Apuntes básico HIBERNATE

Contenido

Primero: conceptos claros

¿Qué es ORM?

Significa Objet Relational Mapping (Mapeo Objeto-Relacional). Es una técnica de programación que consiste en mapear de forma directa los datos de una BBDD relacional (MySQL, Oracle, etc.) con una estructura lógica de objetos entidades.

Hasta ahora, con JDBC, hemos tenido que hacer este mapeo de manera manual. Cuando insertábamos un objeto en base de datos teníamos que ir indicando qué atributo correspondía con cada columna de la bbdd. Igual para consultarlo, actualizarlo, etc.

El objetivo de ORM es que esto sea automático, simplificando la programación.

¿Qué es JPA?

Significa Java Persistence API (API de persistencia para JAVA). Es una especificación de un API para implementar la persistencia en JAVA basada en ORM.

El que sea una especificación significa que no proporciona ninguna clase ni librería para trabajar, simplemente es una descripción de cómo se debería trabajar (como si fuera una interfaz). Otros tendrán que implementar esta especificación.

¿Qué es Hibernate?

Es una implementación (en software libre) de la especificación JPA.

Algunas otras implementaciones son EclipseLink (desarrollado por Eclipse), OpenJPA (desarrollado por Apache), etc.

Ventajas e inconvenientes

Los defensores y detractores de usar JPA vs JDBC son muchos. La realidad es que depende del tipo de proyecto.

- Si la BBDD ya existe y su diseño es complejo o no es muy bueno, tendrás dificultades para hacer un mapeo a entidades JAVA. Igualmente, si tienes tablas con PK compuestas. Quizás sea más fácil en estos casos que uses JDBC. Si la BBDD la vas a crear nueva y te puedes permitir un buen diseño desde cero, o si la que ya existe está bien diseñada, no encontrarás problemas para trabajar con JPA.

- Si la mayoría de operaciones del proyecto consistirán en hacer querys complejas (consulta de datos), JPA no te va a ahorrar mucho trabajo, porque tendrás que seguir escribiendo consultas complejas. Sin embargo, si en tu proyecto vas a tener que hacer casi siempre CRUDs de todas las tablas, JPA te ahorrará mucho trabajo.

- Si necesitas que las consultas a BBDD estén 100% optimizadas para que sean lo más rápidas posible y es un aspecto crítico del proyecto, JPA no es buena opción porque suele ser menos eficiente.

- Si quieres implementar algún tipo de caché fácilmente de las consultas que se hacen en BBDD, Hibernate te proporcionará una forma sencilla de hacerlo.

- Si quieres que tu aplicación funcione con cualquier BBDD, con JPA no tendrás ningún problema. Te bastará con cambiar la configuración de conexión. Sin embargo, con JDBC, tendrás que asegurarte de no escribir nada propio de un proveedor de BBDD concreto. Es decir, tendrás que escribir siempre SQL estándar.

- A veces, muchos programadores, deciden utilizar Hibernate en su proyecto sin conocerlo bien. Esto puede provocar a la larga consecuencias desastrosas. Es importante conocer bien cómo funciona Hibernate para utilizarlo y así asegurarnos que todo funciona correctamente.

Documentación en la red

Hibernate es ampliamente utilizado por todo el mundo. Es, por tanto, muy sencillo encontrar documentación en la red para cualquier duda y para ampliar conocimientos.

En Baeldung existe un tutorial muy completo: https://www.baeldung.com/learn-jpa-hibernate

Documentación oficial de Hibernate: https://hibernate.org/orm/documentation/5.6/

Por dónde empezamos…

Lo primero es preparar el proyecto para comenzar a trabajar.

1. En nuestro proyecto JAVA tendremos que añadir la dependencia MAVEN para trabajar con Hibernate. También tendremos que añadir la dependencia MAVEN del driver JDBC de nuestra BBDD. (Hibernate, internamente, utiliza JDBC)

```
<dependency>
    <groupId>org.hibernate.orm</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>6.4.1.Final</version>
</dependency>
```

2. En la carpeta de recursos del proyecto, añadimos el fichero de configuración de Hibernate. Este fichero se suele nombrar como hibernate.cfg.xml.

```
<?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="hibernate.connection.driver_class">org.mariadb.jdbc.Driver</property>
     <property name="hibernate.connection.url">jdbc:mariadb://localhost:3306/database</property>
     <property name="hibernate.connection.username">usuario</property>
     <property name="hibernate.connection.password">password</property>
     <property name="hibernate.show_sql">true</property>
     <property name="hibernate.connection.autoReconnect">true</property>
     <!-- Para trabajar con getCurrentSession en lugar de openSession -->
     <!--  <property name="current_session_context_class">thread</property>  -->

     <!--  Podemos declarar aquí nuestras entidades o añadirlas al crear el metadata  -->
     <!--      <mapping class="mipaquete.MiClase" />   -->
 </session-factory>
</hibernate-configuration>
```

