UD3-MAPEO OBJETO-RELACIONAL (ORM)

CFGS DAM - MP Acceso a Datos

DESFASE OBJETO-RELACIONAL

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.

Imaginemos que en nuestra aplicación Java tenemos la siguiente definición de una clase Student con sus atributos y métodos:

```
public class Student {
  private int studentId;
  private String firstName;
  private String lastName;
  private String contactNo;

  public Student(. . .) {
    . . .
  }

  // getters y setters
}
```

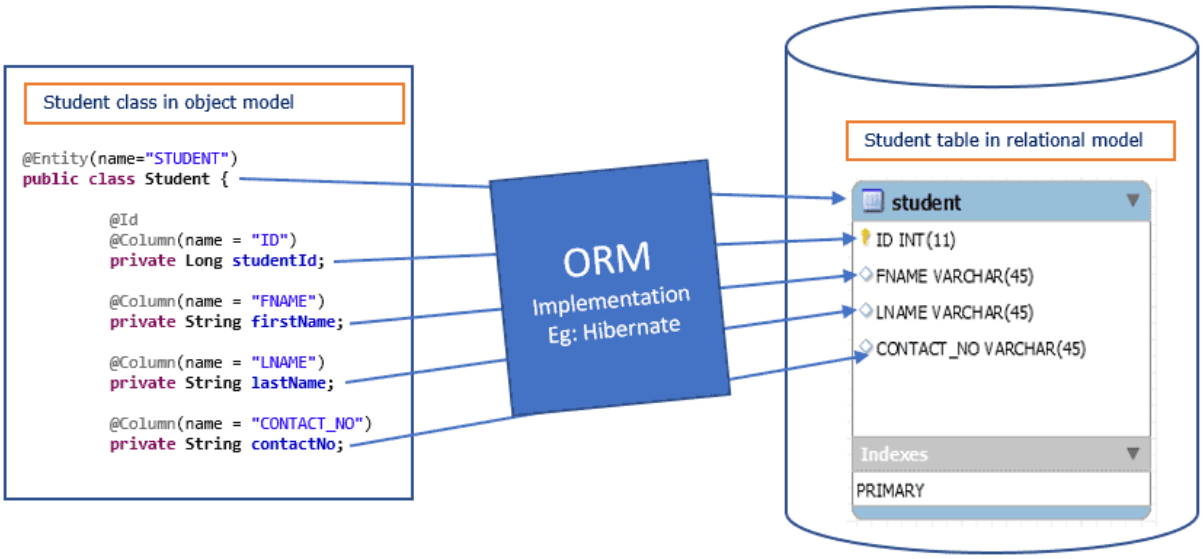
Mientras que en la base de datos tendremos una tabla cuyos campos se tendrán que corresponder con los atributos que hayamos definido anteriormente en esa clase. Puesto que son estructuras que no tienen nada que ver entre ellas, tenemos que hacer el mapeo manualmente, haciendo coincidir (a través de los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

```
CREATE TABLE students (
  ID INT PRIMARY KEY AUTO_INCREMENT;
  FNAME VARCHAR(45) NOT NULL,
  LNAME VARCHAR(45),
  CONTACT_NOLNAME VARCHAR(45)
);
```

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.

¿QUÉ ES EL MAPEO OBJETO-RELACIONAL?

El mapeo objeto-relacional (más conocido por su nombre en inglés, Object-Relational mapping, o sus siglas ORM, O/RM, y O/R mapping) es una técnica de programación para convertir datos entre el sistema de tipos utilizado en un lenguaje de programación orientado a objetos y la utilización de una base de datos relacional como motor de persistencia. En la práctica esto crea una base de datos orientada a objetos virtual, sobre la base de datos relacional. Esto posibilita el uso de las características propias de la orientación a objetos (básicamente herencia y polimorfismo).



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

```
. . .
String sentenceSql = "INSERT INTO student (FNAME, LNAME, CONT_NO)" +
  ") VALUES (?, ?, ?, ?)";
PreparedStatement sentence = connection.prepareStatement(sentenceSql);
sentence.setString(1, student.getFirstName());
sentence.setString(2, student.getLastName());
sentence.setInt(4, student.getContactNumber());
sentence.executeUpdate();

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

JPA

Jakarta Persistence(JPA, anteriormente Java Persistence API) es la especificación de la interfaz de programación para aplicaciones Java EE que describe la gestión de datos relacionales en aplicaciones Java EE empresariales. El objetivo que persigue el diseño de esta API es no perder las ventajas de la orientación a objetos al interactuar con una base de datos (siguiendo el patrón de mapeo objeto-relacional), como sí pasaba con EJB2, y permitir usar objetos regulares (conocidos como POJO).

La persistencia en este contexto cubre las siguientes 3 áreas:

- La API propiamente, definida en el paquete javax.persistence
- El lenguaje Jakarta Persistence Query Language (JPQL)
- Los metadatos objeto/relacional

Existen dos implementaciones estables de referencia del estándar JPA: Hibernate y EclipseLink. En esta asignatura utilizaremos Hibernate.

Javadocs oficials: JPA 2.0 – JSR 317, JPA 2.1/2.2 – JSR 338.

La JPA se origina a partir del trabajo del JSR 220 Expert Group el cual correspondía a EJB3. JPA 2.0 sería el trabajo del JSR 317 y posteriormente JPA 2.1 en el JSR 338. En agosto 2017 se publica la JPA 2.2 como revisión de JPA 2.1 dentro del mismo JSR 338. La JPA fue renombrada como Jakarta Persistence en 2019 y la versión 3.0 fue publicada en 2020.

HIBERNATE

En nuestro caso, en esta materia utilizaremos la librería Hibernate como ORM ya que es la implementación más utilizada de la Jakarta Persistence (JPA). Concretamente trabajaremos con la versión **Hibernate 5.5 (la última versión estable es la 5.6 y salió publicada en octubre 2021)**. Habrá que tenerlo en cuenta puesto que algunas clases/métodos pueden variar respecto a versiones anteriores de este framework, especialmente a la hora de configurar e implementar el gestor de entidades.

EJEMPLO INICIAL HIBERNATE

Anteriormente hemos visto los problemas de codificar manualmente la conexión a un RDBMS desde nuestro código. 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 los objetos 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.

