IES DR LLUÍS SIMARRO
ENSENYAMENT PÚBLIC
XATIVA

Acceso a datos

UD05. Hibernate

Desarrollo de Aplicaciones Multiplataforma

Enric Giménez Ribes

e.gimenezribes@edu.gva.es

ÍNDICE

1. INTRODUCCIÓN

El **desfase objeto-relacional** surge cuando en el desarrollo de una aplicación con un lenguaje orientado a objetos se hace uso de una base de datos relacional. Hay que tener en cuenta que esta situación se da porque tanto los lenguajes orientados a objetos como las bases de datos relacionales están ampliamente extendidas.

En cuanto al desfase, ocurre que en nuestra aplicación Java (como ejemplo de lenguaje Orientada a Objetos) tendremos, por ejemplo, la definición de una clase cualquiera con sus atributos y métodos:

```
public class Personaje {
  private int id;
  private String nombre;
  private String descripcion;
  private int vida;
  private int ataque;

  public Personaje(...) {
       ...
  }
}
```

Mientras que en la base de datos tendremos una tabla cuyos campos se tendrán que corresponder con los atributos que hayamos definido anteriormente en esa clase. Puesto que son estructuras que no tienen nada que ver entre ellas, tenemos que hacer el mapeo manualmente, haciendo coincidir (a través de los getters y setters) cada uno de los atributos con cada uno de los campos (y viceversa) cada vez que queramos leer o escribir un objeto desde y hacia la base de datos, respectivamente.

```
CREATE TABLE personajes (
  id INT PRIMARY KEY AUTO_INCREMENT;
  nombre VARCHAR(50) NOT NULL,
  descripcion VARCHAR(50),
  vida INT DEFAULT 10,
  ataque INT DEFAULT 10;
);
```

Eso hace que tengamos que estar continuamente descomponiendo los objetos para escribir la sentencia SQL para insertar, modificar o eliminar, o bien recomponer todos los atributos para formar el objeto cuando leamos algo de la base de datos.

1.1. Object Relational Mapping (ORM)

Se conoce comúnmente como **ORM** a un componente software, ubicado entre una aplicación y un SGBD, que se encargará de salvar el desfase objeto-relacional, ayudándonos a transformar los objetos de nuestra aplicación en filas de tablas de nuestra base de datos relacional.

Un **mapeador objeto-relacional** nos permite abstraernos del proceso de transformación entre objetos y tablas, y viceversa, acelerando así el desarrollo, pues nos permite centrarnos en programar la lógica de negocio de nuestra aplicación. También permite aumentar la seguridad y fiabilidad.

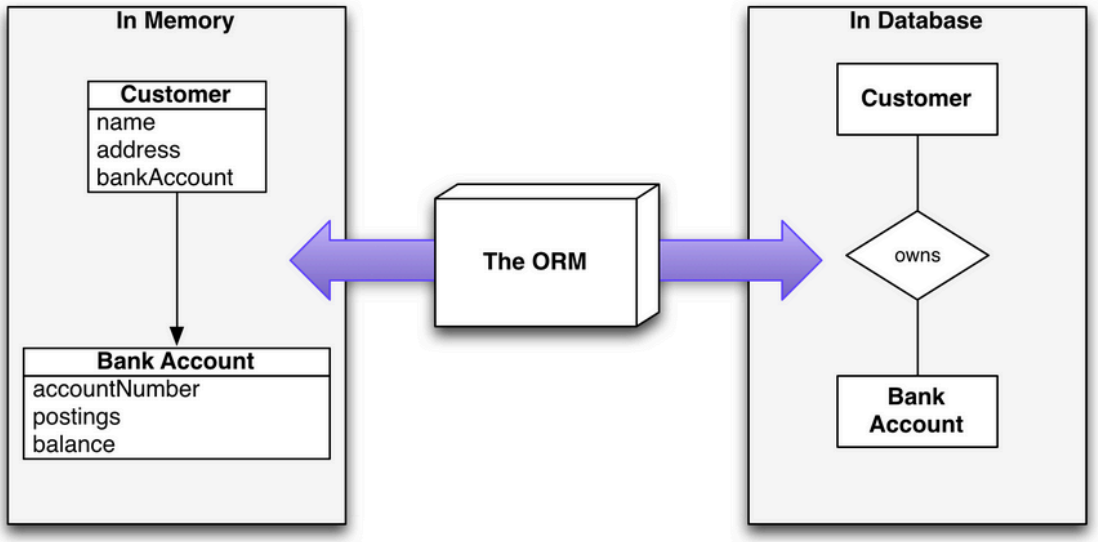
Una de las grandes desventajas achacadas a los ORMs es que pueden ralentizar mucho el acceso a bases de datos. Sin embargo, muchos ORMs nos permiten el uso de cachés, que aceleran el proceso de consulta.

1.1.1. Productos existentes en el mercado

Algunos de los ORMs más conocidos para los diferentes lenguajes de programación:

- **Java** → Hibernate, EclipseLink, JDO, MyBatis.
- **PHP** → Propel, Doctrine.
- **C#** → NHibernate, ADO.NET Entity Framework.
- **Python** → Django ORM, SQLAlchemy, Pony.

2. ¿QUÉ ES EL MAPEO OBJETO-RELACIONAL?



Por ejemplo, si trabajamos directamente con JDBC tendremos que descomponer el objeto para construir la sentencia INSERT del siguiente ejemplo:

```
...

String sentenciaSql = "INSERT INTO personajes (nombre, descripcion, vida, ataque)" +
  ") VALUES (?, ?, ?, ?)";
PreparedStatement sentencia = conexion.prepareStatement(sentenciaSql);
sentencia.setString(1, personaje.getNombre());
sentencia.setString(2, personaje.getDescripcion());
sentencia.setInt(3, personaje.getVida());
sentencia.setInt(4, personaje.getAtaque());
sentencia.executeUpdate();

if (sentencia != null)
  sentencia.close();

...
```

Si contamos con un framework como Hibernate, esta misma operación se traduce en unas pocas líneas de código en las que podemos trabajar directamente con el objeto Java, puesto que el framework realiza el mapeo en función de las anotaciones que hemos implementado a la hora de definir la clase, que le indican a éste con que tabla y campos de la misma se corresponde la clase y sus atributos, respectivamente.

```java
@Entity
@Table(name="personajes")
public class Personaje {

  @Id // Marca el campo como la clave de la tabla
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  @Column(name="id")
  private int id;

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

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

  @Column(name="vida")
  private int vida;

  @Column(name="ataque")
  private int ataque;

  public Personaje(. . .) {
       . . .
  }

  // getters y setters
}
```

Así, podemos simplemente establecer una sesión con la Base de Datos y enviarle el objeto, en este caso invocando al método save que se encarga de registrarlo en la Base de Datos donde convenga según sus propias anotaciones.

```java
...

sesion = HibernateUtil.getSession();
sesion.beginTransaction();
sesion.persist(personaje);
sesion.getTransaction().commit();
sesion.close();

...
```

3. HIBERNATE

En nuestro caso usaremos **Hibernate** como librería ORM. Ya hemos hablado en el punto anterior de lo que era un mapeador objeto-relacional y como nos podía ahorrar muchas líneas de código muy repetitivo. Sin embargo, los beneficios de uso de Hibernate no se quedan solamente ahí:

Ventaja	Descripción
Productividad	Nos permite centrarnos en el desarrollo de la lógica de negocio, ahorrándonos una gran cantidad de código.
Mantenibilidad	El ORM de Hibernate, al reducir el número de líneas de código que tenemos que escribir, nos permite realizar refactorizaciones de forma más fácil.
Rendimiento	Aunque la persistencia manual podría permitir un rendimiento mayor, Hibernate aporta muchas optimizaciones, como el almacenamiento en caché.
Independencia	Hibernate nos permite aislarnos del RDBMS concreto que estemos usando. Aunque no tengamos pensado cambiar de gestor, nos puede permitir usar un RDBMS ligero (como H2) para la fase de desarrollo, usando otro en producción.