- ShowSQL nos indica si queremos mostrar en el log el SQL que Hibernate genera
- AutoReconnect es para indicar si queremos que la conexión a la BBDD se regenere automáticamente si se corta por algún motivo.
- CurrentSessionContext nos permite configurar cómo trabajar con currentSession. Este es un modo de trabajar en el que no se abre una sesión nueva cada vez, sino que se reutiliza la misma continuamente.
- En los mapping podemos indicar las entidades que vamos a mapear con la BBDD. Podemos hacerlo ahí o en una clase como veremos luego.

3. Creamos la clase HibernateUtil que se encargará de inicializar el sessionFactory con el que trabajaremos en cada transacción. Esta clase nos la podemos crear donde queramos. Tendrá el siguiente contenido:

```
public class HibernateUtil {

	private static SessionFactory sessionFactoy;

	public static SessionFactory getSessionFactoy() {
		if (sessionFactoy == null) {
			init();
		}
		return sessionFactoy;
	}

	private static void init() {
		try {
			ServiceRegistry registry = new StandardServiceRegistryBuilder()
				.configure("hibernate.cfg.xml").build();

			Metadata metadata = new MetadataSources(registry)
					// Aquí añadimos las entidades que queremos mapear
					.addAnnotatedClass(MiClase.class)
					.getMetadataBuilder().build();

			sessionFactoy = metadata.getSessionFactoryBuilder().build();
		} catch (Exception e) {
			throw new ExceptionInInitializerError(e);
		}
	}
}
```

Mapeo de entidades

Lo siguiente será mapear las entidades con las tablas de BBDD. Tendremos que crear nuestro modelo: el conjunto de clases con las que trabajaremos y que tendrán su correspondencia con las tablas de base de datos. Veamos cómo se hace:

Tabla independiente sin relaciones

Tenemos una tabla Animal en BBDD con un campo id y otro campo descripción, siendo el primero un PK autogenerada.

La clase Animal de nuestro modelo sería la siguiente:

```
@Entity
public class Animal {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String descripcion;

```

Las anotaciones que hemos indicado son las siguientes:

- Entity → Lo pondremos siempre para indicar que es una entidad que hay que mapear
- Id → Indica cuál es la PK
- GeneratedValue → Indica que la PK es generada de forma automática. La estrategia que está en el ejemplo corresponde con el AUTO_INCREMENT de MySQL

Cuando cambian los nombres

En el ejemplo anterior, se asume que el nombre de la tabla y de las columnas coincide con los de la clase. Si no es así, podemos modificarlos con las anotaciones Table y Column:

```
@Entity
@Table(name = "animales")
public class Animal {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id_animal")
    private Long id;

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

```

Relación 1 a 1 entre dos entidades (unidireccional)

En nuestro modelo, tendremos una clase Propietario que será dueño de un Animal. Es una relación 1 a 1 unidireccional (el propietario tendrá una referencia al animal, pero el animal no tendrá una referencia a su propietario).

En BBDD, en la tabla Propietario, tenemos una columna id_animal que está configurada como FK de la tabla Animal.

Usamos las anotaciones:

- OneToOne → Para indicar el tipo de relación. También podremos indicar aquí el tipo de CASCADE (ver más adelante)
- JoinColumn → Para indicar el nombre de la columna con la que debemos hacer el JOIN

```
16 @Entity
17 public class Propietario {
18
19     @Id
20     @GeneratedValue(strategy = GenerationType.IDENTITY)
21     private Long id;
22
23     @OneToOne
24     @JoinColumn(name = "id_animal")
25     private Animal animal;
26
```

Relación 1 a 1 entre dos entidades (bidireccional)

Este caso es igual al anterior. La BBDD es idéntica. Lo único que queremos cambiar en nuestro modelo es que la clase Animal también tenga una referencia a su Propietario (no solo desde el propietario al animal, sino también al revés). Es una relación 1 a 1 bidireccional.

La clase Propietario no tenemos que modificarla.