```
@Entity
@Table(name="students")
public class Student {
  @Id // Marca el campo como la clave de la tabla
  @GeneratedValue(strategy = IDENTITY)
  @Column(name="FNAME")
  private int firstName;
  @Column(name="LNAME")
  private String lastName;
  @Column(name="CONTACT_NO")
  private String contactNo;

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

  // getters y setters
}
```

Así, podemos simplemente establecer una conexión 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.

```
EntityManagerFactory entityManagerFactory =
Persistence.createEntityManagerFactory("default");
EntityManager entityManager = entityManagerFactory.createEntityManager();
entityManager.getTransaction().begin();
entityManager.persist(employee);
entityManager.getTransaction().commit();
entityManager.close();
entityManagerFactory.close();
```

VERSIÓN JDK

Para este curso os recomiendo que utilices la última versión LTS publicada de JDK que es la 17 que salió en 2021.

https://www.oracle.com/java/technologies/javase/jdk17-archive-downloads.html

CONFIGURACIÓN EN INTELLIJ

Seguiremos el siguiente tutorial oficial para configurar Hibernate en un proyecto Java Enterprise utilizando el IDE IntelliJ IDEA Ultimate de JetBrains:

https://blog.jetbrains.com/idea/2021/02/creating-a-simple-jpa-application/

En resumen, los pasos para configurar nuestro proyecto serán:

1. Creación de un proyecto Java Enterprise con la librería Hibernate 5.6.
2. Adición de las dependencias a la base de datos dentro del fichero pom.xml. Abrimos el pom y dentro de la sección dependencies pulsamos Alt+Insert y añadimos una nueva dependencia llamada mysql 8.0.28. Vamos al menú lateral de Maven y pulsamos “Reload All Maven projects” para que se resuelva la nueva dependencia al conector MySQL.
3. Creación de las entidades (clases persistentes) dentro de un package llamado entity.
4. Creación de una clase Main que creará el gestor de entidades EntityManager a través de un EntityManagerFactory.
5. Creación de una **unidad de persistencia** dentro del fichero **persistence.xml**

dónde se especifique cómo nos conectaremos a la BD.

Comentario: Hibernate facilita muchos mensajes informativos en rojo que también aparecen en la consola. Es normal. Revisad el contenido de los mismos.

FICHERO POM.XML

Dentro del fichero **pom.xml** de Maven será necesario indicar las dependencias a mysql que tengas nuestro proyecto y refrescar maven una vez se haya modificado dicho fichero:

```
<dependency>
   <groupId>mysql</groupId>
   <artifactId>mysql-connector-java</artifactId>
   <version>8.0.26</version>
</dependency>
```

FICHERO PERSISTENCE.XML DE HIBERNATE

En el fichero **persistence.xml (ubicado en la carpeta META-INF de la raíz del proyecto)** debemos de especificar la configuración de conexión a la base de datos donde deseemos realizar el mapeo objeto-relacional de nuestro código. Este fichero (que debe ser único) puede contener más de una unidad de persistencia, cada una con un nombre diferente. Este nombre es el que se indica en el método para crear el EntityManagerFactory.

La cabecera del fichero persistence.xml para JPA 2.2 debe ser:

```
<persistence version="2.2" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_2.xsd">
```

En la UP del fichero persistence.xml hay las propiedades necesarias para conseguir la conexión con el DBMS y también otras propiedades referentes al funcionamiento de la persistencia.

Aquí tienes el siguiente código de referencia para una unidad de persistencia que se conecta a una base de datos local MySQL llamada “ad_ud3”, que escucha en el puerto 3306 con un usuario “alumno” y contraseña “alumno”:

```
<persistence-unit name="default">
   <class>entity.Employee</class>
   <properties>
       <property name="javax.persistence.jdbc.driver"
value="com.mysql.cj.jdbc.Driver"/>
       <property name="javax.persistence.jdbc.url"
value="jdbc:mysql://127.0.0.1:3306/ad_ud3"/>
       <property name="javax.persistence.jdbc.user" value="alumno"/>
       <property name="javax.persistence.jdbc.password" value="alumno"/>
       <property name="hibernate.dialect"
value="org.hibernate.dialect.MySQL8Dialect"/>
       <property name="hibernate.hbm2ddl.auto" value="create"/>
   </properties>

</persistence-unit>
```

Si quisiéramos conectarnos a diferentes DBMS entonces deberíamos especificar:

- Oracle (puerto habitual 1521):
    - Driver: ojdbc8.jar
    - URL: jdbc:oracle:thin:@//IP:PORT/nombreBD
    - Clase JDBC: oracle.jdbc.driver.OracleDriver
- PostgreSQL (puerto habitual 5432):
    - Driver: postgresql-42.2.5.jar
    - URL: jdbc:postgresql://IP:PORT/nombreBD
    - Clase JDBC: org.postgresql.Driver
- MySQL8 (puerto habitual 3306):
    - Driver: mysql-connector-java-8.0.26.jar (disponible en el moodle)
    - URL: jdbc:mysql://IP:PORT/nomBD
    - Clase JDBC: com.mysql.cj.jdbc.Driver

Adicionalmente, en las propiedades se pueden configurar las siguientes exclusivas de Hibernate:

- **hibernate.dialect**. Indica el "dialecto" de SQL que debe usar Hibernate para comunicarse con la base de datos. Lista completa. Si un proyecto no tiene esta propiedad y hay algún problema que no permite la conexión (por ejemplo, contraseña incorrecta), Hibernate no informa correcta de la causa del problema porque no tiene el dialecto informado. Además, en el caso de que Hibernate tenga

que proceder a la creación o modificación de tablas, si el dialecto no es el adecuado, puede no generar adecuadamente las sentencias. Ejemplos de valores:

  - Oracle18g: org.hibernate.dialect.Oracle12cDialect
  - PostgreSQL9.6: org.hibernate.dialect.PostgreSQL96Dialect
  - PostgreSQL10: org.hibernate.dialect.PostgreSQL10Dialect
  - MySQL5: org.hibernate.dialect.MySQL57Dialect
  - MySQL8: org.hibernate.dialect.MySQL8Dialect

- **hibernate.show_sql**. Imprime en la salida estándar de Java las sentencias SQL que Hibernate va ejecutando en la base de datos.