Hibernate no solamente es un ORM. Se trata de un proyecto formado por varios módulos, que nos ofrecen grandes funcionalidades. Entre las que cabe destacar las siguientes:

- **Hibernate ORM:** Este módulo ya ha sido, de alguna forma, presentado. Contiene una serie de servicios para la persistencia de objetos en bases de datos relacionales. Funciona en casi cualquier proyecto Java (Java EE, Java Swing, con Servlets, Spring…).
- **Hibernate EntityManager:** Más adelante hablaremos de JPA, pero podemos decir aquí que este módulo es el que nos permite utilizar Hibernate como implementación del estándar JPA. Las características nativas de Hibernate son un superconjunto de las que ofrece JPA en todos los sentidos.
- **Hibernate Validator:** Hibernate proporciona una implementación del estándar JSR 303 Bean Validation. Se puede usar independientemente de otros proyectos de Hibernate, y proporciona una validación declarativa para nuestras clases modelo.
- **Hibernate Enver:** Nos permite auditar los cambios en nuestra persistencia, manteniendo múltiples versiones de los datos en nuestra base de datos SQL. Esto nos permitirá implementar fácilmente históricos (funciona de forma similar a los sistemas de control de versiones, como Git). Lo abordaremos más adelante.

- **Hibernate Search:** Este módulo mantiene un índice de todas los datos de nuestro dominio dentro de una base de datos de Apache Lucene. En conjunción con Hibernate ORM, podremos realizar búsquedas de texto completo.
- **Hibernate OGM:** Se trata del mapeador objeto-grid, y está orientado hacia bases de datos NoSQL. Proporciona soporte JPA para este tipo de soluciones, reusando el núcleo de Hibernate pero almacenando las entidades en pares clave-valor, documentos u otro tipo de almacenes de datos no relacionales.

3.1. Hibernate vs JDBC

JDBC no es un ORM, pero es la tecnología de base que nos ofrece Java para poder tener conectividad con bases de datos relacionales.

Tecnología	Ventajas	Desventajas
JDBC	Ofrece un rendimiento superior. Además, nos permite utilizar al máximo las funcionalidades concretas de nuestro servidor de base de datos.	El desarrollo y mantenimiento de nuestras aplicaciones es mucho más costoso. Además, es muy fácil introducir errores durante el desarrollo.
Hibernate	Podemos desarrollar más rápidamente, y haciendo uso de entidades (objetos) en lugar de consultas. Es una gran ayuda en la fase de mantenimiento, y nos permite eliminar al 100% los errores de ejecución debido al acceso a bases de datos.	Tiene una curva de aprendizaje mayor, y normalmente no conseguirá el mismo rendimiento que el acceso a base de datos de forma nativa.

3.2. Hibernate y JPA

Muchas personas que se han introducido en el mundo de la persistencia se preguntan: ¿no son Hibernate y JPA lo mismo? Realmente, la diferencia es mucha, ya que JPA es una especificación (inicialmente JSR 220; en la versión actual, la 2.1, el JSR 338) que carece de implementación. Hibernate, por contra, es un producto muy real, un framework con una serie de funcionalidades muy concretas.

Entonces, ¿qué relación tienen? Hibernate, a través de uno de sus módulos, proporciona una implementación la especificación JPA, de forma que podemos usar JPA como interfaz, sabiendo que por debajo, las operaciones con la base de datos se estarán realizando con Hibernate. De alguna manera, **Hibernate es de facto la implementación de referencia de JPA**.

La especificación JPA define lo siguiente:

- Una facilidad para especificar cómo nuestros objetos Java se relacionan con el esquema de una base de datos (a través de XML o de anotaciones).
- Una api sencillo para realizar las operaciones CRUD (create, read, update, delete), haciendo uso de un javax.persistence.EntityManager.
- Un lenguaje y un API para realizar consultas sobre los datos. El lenguaje, llamado JPQL (Java Persistence Query Language) se parece mucho a SQL.
- Como el motor de persistencia interacciona con las instancias transaccionales para buscar instancias sucias, captura de asociaciones u otras funciones de optimización.

Hibernate incluye, como parte de su código, toda la especificación JPA. Es decir, podemos usar Hibernate como motor de persistencia de JPA; sin embargo, incluye más funcionalidades que las que define la especificación.

3.3. Hibernate con Maven

Si queremos trabajar con Hibernate, tenemos que incorporar a nuestro proyecto Java una serie de librerías, que se pueden descargar fácilmente desde la web http://hibernate.org/orm/downloads/.

Sin embargo, el mantenimiento de estas librerías puede llevarnos al conocido como infierno de los Jar, ya que podemos tener algunos problemas:

- Incluir dos versiones diferentes de una misma biblioteca.
- Dos bibliotecas que están relacionadas entre sí, y de las que se incluyen versiones incompatibles.
- …

Por todas estas razones, entre otras, es razonable utilizar algún sistema que nos permita gestionar las dependencias de nuestro proyecto con librerías externas. En nuestro caso, usaremos **Maven**.

Utilizando **Maven**, nuestro proyecto tendrá un fichero, llamado **pom.xml** que, entre otros elementos, nos permitirá definir las librerías que necesitamos.

De esa forma, podremos añadir las dependencias de nuestras librerías. Estas pueden ser encontradas en la web del repositorio de Maven (https://mvnrepository.com/).

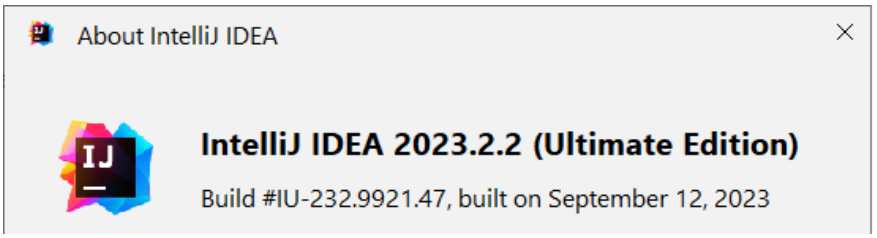
Alguna de las que usaremos durante el curso es la siguiente:

```
<!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-core -->
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>6.3.1.Final</version>
</dependency>
```

4. PRERREQUISITOS ANTES DE COMENZAR

Para poder comenzar con nuestro primer proyecto con Hibernate, debemos tener en cuenta los siguientes prerrequisitos, con respecto al software que vamos a utilizar.

- Como herramienta fundamental de desarrollo vamos a usar **IntelliJ IDEA**. En la escritura de estos apuntes tenemos instalado la versión 2023.2.2 (Ultimate Edition). Tener en cuenta que es muy importante tener la versión Ultimate para la ejecución de nuestros proyectos. Esta versión nos permitirá utilizar más adelante Spring, para implementar proyectos Web MVC con Hibernate y JPA. No nos sirve la versión "Community Edition".