En la clase Animal, tenemos que:

- Añadimos el atributo Propietario
- Indicamos de nuevo la anotación OneToOne
- Como la FK no está en esta tabla y el mapeo ya se ha hecho en la otra clase, especificamos el atributo mappedBy indicando que el mapeo está hecho en el atributo "animal" de la clase Propietario.

```
10 @Entity
11 public class Animal {
12
13     @Id
14     @GeneratedValue(strategy = GenerationType.IDENTITY)
15     private Long id;
16
17     private String descripcion;
18
19     @OneToOne(mappedBy = "animal")
20     private Propietario propietario;
21
```

Relación 1 a muchos entre dos entidades (unidireccional)

Tenemos dos tablas con sus correspondientes clases:

- Persona (una persona puede tener muchas direcciones)
- Dirección (una dirección sólo es de una persona)

En el modelo de clases, queremos tener en la clase Persona un atributo que sea la lista de sus direcciones. No necesitamos en la clase Direccion un atributo con la Persona a la que corresponde. Es, por tanto, una relación unidireccional.

Usamos las anotaciones:

- OneToMany → Para indicar el tipo de relación. También podremos indicar aquí el tipo de CASCADE (ver más adelante)
- JoinColumn → Para indicar el nombre de la columna con la que debemos hacer el JOIN. En este caso la columna estará en la tabla direcciones, no importa.

```
16 @Entity
17 @Table(name = "personas")
18 public class Persona {
19
20     @Id
21     @GeneratedValue(strategy = GenerationType.IDENTITY)
22     @Column(name = "id_persona")
23     private Long id;
24
25     @OneToMany
26     @JoinColumn(name = "id_persona")
27     private List<Direccion> direcciones;
28
```

NOTA IMPORTANTE:

- Es posible que, si enlazo varios OneToMany, hibernate haga un JOIN que produzca un producto cartesiano con registros duplicados al consultar. Para solucionarlo, podemos declarar nuestro atributo como un Set en lugar de como un List.

Relación 1 a muchos entre dos entidades (bidireccional)

Tenemos las mismas tablas y clases que en el caso anterior. Pero ahora, además de en la clase Persona tener un atributo que sea la lista de sus direcciones, necesitamos en la clase Direccion un atributo con la Persona a la que corresponde. Es, por tanto, una relación bidireccional.

Para esto, modificaremos la clase Persona para quitar el JoinColumn y poner un mappedBy. Esto indica que el mapeo se va a definir en el atributo "persona" de la clase Direccion, que es donde está la FK.

```
16 @Entity
17 @Table(name = "personas")
18 public class Persona {
19
20     @Id
21     @GeneratedValue(strategy = GenerationType.IDENTITY)
22     @Column(name = "id_persona")
23     private Long id;
24
25     @OneToMany(mappedBy = "persona")
26     private List<Direccion> direcciones;
27
```

En la clase Direccion, añadiremos el nuevo atributo Persona y usaremos las anotaciones:

- ManyToOne → Para indicar que es una relación de muchos a 1.
- JoinColumn → Para indicar la columna por la que demos hacer el JOIN

```
10 @Entity
11 public class Direccion {
12
13     @Id
14     @GeneratedValue(strategy = GenerationType.IDENTITY)
15     private Long id;
16
17     @ManyToOne
18     @JoinColumn(name = "id_persona")
19     private Persona persona;
20
```

Relación muchos a muchos entre dos entidades (unidireccional)

Tenemos tres tablas:

- Persona (una persona puede tener muchas direcciones)
- Dirección (una dirección puede pertenecer a muchas personas)
- Persona_Direccion (es una tabla que nos sirve para relacionar las dos anteriores)

En el modelo de clases, queremos tener en la clase Persona un atributo que sea la lista de sus direcciones. No necesitamos en la clase Direccion un atributo con la lista de personas a las que pertenece. Es, por tanto, una relación unidireccional.

Usamos las anotaciones:

- ManyToMany → Para indicar el tipo de relación. También podremos indicar aquí el tipo de CASCADE (ver más adelante)
- JoinTable → Para indicar el nombre de la tabla y columnas a través de las que se debe relacionar.

```
18 @Entity
19 @Table(name = "personas")
20 public class Persona {
21
22     @Id
23     @GeneratedValue(strategy = GenerationType.IDENTITY)
24     @Column(name = "id_persona")
25     private Long id;
26
27     @ManyToMany
28     @JoinTable( name = "persona_direccion",
29                 joinColumns = {@JoinColumn(name = "id_persona") },
30                 inverseJoinColumns = {@JoinColumn(name = "id_direccion") })
31     private List<Direccion> direcciones;
32
```

Relación muchos a muchos entre dos entidades (bidireccional)

Tenemos las mismas tablas y clases que en el caso anterior. Pero ahora, además de en la clase Persona tener un atributo que sea la lista de sus direcciones, necesitamos en la clase Direccion un atributo con la lista de personas a las que pertenece. Es, por tanto, una relación bidireccional.