```
<property name="hibernate.show_sql" value="true"/>
```

- **hibernate.format_sql**. Para que el código SQL que genera Hibernate se muestre formateado.

```
<property name="hibernate.format_sql" value="true"/>
```

- **hibernate.hbm2ddl.auto**. El acrónimo hbm2ddl viene a significar “de Hibernate a DDL (lenguaje de definición de datos)”. Por DDL entendemos las sentencias SQL que definen la propia estructura de datos: tablas, índices, claves, columnas… Con esta opción, indicamos a Hibernate qué estrategia debe aplicar para mantener sincronizado el modelo de datos de la aplicación (clases) con la estructura de la base de datos (tablas). Esta sincronización se lleva a cabo durante el arranque de la aplicación. JPA provee la misma funcionalidad con la propiedad jakarta.persistence.schema-generation.database.action pero con menos opciones. La siguiente tabla muestra la equivalencia y cómo funcionan.

Hibernate	JPA	
none	none	No hace nada
create-only	create	Crea todo el esquema.
drop	drop	Elimina el esquema y no lo crea.
create	drop-and-create	Crea siempre el esquema. Si ya existe, lo elimina antes.
create-drop		Igual que create, pero al cerrarse el EntityManagerFactory se elimina.
validate		Comprueba que el modelo de entidades es coherente con el esquema de la base de datos. Si no, se lanza una excepción de tipo SchemaManagementException lo que impide que la aplicación arranque.
update		Intenta mantener siempre la sincronización, creando y actualizado lo que sea necesario

¿Qué alternativa escoger?

La opción **create** sería perfecta en la primera ejecución si queremos que cree las tablas y **drop-and-create** en las siguientes, mientras se afina la definición de las anotaciones.

Mientras desarrollamos, **update** es la más útil, si bien hay que tener en cuenta que si hay datos es posible que algunas actualizaciones no puedan realizarse porque, por ejemplo, haya que redefinir una columna como no nula y ya existan registros con ese valor vacío. Es la opción que usaremos en el curso.

Para producción/explotacion, **none** o a lo sumo **validate** con el fin de asegurar que la aplicación va a usar la versión del esquema correcta. Lo más seguro es proporcionar un script SQL de creación inicial, incluyendo los registros predefinidos, triggers, sentencias check, etc. Cuando llegue el momento de actualizar, se crearán los scripts convenientes para aplicar los cambios producidos entre versiones.

ENTITIES EN HIBERNATE

En nuestras aplicaciones convivirán:

- Classes no persistentes: donde ninguno de sus objetos necesita ser almacenado en una BD.
- Clases persistentes: donde alguno de sus objetos necesita ser almacenado.

En el caso de JPA, para informar de qué clases son persistentes se facilitan dos mecanismos:

- Annotations: uso de anotaciones (similar al marcado JAXB)
- Fichero XML: dónde se especifican las clases persistentes sin tocar el código de la clase en sí.

Estos dos mecanismos pueden convivir y en el caso de coexistir el mapeado XML prevalece sobre las anotaciones. Esto permite que las anotaciones JPA puedan ser sobreescritas con marcas diferentes vía XML.

Independientemente del tipo de mapeo (anotaciones o XML), debemos tener presente que la herramienta ORM hace la traducción:

objeto de clase ← → fila de tabla

Para JPA, las entidades son aquellos objetos de los cuales se desea guardar su estado y que acabarán transformándose en tablas y relaciones.

Todas las entidades deben poder identificarse de forma única a partir de su estado. Normalmente, será suficiente con una pequeña parte de sus atributos para conseguir la identificación. La selección de atributos que cumplan este objetivo se llaman identificadores, y en el SGBD actuarán como clave primaria.

ANNOTATIONS

Primeras anotaciones obligatorias

- Nivel de clase:
  - **@Entity:** para indicar que es una clase persistente
- Nivel de atributos:
  - **@Id:** para indicar campo PK (más adelante ya se explicará cómo hacerlo en claves compuestas)

- **@GeneratedValue**: se indica que la clave primaria se generará de forma automática. Existen 4 valores para el campo strategy:
  - **GenerationType.AUTO**: tipo de generación por defecto y permite que el proveedor de persistencia JPA elija la estrategia de generación en base al DBMS. Solo para pruebas(desarrollo).
  - **GenerationType.TABLE**: utiliza una tabla normal del DBMS que guarda uno o diversos contadores que ayudarán a generar un valor único cada vez que sea necesario. Sirve para cualquier DBMS. Los bloqueos en el acceso a dicha tabla pueden ralentizar su uso.
  - **GenerationType.IDENTITY**: está pensada específicamente para aquellos DBMS que disponen de este tipo autoincremental como MySQL, Access, PostgreSQL, Oracle12+ u otros DBMS.
  - **GenerationType.SEQUENCE**: utiliza una secuencia propia del DBMS para generar valores únicos. Válido para DBMS como SQLServer, PostgreSQL u Oracle.Precisa de la anotación @SequenceGenerator. MySQL8 no contiene secuencias, pero los proveedores JPA pueden simularlas. En el caso de Hibernate, para cada tabla con esta estrategia, crea una tabla contador para simular la secuencia.

  (Nota: en el caso de utilizar esta anotación será necesario borrar el atributo identificador del constructor y eliminar el método setter de dicho atributo ya que en ningún caso se deberá de modificar dicho valor autogenerado).
- **@Transient**: atributo no persistente. Los atributos no marcados así se considerarán persistentes y si no se indica nada más se considerará que se almacenarán en una columna que tenga el mismo nombre que el atributo.
- **@Colum**: anotación muy parametrizable, ya que permite expresar muchas características referidas a la columna. Entre otras, podemos expresar el nombre (parámetro name), el tamaño de la columna (parámetro length) cuando se quiera limitar el número de carácteres, si se aceptan valores nulos (parámetro nullable) o si los valores de la columna tendrán que ser únicos para cada registro(parámetro unique).

Más información sobre annotations en Hibernate en la documentación oficial: https://docs.jboss.org/hibernate/stable/annotations/reference/en/html_single/#entity

ENTITYMANAGER EN HIBERNATE

En JPA, el EntityManager es quién gestiona las entidades persistentes y, por tanto, se necesita crear un EntityManager. Sobre esta clase recae toda la funcionalidad referida a

los procesos de persistencia y sincronización de las entidades. Se trata seguramente de la clase más importante de la biblioteca JPA.

Para crear un EntityManager es necesario tener un EntityManagerFactory y un EntityManagerFactory se crea con Persistence.createEntityManagerFactory al cual se le tiene que indicar un nombre y una unidad de persistencia (UP):

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory
("nombreUP")
```