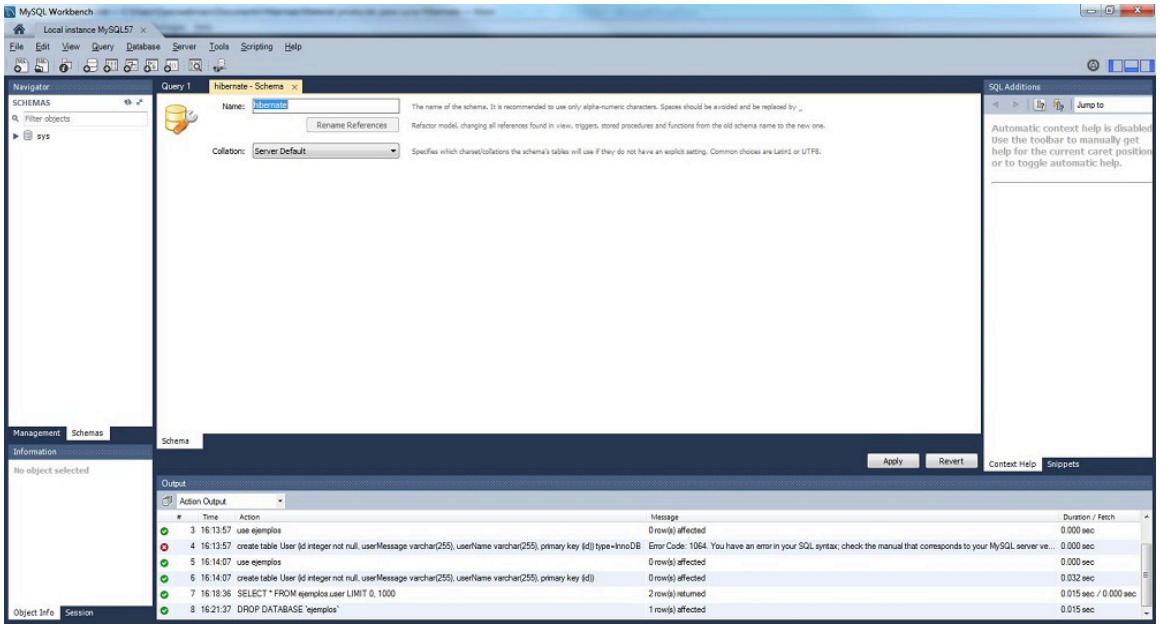


- **Mysql** será nuestro RDBMS de referencia. Las razones son muchas, y no vamos a entrar en detalle ahora mismo. Usaremos la versión community, que es de uso gratuito (si el alumno lo prefiere, puede utilizar MariaDB, que es 100% compatible con Mysql y además es software libre). Para descargar Mysql Community Version, podemos hacerlo desde su web: https://dev.mysql.com/downloads/mysql/.
- Aunque presuponemos unos ciertos conocimientos con bases de datos y SQL antes de iniciar este curso, trataremos de facilitar las operaciones de gestión con el RDBMS, como la creación de usuarios, ejecución de consultas de comprobación…
- La creación de la base de datos, usuarios… lo podemos hacer de diferentes formas:
    - A través del servidor web local: http://localhost/phpmyadmin
    - A través de alguna aplicación de escritorio: HeidiSQL(Windows), DBeaver…
    - A través de la herramienta gráfica oficial, llamada Mysql Workbench. También es gratuita, y se puede descargar desde la web de mysql: https://dev.mysql.com/downloads/workbench/.

4.1. Creación de un nuevo esquema o base de datos

Como punto de partida, vamos a crear un espacio en la base de datos donde podremos crear tablas e insertar datos. En la nomenclatura de MySQL, esto se llama esquema (schema). Un esquema puede ser utilizado por diferentes usuarios, con distintos niveles de privilegios.

Para crear un nuevo esquema desde MySQL Workbench tan solo tenemos que pulsar con el botón derecho del ratón sobre el panel Navigator, y seleccionar la opción Create Schema. A continuación, indicamos el nombre (por ejemplo, **adt5_ejemplo1**), y pulsamos sobre el botón Apply.

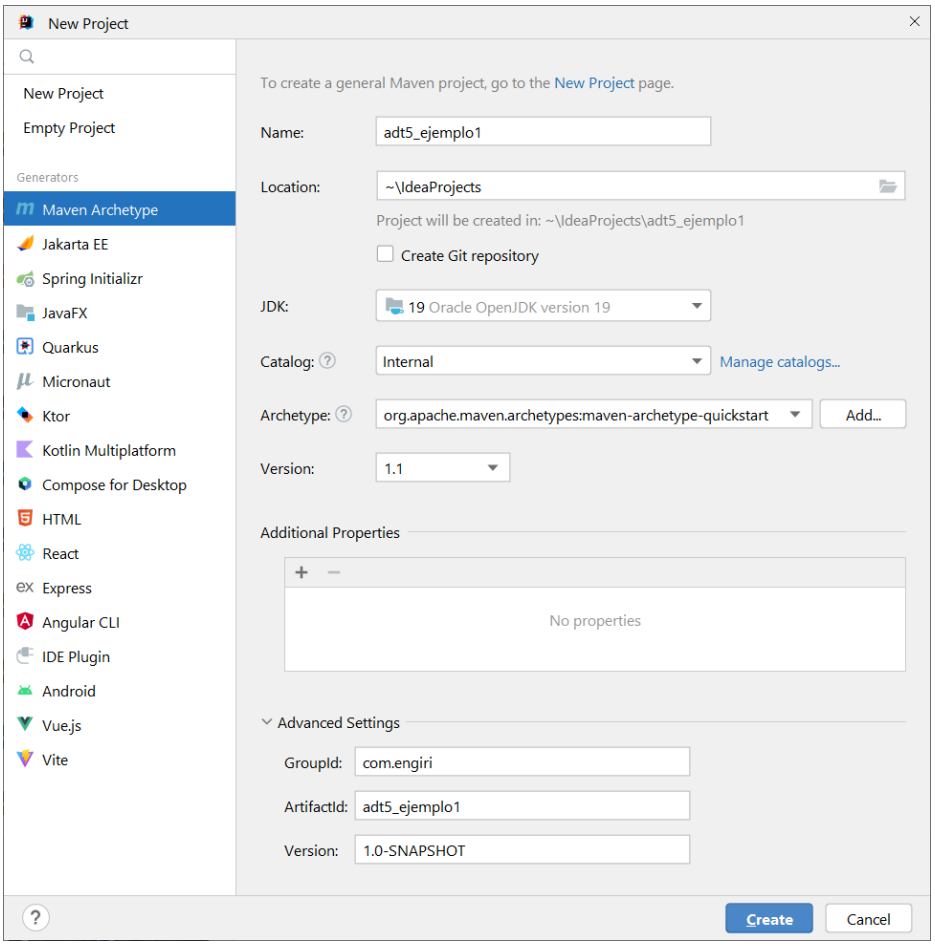


Nosotros haremos uso de este nuevo esquema con un usuario que hemos creado durante la instalación de MySQL que se llama **root** y cuya contraseña está vacía, por lo tanto, **“”**.

5. NUEVO PROYECTO

5.1. Creación y configuración inicial

Vamos a comenzar creando nuestro proyecto Maven:



Algunas propiedades que resaltar en la anterior pantalla:

- **Tipo de proyecto:** Maven Archetype
- **Archetype:** org.apache.maven.archetypes:maven-archetype-quickstart
    - Plantilla de proyecto predefinida que se utiliza como base: incluyen una configuración inicial, una jerarquía de directorios, y posiblemente archivos de configuración predefinidos.
- **GroupId:** com.engiri
- **ArtifactId:** adt5_ejemplo1

5.2. Configuración del fichero pom.xml

Ahora vamos a añadir las dependencias necesarias al fichero **pom.xml**, de forma que el apartado de dependencias queda de la siguiente forma:

```
<dependencies>
     <!-- Otras dependencias, como la de JUnit... -->
     <dependency>
          <groupId>org.hibernate</groupId>
          <artifactId>hibernate-core</artifactId>
          <version>6.3.1.Final</version>
     </dependency>
     <dependency>
          <groupId>com.mysql</groupId>
          <artifactId>mysql-connector-j</artifactId>
          <version>8.2.0</version>
     </dependency>
</dependencies>
```

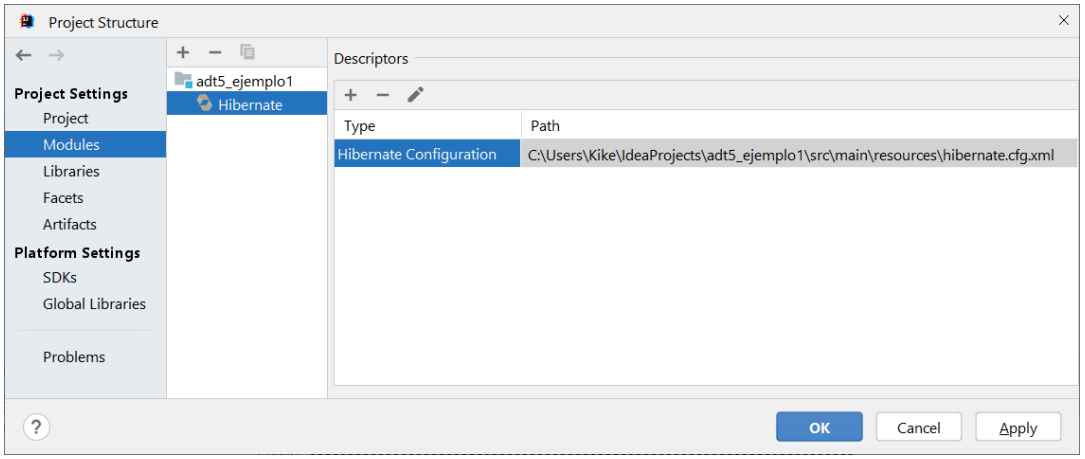
La primera dependencia es la que incluye de manera efectiva a Hibernate, entre otros módulos. La segunda es la que incluye las clases JDBC para conectar a Java con Mysql, y que son usadas por Hibernate.