Para esto, modificaremos la clase Direccion y añadimos el atributo de la lista de personas. Le pondremos la anotación ManyToMany con un mappedBy indicando que el mapeo ya está hecho en el atributo "direcciones" de la clase Persona.

```
13 @Entity
14 public class Direccion {
15
16     @Id
17     @GeneratedValue(strategy = GenerationType.IDENTITY)
18     private Long id;
19
20     @ManyToMany(mappedBy = "direcciones")
21     private List<Persona> persona;
22
```

Configurar qué hacer con entidades asociadas: Cascade

Por defecto, cuando solicitamos salvar en BBDD una entidad, sólo se aplica a dicha entidad y no a sus entidades relacionadas. Igual cuando solicitamos actualizarla o borrarla.

Esta política se puede modificar indicando el parámetro "cascade" dentro de las anotaciones de relación (OneToOne, OneToMany, ManyToMany).

Principalmente existen cuatro formas de configurar cascade:

- CascadeType.ALL → Cualquier operación realizada sobre una entidad también se realizará sobre sus entidades asociadas
- CascadeType.PERSIST → Cuando se realice la operación persist (insertar en BBDD)
- CascadeType.MERGE → Cuando se realice la operación merge (actualizar en BBDD)
- CascadeType.REMOVE → Cuando se realice la operación remove (borrar en BBDD)

Se debe tener mucho cuidado a la hora de configurar operaciones en cascada, sobre todo en las relaciones ManyToMany o OneToMany. Puede afectar mucho al rendimiento de nuestro programa.

Ejemplo de configuración:

```
16 @Entity
17 public class Propietario {
18
19     @Id
20     @GeneratedValue(strategy = GenerationType.IDENTITY)
21     private Long id;
22
23     @OneToMany(cascade = CascadeType.ALL)
24     private List<Direccion> direcciones;
25
```

Consultas perezosas

Cuando consultamos una entidad, existen dos opciones respecto a qué hacer con sus entidades asociadas:

- Podemos querer que también sean consultadas en el acto → EAGER
- Podemos querer que sólo se consulten cuando se necesiten → LAZY

Por defecto en JPA está configurado de este modo según el tipo de relación:

- OneToOne y ManyToOne: EAGER
- OneToMany y ManyToMany: LAZY

Se puede modificar utilizando el atributo "fetch" dentro de estas anotaciones. Ejemplo:

```
16 @Entity
17 public class Persona {
18
19     @Id
20     @GeneratedValue(strategy = GenerationType.IDENTITY)
21     private Long id;
22
23     @OneToOne(fetch = FetchType.LAZY)
24     @JoinColumn(name = "id_animal")
25     private Animal animal;
26
```

IMPORTANTE: las consultas LAZY sólo funcionan si la sesión aún está abierta.

Problemas con JOIN donde la columna no puede ser NULL

Es posible que, al insertar una entidad con una relación OneToMany que también se debe crear, Hibernate nos genere un error porque la columna por la que tenemos que hacer el JOIN (FK) no puede ser NULL o no tiene un valor predeterminado o por defecto.

Esto es porque Hibernate está intentando hacer insertar todos los registros y luego actualizar las claves foráneas.

La solución es indicar en nuestra anotación JoinColumn el atributo nullable = false. Ejemplo:

```
@OneToMany(cascade = CascadeType.ALL)
@JoinColumn(name = "id_pedido", nullable = false)
private List<PedidoLinea> lineas;
```

Realizar operaciones con la BBDD

Para realizar operación con la BBDD, lo primero será obtener un SessionFactory desde nuestra clase HibernateUtil. Luego abriremos una nueva sesión. Debemos controlar excepciones y recordar cerrar la sesión al final:

```
SessionFactory factory = null;
Session session = null;
try {
    factory = HibernateUtil.getSessionFactory();
    session = factory.openSession(); // Aquí abrimos sesión

    // Aquí realizamos las operaciones

}
catch(PersistenceException e) {
    e.printStackTrace(); // Aquí controlamos las excepciones
}
finally {
    if (session!=null) {
        session.close(); // Cerramos sesión
    }
}
```

- Prácticamente todas las excepciones heredan de PersistenceException. Además, esta hereda de RuntimeException (no es obligatorio capturarla). Para saber más sobre las excepciones en Hibernate, puedes consultar este tutorial: https://www.baeldung.com/hibernate-exceptions
- En lugar de openSession podríamos hacer getCurrentSession. En este caso:
    - Siempre se usa una única sesión. Lo cual es más rápido, pero no siempre es posible si tenemos aplicaciones concurrentes.
    - No es necesario cerrar la sesión
    - Hay que configurar la clase del contexto en el fichero hibernate.conf.xml