Una vez obtenido el EntityManagerFactory es necesario obtener el EntityManager vía createEntityManager:

```
EntityManager entityManager = entityManagerFactory.createEntityManager();
```

¿Es posible modificar las propiedades de una UP?

Las propiedades de la UP definidas dentro del fichero persistence.xml pueden ser sustituidas/completadas en la creación del EntityManagerFactory, utilitzando un método alternativo de creación donde se pasen por parámetro los cambios en un map llamado props:

```
HashMap<String,String> props = new HashMap();
props.put("hibernate.hbm2ddl.auto", "create");
EntityManagerFactory emf =
Persistence.createEntityManagerFactory("nomUP",props)
```

CONFIRMANDO LA PERSISTENCIA

Para informar de que los objetos de una clase son persistentes estos deben pasar a ser “entidades” gestionadas por JPA (es decir, objetos almacenados en la BD) y los cambios se aplicarán a través de métodos de la clase EntityManager.

Primeros métodos para gestionar datos persistentes:

- **em.persist(obj)**: marca el objeto para hacerlo persistence (alta) y pasar a ser controlado por el EntityManager. No pasa a la BD. Si el EntityManager ya está controlando un objeto con el mismo ID tendría que fallar con una EntityExistException. En el caso de Hibernate, se genera la excepción al ejecutar el persist.
- **em.flush()**: registra los cambios en la BD (dentro de una transacción, sin hacer commit, podríamos hacer rollback)
- **em.getTransaction.begin()**: inicia la transacción
- **em.getTransaction.commit()**: cierra la transacción activa validando los cambios pendientes en la BD. Falla si la transacción no está activa.
- **em.getTransaction.rollback()**: cierra la transacción activa sin validar los

cambios pendientes a la BD. Falla si la transacción no está activa.
- **em.getTransaction.isActive()**: informa de si tenemos o no una transacción activa.
- **em.find(NomClasse.class,valorDeCampId)**: recupera un objeto persistente de la BD. Recupera null si no existe.
- **em.createQuery (llenguatge JPQL)**: para ejecutar consultas similares a SQL, pero recuperando objetos.

¿Qué pasa cuando se produce alguna PersistenceException? Pues que la transacción activa, si hay, queda marcada como transacción de "solo rollback" y si se intenta hacer un commit, se producirá una excepción. Es responsabilidad del programa invocar rollback, de lo contrario, las modificaciones pasadas a la BD pendientes de validar (vía flush) pueden ser validadas → Mucho cuidado!!

- **em.getTransaction.getRollbackOnly()**: informa si la transacción activa está marcada de "solo rollback". Falla si no hay transacción activa.
- **em.getTransaction.setRollbackOnly()**: marca la transacción activa de "solo rollback". Falla si no hay transacción activa.

EXCEPCIONES

Una vez hemos creado una transacción con la classe EntityManager, la transacción se activa invocando el método begin y finaliza cuando se invoca commit, pero no es necesario invocar rollback en caso de error. **JPA invoca automáticamente la revocación de las acciones, cuando se lance cualquier excepción, a partir de la última invocación de begin.**

Todas las excepciones generadas por JPA son de tipo **RuntimeException**. Este tipo de excepción presenta la particularidad que no debe declararse en la firma del método y por tanto, **el uso de try-catch no es obligatorio**.

Este tipo de transacciones presentan la ventaja de poder escribir un código más limpio (sin sentencias try-catch intermedias), pero por el contrario el desarrollador debe ir mucho más en cuenta de no olvidarse de realizar el tratamiento de las excepciones. Para facilitar este tratamiento, **todas las excepciones JPA heredan de un antecesor común llamado PersistenceException.**

RELACIONES

JPA obliga también a marcar las relaciones que puedan haber entre diferentes entidades según su cardinalidad. Se prevén cuatro tipos de cardinalidad: **@ManyToOne, @OneToOne, @OneToMany o @ManyToMany**.
Las dos primeras, **@ManyToOne y @OneToOne** se llaman de valor único ya que se corresponden con un atributo que hace referencia a un único objeto. La diferencia entre OneToOne i ManyToOne es más conceptual que práctica, ya que ambos (utilizados por defecto) producen la misma conversión: una clave foránea hacia la tabla donde se guardará la relación. Como detalle, aclararemos que tendremos que sustituirla anotación Column por JoinColumn, indicando que nos encontramos con una clave foránea. La especificación de JoinColumn es muy similar a Column.
Por otro lado, las relaciones **@OneToMany y @ManyToMany** se llaman multivaluadas porque se corresponden con atributos de tipo array, lista, map… JPA necesita que los atributos se declaren mediante las interfaces correspondientes. Nunca se tiene que declarar un atributo que representa una relación multivaluada con una clase concreta.

CARGA DE DATOS EN LAS RELACIONES

Cuando vinculamos dos clases a través de relaciones es posible configurar dos actuaciones muy importantes:

- **Fetch**: permite indicar al EntityManager en que momento se cargan en memoria los objetos de las otras clases a causa de una relación. Imaginemos que tenemos una clase Multa relacionada con la clase Conductor y la clase Vehículo y buscamos en memoria la multa 1000:

  ```
  Multa m = em.find(Multa.class, 1000);
  ```

  La clase Multa contiene las referencias a los correspondientes objectos Conductor y Vehículo. JPA los cargará también en memoria?