5.3. Creación del fichero de configuración de Hibernate (hibernate.cfg.xml)

A continuación, creamos una carpeta de recursos, en la ruta /src/main/resources. Para ello, pulsamos sobre el proyecto con el botón derecho y **New > Directory**. Podemos escribir el nombre del directorio o nos sugiere dos opciones: **/src/main/resources** sería la elección correcta.

Ahora, debemos añadir a nuestro proyecto que vamos a trabajar con **Hibernate**. Para ello pulsamos sobre **File > Project Structure** y en el apartado de **Facets** añadimos Hibernate a nuestro proyecto. Es importante, añadir en el apartado **Descriptors** el archivo de configuración **hibernate.cfg.xml** con la ruta del directorio creado anteriormente.

NOTA: Se puede hacer tanto desde Facets como desde Modules.



Nuestro entorno de trabajo (IDE) nos habrá creado el fichero de configuración. Pero nosotros modificaremos el archivo con la siguiente información:

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
       "-//Hibernate/Hibernate Configuration DTD//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>
       <property name="connection.url">jdbc:mysql://localhost/adt5_ejemplo1</property>
       <property name="connection.username">root</property>
       <property name="show_sql">true</property>
       <property name="format_sql">true</property>
       <property name="hbm2ddl.auto">create</property>
       <mapping class="com.engiri.User"/>
   </session-factory>
</hibernate-configuration>
```

- En las primeras tres propiedades configuramos la conexión con el driver de MySQL, la ruta de donde tenemos el SGBD con el nombre de la base de datos y el usuario. Si necesitáramos la contraseña deberíamos indicarlo con la propiedad de "**connection.password**".
- La propiedad "**show_sql**" y "**format_sql**" indica que muestre por consola la instrucciones SQL con formato, nos puede venir bien para ver que hace Hibernate por nosotros.
- La propiedad "**hbm2ddl.auto**" a create: es la encargada de generar las tablas necesarias para albergar nuestro modelo, sin necesidad de que nosotros lo tengamos que hacer manualmente. En el tema siguiente, veremos todas las opciones disponibles (*create, create-drop, update, validate o borrar la propiedad*).

5.4. Definición de nuestras clases modelo

En este ejemplo, tan solo tendremos una clase modelo, cuyo contenido es el siguiente:

```
import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.Id;

@Entity
public class User {

   @Id
   private int id;

   @Column
   private String userName;

   @Column
   private String userMessage;

   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;
   }

   public void setUserMessage(String userMessage) {
       this.userMessage = userMessage;
   }

}
```

Aunque estamos trabajando con Hibernate nativo, usamos las anotaciones de JPA.

- **@Entity:** indica que esta clase es una entidad que deberá ser gestionada por el motor de persistencia de Hibernate.
- **@Id:** indica que, de todos los atributos, ese será tratado como clave primaria.
- **@Column:** indica que ese atributo es una columna de la tabla resultante.

Mapear la clase en hibernate

Debemos añadir la entidad dentro del archivo de configuración de hibernate, es decir, hacer el mapping de esta clase. Lo podemos hacer desde el asistente o desde el código fuente:

```
<mapping class="com.engiri.User"/>
```

5.4.1. Librería: javax vs jakarta

La diferencia entre **javax** y **jakarta** radica en la evolución y la historia de las bibliotecas de persistencia en Java, así como en las organizaciones que las mantienen.

1. **javax.persistence (Java EE 7 y versiones anteriores):**
    - Es un paquete que se utilizaba en las versiones anteriores de la plataforma Java Enterprise Edition (Java EE), que incluía tecnologías empresariales como EJB y JPA.
    - Hasta Java EE 7, JPA estaba bajo el paquete javax.persistence. Sin embargo, la evolución de la plataforma y la necesidad de avanzar en la dirección de la modularidad e independencia llevaron a cambios en la estructura de las bibliotecas.
2. **jakarta.persistence (Java EE 8 y posteriores):**
    - A partir de Java EE 8, las especificaciones Java EE pasaron a formar parte del proyecto Eclipse Foundation bajo el nombre "Jakarta EE" debido a la necesidad de evolucionar la plataforma de forma independiente de Oracle (que anteriormente estaba bajo el control de Java EE).
    - Como parte de esta transición, las bibliotecas y paquetes que solían pertenecer a javax se movieron a jakarta. Por lo tanto, JPA, junto con otras especificaciones, se encuentra ahora bajo el paquete jakarta.persistence.

5.5. Clase de aplicación

Por último, nos falta implementar el código necesario para cargar la configuración del fichero hibernate.cfg.xml y realizar las operaciones necesarias.

```
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class App
{

   public static void main( String[] args )
   {
       // Configurar Hibernate
       Configuration config = new Configuration().configure("hibernate.cfg.xml");
       SessionFactory sessionFactory = config.buildSessionFactory();

       // Crear una nueva sesión de Hibernate
       Session session = sessionFactory.openSession();

       // Construimos un objeto de tipo User
       User user1 = new User();
       user1.setId(1);
       user1.setUserName("Pepe");
       user1.setUserMessage("Hola mundo Pepe");

       // Iniciar una transacción
       session.beginTransaction();

       // Guardar el user1 en la base de datos
       session.persist(user1);

       // Commit de la transacción
       session.getTransaction().commit();

       // Cerrar la sesión
       session.close();
       sessionFactory.close();
   }

}
```

Si comprobamos en MySQL, tendremos una nueva tabla, llamada user con una fila insertada.

5.6. HibernateUtil

Podemos hacer una clase para tener todas las operaciones que utilizamos en Hibernate:

```
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class HibernateUtil {

   private static final SessionFactory sessionFactory;

   static {
       try {

           // Crea la SessionFactory utilizando la configuración de hibernate.cfg.xml
           Configuration configuration = new Configuration();
           configuration.configure("hibernate.cfg.xml");

           // Crea una factoría
           sessionFactory = configuration.buildSessionFactory();

       } catch (Throwable ex) {
           // Maneja los errores
           System.err.println("Error en la creación de la SessionFactory: " + ex);
           throw new ExceptionInInitializerError(ex);
       }
   }

   // Devuelve una sesión de Hibernate
   public static Session getSession() {
       return sessionFactory.openSession();
   }

   // Cierra Hibernate
   public static void closeSessionFactory() {
       if (sessionFactory != null)
           sessionFactory.close();
   }
}
```

Con esta versión, puedes utilizar **HibernateUtil.getSession()** para obtener una instancia de Session en tu código. Asegúrate de manejar y cerrar la Session adecuadamente cuando ya no la necesites para evitar problemas de recursos.

5.6.1. El bloque static

El **bloque static** de inicialización es un bloque de código en una clase Java que se ejecuta una sola vez cuando la clase se carga por primera vez en la JVM (Máquina Virtual de Java). Este bloque se utiliza para inicializar variables o realizar tareas de inicialización que deben realizarse solo una vez, independientemente de cuántas instancias de la clase se creen.

El **bloque static** de inicialización se declara dentro de la clase utilizando la palabra clave **static** y se coloca dentro de llaves **{ }**. Por lo general, se coloca después de las declaraciones de variables estáticas en la clase. Aquí hay un ejemplo de cómo se ve un bloque static de inicialización:

```
public class Ejemplo {
    // Variable estática
    static int numeroEstatico;

    // Bloque static de inicialización
    static {
        // Inicialización de la variable estática
        numeroEstatico = 42;

