        session.getTransaction().commit();
    }catch (Exception e){
        System.out.println("Error de BD Hibernate");
        session.getTransaction().rollback(); //Rollback
    }
    session.clear();
}
```

Búsqueda

Busca contenido en la tabla y lo devuelve (**No usa necesariamente transactions, usa Queries)**

```
public Coche buscarCoche(String matricula) {
    Coche c = null;
    try{
        c = session.createQuery(" from Coche where matricula =
'"+matricula+"'", Coche.class).uniqueResult();
        //Hace una query y coge el único resultado de tipo Coche
    }catch (Exception e){
        session.clear(); //Limpia la sesión
        System.out.println("Error de BD Hibernate");
    }
    return c;
}
```

Nota: Para las queries cuando pones FROM, a continuación, **tiene que ir el nombre del objeto, no el de la tabla**

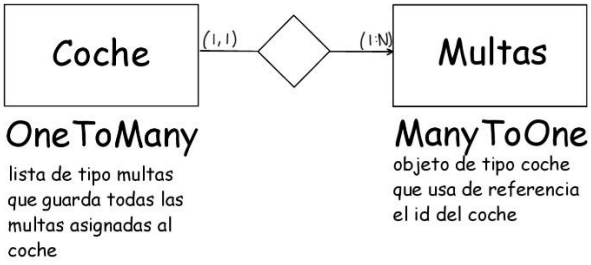
Listado

Busca varios objetos en la tabla y lo devuelve en una lista

```
public List<Coche> listarCoches() {
    List<Coche> coches = new ArrayList<>();
    try{
        coches = session.createQuery(" from Coche",
Coche.class).getResultList(); //Coge la query entera
    }catch (Exception e){
        session.clear(); //Limpia la sesión
        System.out.println("Error de BD Hibernate");
    }
    return coches;
}
```

1:N con Hibernate

Como es una base de datos relacional, trabaja también con relaciones 1:N y N:N. En este caso, las relaciones 1:N Funcionan de la siguiente forma:



OneToMany y ManyToOne son dos anotaciones de Hibernate para referenciar a una relación 1:N. OneToMany va a la que es 1,1 y ManyToOne va a la que es 1,N

Quedaría algo tal que así en las clases:

```
@Entity
@Table(name = "coches")
public class Coche implements Serializable {
    /*
  TABLE coches (
  id integer NOT NULL AUTO_INCREMENT,
  matricula varchar(50) DEFAULT NULL,
  marca varchar(50) DEFAULT NULL,
  modelo varchar(50) DEFAULT NULL,
  tipo varchar(50) DEFAULT NULL,
  PRIMARY KEY (id)
     */
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private int id;

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

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

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

    @Column(name = "tipo")
    private String tipo;
    @OneToMany(mappedBy = "coche", cascade = CascadeType.ALL)
//OneToMany: Lo tiene el de la relación 1
    private List<Multa> multas; //Un coche puede tener varias
multas
}
```

```java
@Entity
@Table(name = "multas")
public class Multa implements Serializable {
  /*
  TABLE multas (
  id_multa integer unsigned NOT NULL AUTO_INCREMENT,
  precio DOUBLE NOT NULL,
  fecha DATE DEFAULT NULL,
  matricula varchar(7) NOT NULL,
  PRIMARY KEY (id_multa)
  )
  */
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id_multa")
    private int id;

    @Column(name = "precio")
    private double precio;

    @Column(name = "fecha")
    private LocalDate fecha;

    @ManyToOne //ManyToOne: Lo tiene el de la relación de la N
    @JoinColumn(name = "matricula", referencedColumnName =
"matricula")
    private Coche coche;
}
```

- **OneToMany: Lista del objeto de la otra tabla**
  - **mappedBy:** Nombre del objeto que mapea de la otra clase
  - **Cascade:** El borrado/actualización que tiene Si usa on delete cascade o no)
- **ManyToOne: Objeto de la otra tabla**
  - **JoinColumn:** Columna que tiene la clave foránea
  - **name:** Nombre de la columna
  - **ReferencedColumnName:** Nombre de la columna de la otra tabla