  Parece que la respuesta debería ser afirmativa y... si Conductor contuviera una referencia hacia sus multas (List<Multa> multas), ¿implicaría que por el hecho de cargar el Conductor en memoria, también cargaría sus multas? Y cada multa, su conductor y su vehículo??

  JPA incorpora la solución gracias a la cláusula fetch.

  La cláusula fetch en los campos relacionales permite decidir la actuación que se desea:

  - **FetchType.LAZY** (actuación vago-diferida): Carga objetos referenciados cuando se necesiten.
  - **FetchType.EAGER** (actuación inmediata): Carga objetos referenciados inmediatamente.

  Esta cláusula fetch, muy interesante en los campos relacionales, es aplicable a todos los atributos y puede ser especialmente interesante en atributos pesados (BLOB, CLOB,...).

  La cláusula fetch, por defecto (en todo tipo de atributo) tiene el valor EAGER excepto en las relaciones OneToMany y ManyToMany, donde tiene el valor LAZY (lógico, pues en estos casos un objeto puede estar relacionado con muchos-muchos-muchos objetos)

- **cascade**: Permite indicar al EntityManager cómo actuar sobre los objetos referenciados cuando un objeto lo hacemos persistente (persist) o cuando le

aplicamos otras acciones (merge, refresh, remove y detach, aún pendientes de conocer).

Las actuaciones permitidas son:

- CascadeType.PERSIST: propagates the persist operation from a parent to a child entity.
- CascadeType.REFRESH:the child entity also gets reloaded from the database whenever the parent entity is refreshed.
- CascadeType.DETACH: when the entity parent is deleted, the child entity is detached, but not deleted
- CascadeType.MERGE: copies the state of the given object onto the persistent object with the same identifier. This type propagates the merge operation from a parent to a child entity.
- CascadeType.REMOVE: propagates the remove operation from parent to child entity. Similar to JPA's CascadeType.REMOVE, we have CascadeType.DELETE, which is specific to Hibernate. There is no difference between the two.
- CascadeType.ALL (las engloba a todas): propagates all operations from a parent to a child entity.

EJEMPLO RELACIÓN 1:1

Un trabajador tendrá una única tarjeta identificativa y una tarjeta identificativa estará asociada a un único trabajador.

```
@Entity
@Table(name="employees")
public class Employee {
   ...
   @OneToOne
   @JoinColumn(name="idcard")
   private Card card;
   ...
}
```

```
@Entity
@Table(name="cards")
public class Card {
    ...
    @OneToOne(mappedBy=”card”)

    private Employee employee;
    ...
}
```

EJEMPLO RELACIÓN 1:N

Una tarea será realizada por un único trabajador y un trabajador tendrá múltiples tareas a realizar.

```
@Entity
@Table(name="tasks")
public class Task {
    …
    @ManyToOne
    @JoinColumn(name="idemployee")
    private Employee employee;
    ...
}


@Entity
@Table(name="employees")
public class Employee {
    …
    @OneToMany(mappedBy = "employee")
    private List<Task> ltasks;

   ...
}
```

EJEMPLO RELACIÓN N:N

Un trabajador puede estar asociado a múltiples departamentos y en un departamento puede haber múltiples trabajadores.

```
@Entity
@Table(name="employees")
public class Employee {
    ...
    @ManyToMany
    @JoinTable(name = "emp_dep",joinColumns =
@JoinColumn(name="idemployee"),
    inverseJoinColumns = @JoinColumn(name="iddepartment"))
    private List<Department> ldepartments;
   ...
}
```

```
@Entity
@Table(name="departments")
public class Department {
    ...
    @ManyToMany(mappedBy = "ldepartments")
    private List<Employee> lemployees;
   ...
}
```