        // Otras tareas de inicialización que deben realizarse una vez
    }

    // Resto de la clase
}
```

- **Se ejecuta una sola vez:** El bloque static de inicialización se ejecuta una sola vez, cuando la clase se carga en la memoria, antes de la creación de cualquier instancia de la clase.
- **No es necesario llamarlo:** No es necesario llamar explícitamente el bloque static de inicialización. Se ejecutará automáticamente cuando se cargue la clase.
- **Útil para inicializar variables estáticas:** Para inicializar variables estáticas, como constantes o configuraciones que deben configurarse una vez para todas las instancias de la clase.
- **Puede contener múltiples bloques:** Una clase puede contener múltiples bloques static de inicialización, que se ejecutarán en el orden en que aparecen en la clase.

En resumen, el bloque static de inicialización es una parte importante de Java para inicializar datos estáticos o realizar tareas de inicialización específicas de la clase que deben ejecutarse sólo una vez.

6. MAPEO DE ENTIDADES CON CLASES JAVA

En el caso de las entidades, se deben anotar tanto la propia clase como cada uno de los atributos (se puede hacer en el atributo o en su getter/setter) para indicar con qué tabla mapearla y cómo mapear los atributos con los campos que corresponda, respectivamente.

Las anotaciones que nos podemos encontrar para anotar una clase Java que debe ser mapeada con una tabla son:

- **@Entity:** Indica que la clase es una tabla en la base de datos. Es decir, una entidad a persistir.
- **@Table(name = "nombre_tabla", catalog = "nombre_base_datos"):** Indica el nombre de la tabla y la base de datos a la que pertenece (este último parámetro no es necesario ya que esa información viene en el fichero de configuración).

En el caso de los atributos simples que deben ser mapeados con los campos de la tabla correspondiente:

- **@Id:** Indica que un atributo es la clave primaria.
- **@GeneratedValue(strategy=GenerationType.IDENTITY):** Indica que el atributo es un valor autonumérico (PRIMARY KEY en MySQL, por ejemplo). Por lo tanto, Hibernate debe generar la clave primaria.
- **@Column(name="nombre_columna"):** Se utiliza para indicar el nombre de la columna en la tabla donde debe ser mapeado el atributo.

Las anotaciones se pueden hacer a nivel de atributos o a nivel de getters, en cualquier caso, para la entidad siempre se realizarán en el mismo nivel:

```
@Entity
@Table(name="actores", catalog="db_peliculas")
public class Actor {

  private Integer id;
  private String nombre;
  private LocalDate fechaNacimiento;

  // Constructor/es
  public Actor() { . . . }
```

```
    . . .

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name = "id")
    public Integer getId() {
        return this.id;
    }

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

    @Column(name = "nombre")
    public String getNombre() {
        return this.nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    @Column(name = "fecha_nacimiento")
    public LocalDate getFechaNacimiento() {
        return this.fechaNacimiento;
    }

    public void setFechaNacimiento(LocalDate fechaNacimiento) {
        this.fechaNacimiento = fechaNacimiento;
    }

    . . .

    // El resto de métodos se implementan como siempre
    @Override
    public String toString() {
        . . .
    }

    . . .
}
```

NOTA: *'java.util.Date'* ha sido reemplazado por las clases *'java.time'* en versiones más recientes de Java (a partir de Java 8). Por lo tanto, para representar solamente una fecha (sin hora), podemos utilizar la clase **LocalDate**. Si necesitamos trabajar con fechas y horas, podemos utilizar la clase **LocalDateTime**.

7. OPERACIONES SOBRE LA BASE DE DATOS

7.1. Guardar un objeto

Para registrar un nuevo objeto en la Base de Datos necesitamos haber creado previamente la clase y haberla mapeado correctamente con la tabla que le corresponda. Entonces, utilizando la clase **HibernateUtil** podremos obtener una sesión (conexión con la Base de Datos) para registrar ese objeto directamente en la Base de Datos de la siguiente forma.

```
. . .
UnaClase unObjeto = new UnaClase();
. . .
Session sesion = HibernateUtil.getSession();
sesion.beginTransaction();
sesion.persist(unObjeto);
sesion.getTransaction().commit();
sesion.close();
. . .
```

Hay que tener en cuenta que entre el inicio y cierre de la transacción podemos realizar más de una operación y éstas se ejecutarán como tal. Es la forma correcta en el caso de que queramos registrar más de un objeto cuando éstos estén relacionados de alguna forma y dependan entre ellos.

Un caso muy claro sería el del registro de un pedido junto con todas sus líneas de detalle puesto que no tendría sentido registrarlo sin los detalles, por lo que la forma más segura sería darlos de alta dentro de una misma transacción.

```
. . .
Session sesion = HibernateUtil.getSession();
sesion.beginTransaction();
sesion.persist(unPedido);
for(DetallePedido detallePedido : detallesDelPedido)
  sesion.persist(detallePedido);
sesion.getTransaction().commit();
sesion.close();
. . .
```

7.2. Modificar un objeto

En el caso de que queramos modificar un objeto, la operación se realiza de la misma forma que para el caso de registrar uno nuevo. Hibernate decide qué hacer (si registrar o modificar) comprobando si el objeto que se le envía tiene un valor válido para el campo id. Así, la única diferencia con el ejemplo anterior es que ahora dispondremos de un objeto que hemos obtenido previamente de la Base de Datos y al que hemos realizado algunas modificaciones (nunca el campo id).

```
. . .
Session sesion = HibernateUtil.getSession();
sesion.beginTransaction();
sesion.merge(unObjeto);
sesion.getTransaction().commit();
sesion.close();
. . .
```

7.3. Eliminar un objeto

Por último, en el caso que queramos eliminar un objeto:

```
. . .
Session sesion = HibernateUtil.getSession();
sesion.beginTransaction();
sesion.remove(unObjeto);
sesion.getTransaction().commit();
sesion.close();
. . .
```

Hibernate anteriores a la versión 6

Acción	Método
Guardar	sesion.save(unObjeto) o sesion.saveOrUpdate(unObjeto)
Actualizar	sesion.update(unObjeto) o sesion.saveOrUpdate(unObjeto)
Borrar	sesion.delete(unObjeto)

8. RELACIONES

Para el mapeo de las relaciones, además de crear el correspondiente objeto que permita mantener la relación entre las clases, éstos atributos deben ser mapeados según convenga. Para todos los casos, se indicará el tipo de relación visto desde el lado correspondiente:

- **@OneToOne** Indica que el objeto es parte de una relación 1-1.
- **@ManyToOne** Indica que el objeto es parte de una relación N-1. En este caso el atributo sería el lado 1.
- **@OneToMany** Indica que el objeto es parte de una relación 1-N. En este caso el atributo sería el lado N.
- **@ManyToMany** Indica que el objeto es parte de una relación N-M. En este caso se indica la tabla que mantiene la referencia entre las tablas y los campos que hacen el papel de claves ajenas en la base de datos.

En el otro lado de la relación indicaremos el tipo de relación acompañado de la anotación **@MappedBy** añadiendo el atributo de la otra clase donde se especifica toda la información sobre el mapeo.

8.1. Relación unidireccional o bidireccional

La elección entre una **relación unidireccional** y **bidireccional** en Hibernate depende de las necesidades y requisitos específicos de tu aplicación. Ambas opciones tienen ventajas y desventajas, y la elección correcta dependerá del contexto.

Relación Bidireccional:

- **Navegación eficiente en ambas direcciones:** Una relación bidireccional te permite navegar fácilmente desde la entidad "uno" a la entidad "muchos" y viceversa. Esto puede ser útil cuando necesitas acceder a ambas partes de la relación con frecuencia.
- **Optimización de consultas:** En situaciones donde deseas optimizar las consultas de base de datos y evitar consultas adicionales, una relación bidireccional puede ser útil. Puedes realizar consultas que recuperan objetos relacionados de manera más eficiente.

- **Mantenimiento consistente:** Una relación bidireccional garantiza que las dos partes de la relación se mantengan consistentes automáticamente. Si actualizas una parte de la relación, la otra parte se actualiza automáticamente, lo que evita errores y problemas de integridad.

Relación Unidireccional:

- **Sencillez y menos complejidad:** En algunos casos, una relación unidireccional puede ser más simple y fácil de manejar. Si no necesitas acceso a ambas partes de la relación con regularidad, una relación unidireccional puede simplificar el diseño.
- **Menos sobrecarga de memoria:** Las relaciones bidireccionales a veces pueden llevar a problemas de sobrecarga de memoria debido a las referencias circulares entre objetos. Una relación unidireccional puede evitar esto y ser más eficiente en términos de uso de memoria.
- **Menor riesgo de ciclos:** Las relaciones bidireccionales pueden introducir ciclos en el gráfico de objetos, lo que puede complicar la serialización y la gestión de objetos en memoria.

En resumen, debes considerar las necesidades específicas de tu aplicación y el rendimiento al decidir entre una relación bidireccional o unidireccional. No hay una respuesta única para todos los casos, y la elección dependerá de factores como la complejidad de tu dominio, la eficiencia de las consultas, la simplicidad del diseño y la gestión de la memoria. Es importante comprender las implicaciones de ambas opciones y elegir la que mejor se adapte a tu situación particular.

NOTA: Hay que mencionar que las relaciones (1-1, N-1 o 1-N) que veremos en los siguientes apartados, vamos a realizar las relaciones bidireccionales y no siempre es la mejor opción. Solamente en los casos de las relaciones Maestro-Detalle, donde una entidad depende de la otra y, por lo tanto, las entidades no se pueden excluir, es un claro ejemplo de relación bidireccional.

8.2. Relaciones 1 a 1 (OneToOne)