Para insertar, actualizar o borrar datos, será necesario iniciar una transacción y completarla con un commit o un rollback. Igual que hacíamos con JDBC. Con hibernate, esto se hace del siguiente modo (**para las consultas no es necesario**):

```
SessionFactory factory = null;
Session session = null;
try {
    factory = HibernateUtil.getSessionFactory();
    session = factory.openSession(); // Aquí abrimos sesión

    session.getTransaction().begin(); // Iniciamos transacción

    // Aquí realizamos las operaciones

    session.getTransaction().commit(); // Commit
}
catch(PersistenceException e) {
    if (session!=null) {
        session.getTransaction().rollback(); // rollback
    }
    e.printStackTrace();
}
finally {
    if (session!=null) {
        session.close(); // Cerramos sesión
    }
}
```

Insertar una nueva entidad

Utilizamos el método persist. El objeto que pasamos por parámetro se actualizará con la PK generada.

Hay que tener en cuenta, para las entidades relacionadas, que, si no tenemos configurado el atributo cascade, habrá que insertarlas también manualmente.

Ejemplo de inserción:

```
Animal a = new Animal();
a.setDescripcion("test");
session.persist(a);
```

Nota: el método save funciona de modo similar.

Actualizar una entidad

Utilizamos el método merge. El objeto que pasamos por parámetro no se modifica. En su lugar, se devuelve un nuevo objeto que es el resultado de la actualización.

Hay que tener en cuenta, para las entidades relacionadas, que, si no tenemos configurado el atributo cascade, habrá que actualizarlas también manualmente.

```
Animal a = new Animal();
a.setId(123L);
a.setDescripcion("test");
a = (Animal) session.merge(a);
```

Nota: el método update funciona de modo similar.

Eliminar una entidad

Utilizamos el método remove. Hay que tener en cuenta, para las entidades relacionadas, que, si no tenemos configurado el atributo cascade, habrá que borrarlas también manualmente.

```
Animal a = new Animal();
a.setId(123L);
session.remove(a);
```

Nota: el método delete funciona de modo similar.

Consultar una entidad

Utilizamos el método get. Si no se encuentra la entidad con ese ID, se devuelve null.

```
Persona p = session.get(Persona.class, 123L);
System.out.println(p);
```

Nota: el método find funciona de modo similar. El método load también, pero si no se encuentra la entidad, no devuelve null, lanza la excepción ObjectNotFoundException

Realizar una búsqueda utilizando Criteria

Podemos realizar una búsqueda configurando un criterio de filtrado. Se haría así:

1. Creamos una CriteriaQuery para la entidad por la que vamos a consultar:

```
CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();

CriteriaQuery<Animal> criteria = criteriaBuilder.createQuery(Animal.class);
```

2. Obtenemos el root llamando al método from. El root será como una representación de todos los datos sobre los que queremos luego filtrar:

```
Root<Animal> root = criteria.from(Animal.class);
```

3. Sobre el objeto criteria, aplicamos los filtros que queramos dentro del método where:

```
criteria.where(criteriaBuilder.like(root.get("descripcion"), "%perro%"));
```

4. Una vez configurado el criterio de filtro, lo aplicamos creando una query desde la sesión y obteniendo los resultados:

```
List<Animal> lista = session.createQuery(criteria).getResultList();
System.out.println(lista);
```

Si queremos obtener todos los resultados de la tabla, se hace del mismo modo, pero sin tener que aplicar ningún filtro: nos saltaríamos el paso 3.

Para saber más sobre Criteria API: https://www.baeldung.com/hibernate-criteria-queries

Realizar una búsqueda utilizando SQL

Podemos ejecutar una consulta utilizando SQL normal. Se haría del siguiente modo:

```
String sql = "select * from animal where descripcion like '%perro%'";
NativeQuery<Animal> query = session.createNativeQuery(sql, Animal.class);
lista = query.getResultList();
System.out.println(lista);
```

Realizar una búsqueda utilizando HQL

HQL es un lenguaje similar a SQL. Significa Hibernate Query Languaje. La principal diferencia con SQL es que las consultas se hacen y se escriben pensando en las entidades de nuestro modelo, no en las tablas de BBDD. Se puede obtener más información en la documentación oficial de Hibernate. Ejemplo:

```
String sql = "from Animal where descripcion like '%perro%'";
Query<Animal> query = session.createQuery(sql, Animal.class);
List<Animal> lista = query.getResultList();
System.out.println(lista);
```

Observa que “Animal” es el nombre de la clase, no de la tabla.