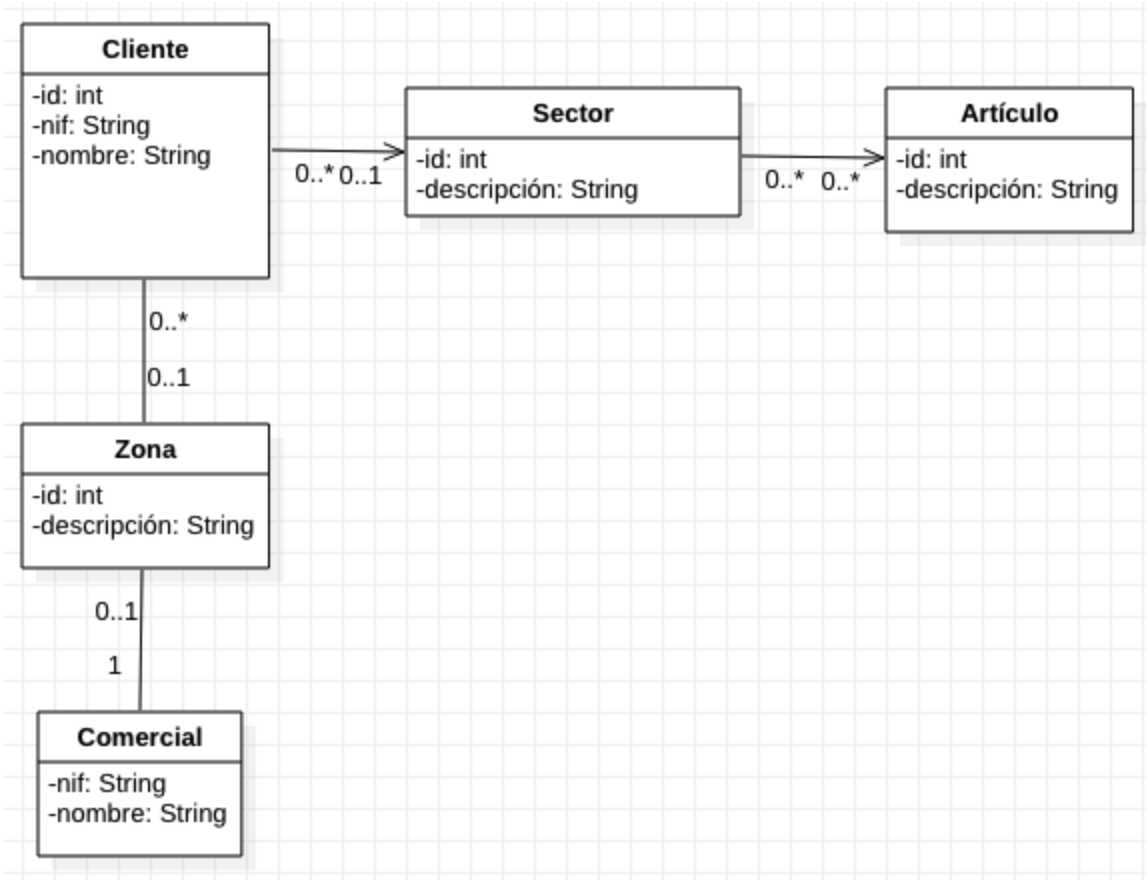
Explicación atributo mappedby

Cuando deseamos realizar una relación **bidireccional** entre dos entidades, tendremos que identificar una entidad como propietaria de la relación y la otra será la referenciada. Esto es necesario para que Hibernate no duplique el almacenaje de la relación en la base de datos. En el atributo mappedBy es dónde especificaremos el nombre del atributo de mapeo de asociación en el lado propietario. En el ejemplo anterior, Employee sería la entidad propietaria y Department sería la entidad referenciada.

El lado que tiene el atributo mappedBy es el lado inverso. El lado que no tiene el atributo mappedBy es el propietario.

EJEMPLO COMPLETO

Si tomamos como referencia el siguiente diagrama de clases:



El código que se corresponde con dicho diagrama es:

```
@Entity
@Table(name="articulos")
public class Articulo {
@Id
@GeneratedValue
private Integer id;
private String descripcion;
...
}
@Entity
public class Cliente{
@Id
@GeneratedValue
```

```
private Integer id;
private String nif;
private String nombre;
@ManyToOne
private Zona zona=null;
@OneToOne
private Sector sector=null;
...
}
@Entity
public class Comercial{
@Id
private String nif;
private String nombre;
@OneToOne
private Zona zona;
...
}
@Entity
public class Sector{
@Id
private String id;
private String descripcion;
@ManyToMany(fetch= FetchType.LAZY, cascade= CascadeType.ALL)
private List<Articulo> articulos;
...
}
@Entity
public class Zona{
@Id
private String id;
private String descripcion;
@OneToMany(mappedBy = "zona", fetch=FetchType.LAZY)
@OrderBy(value="nom")
private List<Cliente> clientes;
@OneToOne(mappedBy="zona")
private Comercial comercial;
...
}
```

OBJETOS INCRUSTADOS (EMBEDDED OBJECTS)

Llamamos objetos incrustados a aquellos objetos no marcados como entidades que están contenidos, directa o indirectamente, en una entidad.

Los objetos incrustados tienen total dependencia con la entidad que los contiene. Por eso no tienen identificador. De hecho, podríamos decir que son una parte de la entidad a la que pertenecen, pero que por razones de diseño sus datos se han extraído constituyéndose en un objeto propio.

Para marcar una clase como objeto incrustado debemos marcarla como **@Embeddable** y no como @Entity que se utiliza para las clases persistentes.

```
@Embeddable
public class Address {
    private String via;
    private String codigoPostal;
    @ManyToOne
    @JoinColumn(name="idpoblacion")
    private Poblacion poblacion;
}
```

Si queremos que la clase Employee contenga un objeto embebido de la clase Address entonces lo indicaremos con la notación **@Embeddeb**:

```
@Entity
public class Employee {
    ...
    @Embedded
    private Address address;
    ...
}
```

Eso sí, si quisiéramos cambiar el nombre de alguno de los atributos de la classe Address desde la clase Employee, tendríamos que utilizar la anotation @AttributeOverride de la siguiente forma:

```
@Entity
public class Employee {
    ...
    @Embedded
    @AttributeOverride(name="via", column=@Column(name="calle"))
    private Address address;
    ...
}
```

En este ejemplo, hemos sustituido el nombre del atributo vía por calle.

Ciertamente, a nivel de BD, podríamos tener una tabla Address o una tabla Email con PK a la cual se hiciese referencia desde la tabla Employee … pero cuando la clase employee solamente tienen un único objeto Address o Email, es habitual que éste esté incrustado dentro de la tabla Employee.

COLECCIONES DE OBJETOS BÁSICOS

Si queremos incorporar a la clase Employee la posibilidad de introducir una lista de los teléfonos (objetos String) de forma que la clase quedase así:

```
class Employee {
    ...
    List<String> phones;
}
```

La implementación lógica consiste en generar una tabla que “cuelgue” de la tabla Employee y que tenga los teléfonos asociados.

Para conseguirlo tenemos las siguientes anotations:

- @ElementCollection: para indicar que el campo es una colección.
- @CollectionTable: para definir cómo debe ser la tabla que recoge los elementos.
    - name: el nombre de la tabla
    - joinColumns: el nombre de la columna que hace de enlace y el nombre de la FK
    - uniqueConstraints: las restricciones de unicidad que pueda interesar (no se puede definir PK).
- @Column: Para indicar el nombre de la columna que contiene el valor de la colección debe tener dentro de la nueva tabla

La configuración de las anotaciones que permiten reflejar dicho comportamiento con Hibernate en nuestro supuesto deberían de configurarse de la siguiente forma:

```
@Entity
public class Employee {
   ...
   @ElementCollection
   @CollectionTable(name="employee_phones",
   joinColumns = @JoinColumn(name="employeeid",
   foreignKey= @ForeignKey(name="employee_phones_fk_employeeid")),
   uniqueConstraints =@UniqueConstraint(columnNames={"employeeid","phone"},
   name="emloyee_phones_uniq_employeeid_phone"))
   @Column(name="phone", length=15, nullable=false)
   @OrderColumn(name="order", nullable=false,columnDefinition="NUMERIC(2)")
   private List<String> phones;

}
```

COLECCIONES DE OBJETOS INCRUSTADOS

En este caso es necesario combinar la sintaxis que se ha visto para objetos incrustados y para colecciones de objetos básicos.