```
@Entity
@Table(name = "personajes")
public class Personaje {
  . . .
  @OneToOne(cascade = CascadeType.ALL)
  @JoinColumn(name = "id_arma")
  private Arma arma;
  . . .
}
```

- **@OneToOne(cascade = CascadeType.ALL):** Esta anotación indica la relación uno a uno de las 2 tablas. También indicamos el valor de cascada, esto significa que las operaciones de persistencia se propagarán desde Personaje a Arma, es decir, cuando guardemos un personaje se guardará también el arma asociada sin tenerlo que hacer explícitamente. Sin embargo, esta cascada no funciona en la dirección inversa porque no está definida.
- **JoinColumn:** Indicaremos el nombre de la columna que se usará como clave ajena.

```
@Entity
@Table(name = "armas")
public class Arma {
  . . .
  @OneToOne(mappedBy = "arma")
  private Personaje personaje;
  . . .
}
```

- **@OneToOne(mappedBy = “”):** Este atributo contendrá el nombre de la propiedad Java de la clase hija que enlaza con la clase padre.
    - Si queremos una **relación unidireccional**, no haríamos este mapeo, en la clase Arma haciendo referencia a Personaje.

En el repositorio proyecto: adt5_ejemplo2

8.3. Relaciones n a 1 (ManyToOne) / Relaciones 1 a n (OneToMany)


```
@Entity
@Table(name = "personajes")
public class Personaje {
  . . .

  @ManyToOne(cascade = CascadeType.ALL)
  @JoinColumn(name = "id_arma")
  private Arma arma;
  . . .
}
```

- **ManyToOne:** Indicamos que desde este lado es una relación muchos a uno.
- **JoinColumn:** Indicaremos el nombre de la columna que se usará como clave ajena. En nuestro ejemplo, la columna id_arma de la tabla personajes será una clave ajena que se relaciona con la clave primaria de la clase Arma.

```
@Entity
@Table(name = "armas")
public class Arma {
  . . .

  @OneToMany(mappedBy = "arma", cascade = CascadeType.ALL)
  private List<Personaje> personajes;
  . . .
}
```

- **OneToMany:** Esta propiedad contendrá la lista de hijos.
  - **mappedBy:** Este atributo contendrá el nombre de la propiedad Java de la clase hija que enlaza con la clase padre. En nuestro ejemplo es el nombre de la propiedad arma que se encuentra en la clase Personaje.
  - Si queremos una **relación unidireccional**, no haríamos este mapeo, en la clase Arma.

En el repositorio proyecto: adt5_ejemplo3

8.4. Relaciones n a m (ManyToMany)


```
@Entity
@Table(name = "personajes")
public class Personaje {

  . . .

  @ManyToMany(cascade = CascadeType.ALL)
  @JoinTable(name="personaje_arma",
       joinColumns={@JoinColumn(name="id_personaje")},
       inverseJoinColumns={@JoinColumn(name="id_arma")})
  private List<Arma> armas = new ArrayList<>();

  . . .

}
```

- **ManyToMany:** Como su nombre indica le dice a Hibernate que la propiedad contendrá una lista de objetos que participa en una relación muchos a muchos.
    - **cascade:** CascadeType.ALL. Para hacerlo en cascada.
- **JoinTable:** Esta anotación contiene la información sobre la tabla que realiza la relación muchos a muchos
    - **name:** Nombre de la tabla (personaje_arma) que realiza la relación muchos a muchos.
    - **joinColumns:** Contiene cada una de las columnas que forman la clave primaria de esta clase que estamos definiendo. Cada columna se indica mediante una anotación @JoinColumn y en el atributo name contiene el nombre de la columna.
    - **inverseJoinColumns:** Contiene cada una de las columnas que forman la clave primaria de la clase clase con la que tenemos la relación. Cada columna se indica mediante una anotación @JoinColumn y en el atributo name contiene el nombre de la columna.

```java
@Entity
@Table(name = "armas")
public class Arma {

  . . .

  @ManyToMany(mappedBy = "armas", cascade = CascadeType.ALL)
  private List<Personaje> personajes = new ArrayList<>();

  . . .
}
```

- **ManyToMany:** Indica que la propiedad contiene una lista de objetos que participan en una relación muchos a muchos.
    - **mappedBy:** Contiene el nombre de la propiedad Java de la otra clase desde la cual se relaciona con ésta. En nuestro ejemplo es la propiedad armas.
    - **cascade:** Este atributo tiene el mismo significado que el del fichero de mapeo de Hibernate.
    - Si queremos una **relación unidireccional**, no haríamos este mapeo, en la clase Arma.

NOTA: Al poner el atributo mappedBy ya no es necesario incluir la anotación @JoinTable ya que dicha información ya se indica en el otro lado de la relación.

En el repositorio proyecto: adt5_ejemplo4

📖 **Práctica Recomendada:**

Por lo general, se recomienda manejar la persistencia de entidades desde la "parte propietaria" de la relación, que en los ejemplos vistos (adt5_ejemplo2, adt5_ejemplo3 y adt5_ejemplo4) siempre ha sido la clase Personaje. Esto garantiza que la integridad de datos se mantenga y que las operaciones de cascada funcionen como se espera.

8.5. Operaciones en cascade y su significado

Hibernate y JPA ofrecen varias opciones de cascada que se pueden utilizar para definir cómo las operaciones realizadas en una entidad se propagan a las entidades relacionadas. Estas opciones de cascada son especialmente útiles en relaciones entre entidades.

Aquí están las opciones de cascada disponibles y su significado:

- **CascadeType.ALL:** aplica todas las operaciones de cascada. Esto significa que todas las operaciones de persistencia, eliminación, actualización… se propagarán de la entidad padre a las entidades relacionadas.
- **CascadeType.PERSIST:** propaga la operación de persistencia. Si una entidad se persiste, todas sus entidades relacionadas también se persistirán.
- **CascadeType.MERGE:** propaga la operación de actualización (merge).
- **CascadeType.REMOVE:** propaga la operación de eliminación. Si se elimina una entidad, todas las entidades relacionadas también se eliminarán. Esta opción debe usarse con cuidado para evitar la eliminación no deseada de datos.
- **CascadeType.DETACH:** propaga la operación de desacoplamiento. Si una entidad se desacopla del contexto de persistencia, todas las entidades relacionadas también se desacoplan.
- **CascadeType.REFRESH:** propaga la operación de sincronización. Si se actualiza el estado de una entidad para reflejar los valores actuales en la base de datos, todas las entidades relacionadas también se sincronizarán.
- **CascadeType.LOCK:** propaga la operación de bloqueo. Si se aplica un bloqueo a la entidad, el bloqueo se extiende también a las entidades relacionadas.

8.5.1. Ejemplo práctico: adt5_ejemplo5

Para entender la diferencia entre **CascadeType.MERGE** y **CascadeType.REFRESH** en JPA y Hibernate, voy a proporcionar un ejemplo con dos entidades, Persona y Dirección. La entidad **Persona** tiene una relación con **Dirección**. Vamos a ver cómo estas opciones de cascada afectan el comportamiento cuando se realizan operaciones con estas entidades.