Dada la siguiente clase embeddedable Email:

```
@Embeddable
public class Email {
   @Column(length = 320)
   private String email;
   private static Pattern pattern = Pattern.compile("^(.+)@(.+)$");
   protected Email() {}
   public Email(String email) {
       setEmail(email);
   }
   public String getEmail() {
       return email;
   }
   public void setEmail(String email) {
       //Matcher matcher = pattern.matcher(email);
       if (!email.matches(pattern) || email.length()>320 ||
           email.substring(0,email.indexOf('@')).length()>64) {
```

```
            throw new RuntimeException(email + " no es un email electrónico
            válido");
        }
        this.email = email.toLowerCase();
    }
}
```

Tenemos una clase Employee que tendrá una lista de objetos incrustados Email

```
@Entity
public class Employee {
    @Id
    @Column(length=15)
    private String dni;

    @Basic(optional=false)
    @Column(length=50, nullable=false)
    private String nom;

    @ElementCollection
    @AttributeOverrides({
        @AttributeOverride(name = "email", column = @Column(name="email",
        length=320))
    })
    @CollectionTable(name="persona_emails",
    joinColumns = @JoinColumn(name="employeeid",
    foreignKey= @ForeignKey(name="employee_emails_fk_employeeid")),
    uniqueConstraints =@UniqueConstraint(columnNames={"employeeid","email"},
    name="employee_emails_uniq_employeeid_email"))
    @OrderColumn(name="order",nullable=false,columnDefinition="numeric(2)")
    private List<Email> emails;
    ...
    public boolean addEmail(Email email) {
        if (!emails.contains(email)) {
            emails.add(email);
            return true;
        } else {
            return false;
        }
    }
    public Iterable<Email> getEmails() {
        return emails;
    }
```

```
    public boolean removeEmail(Email email) {
        return emails.remove(email);
    }
}
```

HERENCIA

Supongamos que tenemos una clase Persona con dos clases derivadas Alumno y Profesor.

¿Cómo debe ser la persistencia de sus objetos en un DBMS? ¿Cómo se haría el diseño?

Si pensamos en el modelo E-R, seguro que diseñaría una entidad PERSONA con una especialización ALUMNO y PROFESOR. Y, ¿cómo sería la traducción a un modelo relacional? Recordaréis que existen tres estrategias y JPA permite implementar todas ellas;

- 1 única tabla PERSONA para almacenar todas las personas, sean alumnos o profesores. Podría considerarse la implementación más sencilla. Por tanto, contendrá columnas para los datos de PERSONA y columnas para los datos específicos de ALUMNO y de PROFESOR. JPA permite esta implementación con la llamada estrategia **SINGLE_TABLE**. Esto implica que esta tabla debe admitir muchos valores nulos:
    - Una fila que almacene un ALUMNO, tendrá nulls en las columnas correspondiente a PROFESOR.
    - Una fila que almacene un PROFESOR, tendrá a nulls en las columnas correspondientes a ALUMNO.
    - Una fila que almacene a una PERSONA que no sea ni ALUMNO ni PROFESOR (en caso de que se permita), tendrá a nulls en las columnas dedicadas a los datos específicos de ALUMNO y a los datos específicos de PROFESOR.
    - La tabla también debe contener una columna que permita distinguir las filas de las diversas entidades.

```
@Entity
@Inheritance(strategy= InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="tipo_persona",discriminatorType=
```

```
DiscriminatorType.INTEGER)
@DiscriminatorValue(value="0")
public class Persona{
	...
}


@Entity
@DiscriminatorValue(value="1")
public class Alumno extends Persona {
	...
}

@Entity
@DiscriminatorValue(value="2")
public class Profesor extends Persona {
	...
}
```

- 1 tabla para cada entidad, totalmente independientes. Esta estrategia consiste en tener una tabla para cada clase, estando los campos correspondientes a los atributos comunes (los atributos del padre) repetidos en cada una de estas tablas. JPA permite esta implementación con la llamada estrategia TABLE_PER_CLASS.
    - Tabla ALUMNO pensada para guardar las instancias que sean ALUMNO.
    - Tabla PROFESOR pensada para guardar las instancias que sean PROFESOR.
    - Tabla PERSONA pensada para guardar las instancias que sean PERSONA (ni ALUMNO ni PROFESOR en caso de que se permita este hecho). En caso de que la clase BASE sea abstracta, la correspondiente tabla no tiene razón de existir. Es decir, si la clase Persona es abstracta (no puede haber objetos Persona), no existirá su tabla.

```
@Entity
@Inheritance(strategy= InheritanceType.TABLE_PER_CLASS)
public class Persona{
	...
}

@Entity
```

```
public class Alumno{
	...
}

@Entity
public class Profesor {
	...
}
```

Nota: si la clase Persona fuese abstracta le tendríamos que añadir la anotation @MappedSuperclass y suprimir la anotación @Entity para que no se cree una tabla Persona en la BD ya que no se podrán crear objetos de dicha clase.

- 1 tabla con los datos comunes y tablas para los datos específicos de cada clase. JPA permite esta implementación con la llamada estrategia **JOINED**.
    - Tabla PERSONA pensada para guardar los datos comunitarios
    - Tabla ALUMNO pensada para guardar los datos específicos de alumno, con FK a PERSONA
    - Tabla PROFESOR pensada para guardar los datos específicos de profesor, con FK a PERSONA.

  La tabla puede contener o no una columna que permita distinguir las filas de cada entidad.

```
@Entity
@Table(name=”personas”)
@Inheritance(strategy= InheritanceType.JOINED)
@DiscriminatorColumn(name="tipo_persona",discriminatorType=DiscriminatorTyp
e.INTEGER)
@DiscriminatorValue(value="0")
public class Persona {
	...
}


@Entity
@Table(name=”alumnos”)
@PrimaryKeyJoinColumn(name="idpersona")
@DiscriminatorValue(value="1")
public class Alumno extends Persona{
```

```
	...
}
```

```
@Entity
@Table(name=”profesores”)
@PrimaryKeyJoinColumn(name="idpersona")
@DiscriminatorValue(value="2")
public class Profesor extends Persona{
	...
}
```

Comentario: Hibernate, para gestionar la persistencia de los objetos con estrategias TABLE_PER_CLASS y JOINED, crea unas tablas de soporte con prefijo HT, que no deben preocuparnos (contienen datos temporales).

IDENTIFICADORES COMPUESTOS

Una clave primaria compuesta es una combinación de dos o más columnas para formar una clave primaria para una tabla.

En JPA, tenemos dos opciones para definir las claves compuestas: las anotaciones @IdClass y @EmbeddedId.

Más información y ejemplos: https://www.baeldung.com/jpa-composite-primary-keys

LENGUAJE DE CONSULTA JPQL

JPQL (Java Persistence Query Language) es el lenguaje de consulta que utiliza JPA. Es una sintaxis muy similar a SQL con adaptaciones específicas para trabajar con las entidades persistentes de la aplicación en lugar de tablas de una base de datos.

Para ejecutar una SELECT, se procede a crear una Query q con distintas posibilidades:

- q=em.createQuery(consulta) por una consulta JPA (con o sin parámetros)
- q=em.createNamedQuery(nombreConsulta) por una consulta JPA con nombre (con o sin parámetros)
- q=em.createNativeQuery(consulta) por una consulta SQL tradicional (con o sin

parámetros).

En cualquier caso, la consulta se ejecuta con los métodos q.getResultList() si la consulta retorna diversas entidades o con q.getSingleResult() si retorna una única instancia.

```
Query qry = em.createQuery( "select z from Zona z where z.descripcion='Vigo
y alrededores'");
zona = (Zona) qry.getSingleResult();

qry = em.createQuery("select z from Zona z");
List<Zona> list = qry.getResultList();
for(Zona z: list){
     System.out.println("Zona:" + z.toString());
}
```

En el caso de necesitar pasar parámetros a nuestra consulta, se podrá hacer hacer con dos tipos de sintaxis: posicional o nominal. En el primer caso, se antepondrá el carácter ? a un número que representa un identificador numérico del parámetro. Mientras que en el segundo caso, se antepondrá el carácter : seguido de un nombre.

Para borrar entidades de una tabla lo podremos hacer de dos formas: invocando al método remove del entity manager o creando una query con una sentencia DELETE.

Opción 1 remove de una única entidad buscándola con find:

```
Zona zona = em.find(Zona.class, "Europa");
em.getTransaction().begin();
em.remove(zona);
e.getTransaction().commit();
if(em.find(Zona.class, "Europa")==null){
System.out.println("Eliminación correcta");
} else {
System.out.println("No se ha eliminado la zona");
}
```

Opción 1 remove de múltiples entidades buscándolas con una Query:

```
Query qry = em.createQuery("select z from Zona z");
List listZona = qry.getResultList();
em.getTransaction().begin();
for(Zona v: listZona){
em.remove(v); }
em.getTransaction().commit();
```

Para ejecutar instrucciones NO SELECT, se creará igualmente una Query q con las mismas 3 posibilidades anteriores, pero la instrucción, en lugar de ser una SELECT, será una instrucción:

- DELETE/UPDATE para eliminar/modificar instancias de una clase, en caso de Query o NamedQuery gestionadas por JPA.
- Cualquier instrucción SQL NO SELECT en caso de NativeQuery gestionada directamente por el SGBD vía JDBC

Y la diferencia está en cómo ejecutarla. En lugar de usar los métodos q.getResultList o q.getSingleResult, usaremos q.executeUpdate() que devuelve:

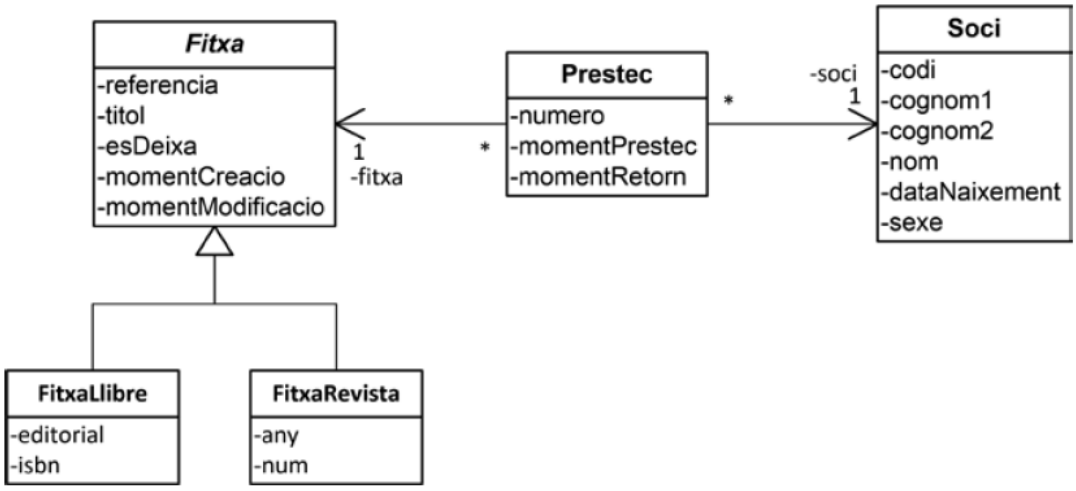
- Si es una Query o NamedQuery de JPA, el número de instancias eliminadas/modificadas.
- Si es una instrucción SQL vía NativeQuery, el número de filas afectadas o cero si es instrucción DDL-DCL (como en JDBC)

En cualquier caso (Query, NamedQuery o NativeQuery), es necesario tener una transacción abierta y efectuar commit para validar los cambios o rollback para deshacerlos.

Eso sí, recordad que una instrucción SQL-DDL provoca un commit automático en la BD e inicio de nueva transacción. Por tanto, si se ejecuta una SQL-DDL vía NativeQuery, se habrá producido un commit automático, por lo que es mejor no ejecutar instrucciones SQL-DDL desde los programas.

EJERCICIO 1

Crea el código fuente en Java que se corresponda con el siguiente diagrama.



WEBGRAFÍA

https://datos.codeandcoke.com/apuntes:hibernate