```java
@Entity
public class Persona {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    @Column
    private String nombre;

    @OneToOne(cascade = {CascadeType.MERGE, CascadeType.REFRESH})
    @JoinColumn(name = "id_direccion")
    private Direccion direccion;

    // Constructores, getters y setters
}
```

```java
@Entity
public class Direccion {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    @Column
    private String calle;

    // Constructores, getters y setters
}
```

Diferencia entre CascadeType.MERGE y CascadeType.REFRESH

CascadeType.MERGE: es utilizada durante la operación actualización (merge). Si tienes una instancia desacoplada de Persona y la modificas (por ejemplo, cambiando el nombre o la dirección asociada), al llamar a merge() en el EntityManager, no solo se actualizará Persona, sino también su Dirección asociada en la base de datos.

```java
// Supongamos que persona es una entidad desacoplada con cambios
persona.setNombre("Nombre Actualizado");
persona.getDireccion().setCalle("Calle Actualizada");

session.merge(persona);
```

En este caso, tanto los cambios en Persona como en Dirección se actualizarán con la base de datos.

CascadeType.REFRESH: se utiliza con la operación de sincronización (refresh). Si, por ejemplo, después de realizar ciertas operaciones, quieres asegurarte de que tu instancia de Persona y su Dirección asociada reflejen el estado actual de la base de datos, usas refresh().

```
// Refrescar la instancia persona para sincronizarla con la base de datos
session.refresh(persona);
```

Esto hará que cualquier cambio realizado en la instancia persona y su dirección asociada en la memoria se reemplace con los valores actuales de la base de datos.

Ejemplo de Uso

Imagina un escenario donde recuperas una **Persona** de la base de datos, luego la desacoplas (por ejemplo, pasándola a una capa de vista donde se modifican sus propiedades), y finalmente deseas actualizar esos cambios con la base de datos. Utilizarías **merge()** en este caso. Por otro lado, si después de ciertas operaciones en la base de datos quieres asegurarte de que tus entidades en memoria reflejen el estado actual de la base de datos, usarías **refresh()**.

Uso Práctico de las Opciones de Cascada

- **Selección Cuidadosa:** es importante seleccionar las opciones de cascada que se ajustan a las necesidades de tu aplicación. No todas las situaciones requieren todas las formas de cascada, y un uso inadecuado puede llevar a resultados no deseados, como la eliminación accidental de datos.
- **Ejemplo con PERSIST y MERGE:** en relaciones como 'Uno a Muchos' o 'Muchos a Muchos', las opciones PERSIST y MERGE suelen ser suficientes para la mayoría de los casos de uso, asegurando que las entidades relacionadas se guardan y actualizan adecuadamente.
- **Consideraciones de Rendimiento:** el uso excesivo de cascadas puede afectar el rendimiento, especialmente en relaciones con un gran número de entidades relacionadas.

Elegir las opciones correctas de cascada es clave para garantizar el correcto funcionamiento de tu aplicación y la integridad de la base de datos.

8.6. OrphanRemoval

La propiedad **orphanRemoval** se usa en relaciones 'Uno a Muchos' (@OneToMany) o 'Uno a Uno' (@OneToOne) en Hibernate. Esta propiedad determina el comportamiento de Hibernate cuando una entidad hija es eliminada de la colección en la entidad padre.

- **orphanRemoval = true:** cuando orphanRemoval está configurado como true, significa que si una entidad hija es eliminada de la colección en la entidad padre, Hibernate eliminará automáticamente esa entidad hija de la base de datos. Esto es útil para garantizar que no queden datos huérfanos (orphan) en la base de datos, es decir, datos que ya no tienen una relación significativa.
- **orphanRemoval = false (valor por defecto):** eliminar una entidad hija de una colección en la entidad padre no provocará su eliminación de la base de datos. La entidad hija seguirá existiendo en la base de datos, pero ya no estará asociada con la entidad padre.

Ejemplo

Vamos a crear un ejemplo simple usando una relación @OneToMany entre **Curso** y **Estudiante**, donde un curso puede tener varios estudiantes.

```
@Entity
public class Curso {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    @Column
    private String nombre;

    @OneToMany(mappedBy = "curso", cascade = CascadeType.ALL, orphanRemoval = true)
    private List<Estudiante> estudiantes = new ArrayList<>();

    // Constructores, getters y setters

}
```

```java
@Entity
public class Estudiante {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    @Column
    private String nombre;

    @ManyToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "id_curso")
    private Curso curso;

    // Constructores, getters y setters
}
```

```java
// Recuperar un curso de la base de datos con un id = 2
Curso curso = session.get(Curso.class, 1);

// Recuperar un estudiante del curso anterior
Estudiante estudiante = curso.getEstudiantes().get(0);

// Eliminar del listado del curso el estudiante anterior y viceversa
curso.getEstudiantes().remove(estudiante);
estudiante.setCurso(null);

// Guardar en la base de datos
session.merge(curso);
```

- **Ejemplo con orphanRemoval = true:** si tienes un objeto Curso con estudiantes y eliminas un Estudiante de la lista, al guardar el objeto Curso, Hibernate también eliminará el Estudiante de la base de datos.
- **Ejemplo con orphanRemoval = false:** si realizas la misma operación, el Estudiante no será eliminado de la base de datos, solo será desvinculado del Curso.

Conclusión

Usar orphanRemoval = true es una forma efectiva de asegurarse de que los datos huérfanos no permanezcan en la base de datos y puede simplificar el manejo de las entidades en ciertos casos. Sin embargo, debe usarse con cuidado, ya que puede llevar a eliminaciones no deseadas si no se gestiona correctamente la relación entre entidades.

9. BÚSQUEDAS (LENGUAJE HQL)

Para el caso de las búsquedas se utiliza el **lenguaje HQL** (**Hibernate Query Language**). HQL es un lenguaje de consulta de objetos que se utiliza en Hibernate para recuperar y manipular datos de una base de datos. Es similar a SQL, pero en lugar de trabajar con tablas y columnas, trabaja con clases y atributos de Java.

Aquí hay algunos conceptos básicos de HQL que deberías conocer:

- La sintaxis básica de una consulta HQL es "SELECT ... FROM ... WHERE ...". La cláusula SELECT especifica qué propiedades o valores quieres recuperar, la cláusula FROM especifica de qué clases o entidades quieres recuperar los datos y la cláusula WHERE especifica cuáles objetos o filas deben incluirse en la consulta.
- Puedes utilizar las cláusulas FROM y WHERE para filtrar los resultados de la consulta. Por ejemplo, si quieres recuperar solo los objetos de una clase con una propiedad determinada, puedes utilizar una cláusula WHERE para especificar esto.
- Puedes utilizar parámetros en tus consultas HQL para hacerlas más flexibles. Por ejemplo, si quieres recuperar solo los objetos de una clase con una propiedad específica, pero no sabes el valor de esa propiedad de antemano, puedes utilizar un parámetro en tu consulta y establecer el valor del parámetro en tiempo de ejecución.
- HQL también admite operadores lógicos como AND, OR y NOT, así como funciones de agregación como SUM, AVG y COUNT. Esto te permite realizar consultas más complejas y recuperar informes agregados de tus datos.

Ejemplos

Aquí tienes algunos ejemplos de consultas HQL que pueden ser útiles:

- Obtener un **objeto identificado por el id:**

```
Session session = HibernateUtil.getSession();
. . .
int id = 10;
Cliente cliente = session.get(Cliente.class, id);
. . .
```

- Obtener **todos los objetos** de una clase:

```
. . .
Query<Cliente> query = session.createQuery("FROM Cliente", Cliente.class);

List<Cliente> listaClientes = query.list();
. . .
```

- Obtener objetos de una clase **añadiendo algún criterio de búsqueda:**
    - Si el criterio especificado nos devuelve un solo objeto:

```
. . .
String nombre = "Pepe";
. . .
Query<Cliente> query = session.createQuery("FROM Cliente c WHERE c.nombre = :nombre",
Cliente.class);
query.setParameter("nombre", nombre);

Cliente cliente = query.uniqueResult();
. . .
```

    - Si el criterio especificado nos puede devolver más de un objeto:

```
. . .
String ciudad = "Valencia";
. . .
Query<Cliente> query = session.createQuery("FROM Cliente c WHERE c.ciudad = :ciudad",
Cliente.class);
query.setParameter("ciudad", ciudad);

List<Cliente> listaClientes = query.list();
. . .
```

- Obtener objetos de una clase utilizando las **relaciones entre clases:**

```
. . .
Query<DetallePedido> query = session.
  createQuery("FROM DetallePedido dp WHERE dp.pedido.numeroPedido = :numeroPedido",
DetallePedido.class);
query.setParameter("numeroPedido", numeroPedido);

List<DetallePedido> listaDetalles = query.list();
. . .
```

- También es posible lanzar **consultas directamente en lenguaje SQL**, trabajando entonces directamente con las tablas y campos de la base de datos

```
. . .
Query<Object[]> sqlQuery = session.
    createNativeQuery("SELECT nombre, apellidos FROM clientes WHERE ciudad = :ciudad",
Object[].class);
query.setParameter("ciudad", ciudad);

List<Object[]> listado = query.list();

for (Object[] resultado : listado) {
  String nombre = resultado[0];
  String apellidos = resultado[1];
  . . .
}
. . .
```

10. HIBERNATE ENVERS: AUDITORÍA

Hibernate Envers es una extensión de Hibernate, proporciona la capacidad de llevar un registro de cambios en los datos de una aplicación Java a través de la auditoría y la revisión de datos.

Con Hibernate Envers, puede agregar fácilmente la auditoría de datos a su aplicación Java mediante el uso de anotaciones de Java y configuración de archivos de propiedades. Hibernate Envers almacena información sobre las revisiones de datos en **tablas separadas** en la base de datos, lo que le permite consultar y recuperar información sobre cambios específicos en los datos en diferentes momentos en el tiempo.

Hibernate Envers es útil en muchas situaciones donde es necesario llevar un registro de cambios en los datos, como en aplicaciones de gestión de contenidos, sistemas de seguimiento de cambios y aplicaciones de gestión de proyectos. También puede ser útil en situaciones donde es necesario cumplir con requisitos de cumplimiento o regulación que exigen la auditoría.

10.1. Funcionamiento

Hibernate Envers funciona mediante la adición de anotaciones de Java a las entidades de su aplicación para indicar qué campos deben ser auditados. También puede configurar Hibernate Envers mediante un archivo de propiedades para especificar cómo se deben llevar a cabo las revisiones.

Cuando se realiza un cambio en una entidad marcada para ser auditar, Hibernate Envers crea una revisión de los datos en una tabla separada en la base de datos. Esta revisión contiene información sobre el estado de los datos en ese momento, como el valor de los campos de la entidad y la fecha y hora en que se realizó el cambio.

Para consultar y recuperar información sobre revisiones específicas, Hibernate Envers proporciona una API de consulta que permite realizar consultas en las tablas de revisión. Por ejemplo, puede realizar una consulta para recuperar el estado de los datos en un momento específico o para encontrar todos los cambios realizados en un rango de tiempo determinado.

En resumen, Hibernate Envers le permite llevar un registro de cambios en los datos de su aplicación de manera fácil y sencilla, lo que puede ser útil en muchas situaciones donde es necesario auditar o revisar los datos.

Ejemplo

1. Primero, debe agregar la dependencia de Hibernate Envers. Tener en cuenta que la versión de hibernate-core debe ser compatible con hibernate-envers (versión 6.1.7.Final).

```
<dependency>
     <groupId>org.hibernate</groupId>
     <artifactId>hibernate-core</artifactId>
     <version>6.1.7.Final</version>
</dependency>
<dependency>
     <groupId>org.hibernate.orm</groupId>
     <artifactId>hibernate-envers</artifactId>
     <version>6.1.7.Final</version>
</dependency>
```

2. Luego, debe marcar las entidades que desea auditar con la anotación @Audited.

```
import org.hibernate.envers.Audited;

@Entity
@Audited
public class Empleado {
    // Atributos y métodos de la entidad
}
```

3. Usamos la API de Hibernate para guardar y actualizar entidades de la siguiente manera:

```
// Crea una nueva entidad
Empleado empl = new Empleado();
empl.setNombre("John Smith");

// Da de alta el objeto en la tabla de datos
Session session = HibernateUtil.getSession();
session.beginTransaction();
session.persist(empl);
session.getTransaction().commit();
```

```
// Actualiza la entidad
empl.setNombre("John Doe");
session.beginTransaction();
session.merge(empl);
session.getTransaction().commit();
```

Cuando se realizan cambios en la entidad, Hibernate Envers crea revisiones en una tabla separada en la base de datos. Por ejemplo, después de ejecutar el código anterior, la tabla de revisiones podría tener dos entradas, una para el estado de la entidad cuando se guardó por primera vez y otra para el estado después de la actualización.

4. Para consultar y recuperar información sobre revisiones específicas, puede usar la API de consulta de Hibernate Envers. Por ejemplo:

```
// Recupera todas las revisiones de la entidad Empleado
AuditReader reader = AuditReaderFactory.get(session);
List<Number> revisions = reader.getRevisions(Empleado.class, employeeId);

// Recupera el estado de la entidad en una revisión específica
Empleado empleadoRevision = reader.find(Empleado.class, employeeId, revisions);
```

10.2. Tablas para la auditoría

Hibernate Envers genera una serie de tablas de auditoría para rastrear los cambios en las entidades de tu aplicación. Estas tablas son fundamentales para mantener un historial de las revisiones realizadas en los datos. Aquí te explico qué tablas se generan y qué significa cada una:

1. **Tabla de Auditoría para Cada Entidad Auditada:** Por cada entidad auditada en tu aplicación, Hibernate Envers crea una tabla de auditoría correspondiente. El nombre de la tabla se forma generalmente añadiendo un sufijo (como _AUD) al nombre de la tabla original de la entidad. Esta tabla contiene las mismas columnas que la tabla original de la entidad, más algunas columnas adicionales para la auditoría:
    - **REVTYPE:** Indica el tipo de operación realizada. Los valores posibles son 0 (INSERT), 1 (UPDATE), y 2 (DELETE).
    - **REV:** Es una clave foránea que apunta a la tabla de revisiones, indicando la revisión específica en la que se realizó el cambio.

2. **Tabla de Revisiones (por defecto REVINFO):** Esta tabla almacena los metadatos sobre cada revisión. Normalmente, contiene las siguientes columnas:
    - **REV:** Es el identificador único de la revisión. Suele ser un número secuencial.
    - **REVTSTMP:** Almacena la fecha y hora en la que se realizó la revisión, normalmente en formato de timestamp.

```
// En MySQL
SELECT DATE_FORMAT(FROM_UNIXTIME(REVTSTMP / 1000), '%Y-%m-%d %H:%i:%s')
FROM revinfo;
```

3. **Tablas de Asociaciones de Auditoría (si las hay):** Si tienes entidades con relaciones (como OneToMany, ManyToOne, etc.), Envers puede generar tablas adicionales para auditar estas relaciones. Estas tablas permiten rastrear cómo las asociaciones entre entidades cambian con el tiempo.
4. **Tablas Personalizadas (si se usan):** Puedes personalizar o extender las tablas de auditoría, por ejemplo, para añadir columnas adicionales que almacenen más metadatos sobre las revisiones. Esto se hace mediante la API de Envers o la configuración XML.

Es importante recordar que, aunque Hibernate Envers facilita enormemente la auditoría, el diseño y la gestión de las tablas de auditoría deben alinearse con los requisitos específicos de tu aplicación y las políticas de gestión de datos.

10.2.1. En el repositorio proyecto: adt5_ejemplo6

Puede acceder a estas tablas directamente desde su SGBD y consultar la información de auditoría almacenada en ellas.

empleado

id	nombre
1	Mario Bros
2	Luigi
3	Princesa Daisy

empleado_aud

id	REV	REVTYPE	nombre
1	1	0	Mario Bros
2	1	0	Luigi
3	1	0	Bowser
3	2	1	Princesa Daisy

revinfo

REV	REVTSTMP
1	1.699.701.563.807
2	1.699.701.563.842

11. BIBLIOGRAFIA

Curso Online de Hibernate y JPA de OpenWebinars:

- https://openwebinars.net/academia/aprende/hibernate/

Apuntes de Acceso a Datos de Santiago Faci:

- https://datos.codeandcoke.com/

Curso de hibernate:

- http://www.cursohibernate.es/doku.php

Videotutoriales de JPA + Hibernate de Makigas: tutoriales de programación:

- https://www.youtube.com/playlist?list=PLTd5ehIj0goPcnQs34i0F-Kgp5JHX8UUv