UNIDAD 3. PERSISTENCIA DE DATOS CON JPA E HIBERNATE

Contenido

1. Object-Relational Mapping (ORM)

El mapeo objeto-relacional, conocido por sus siglas en inglés como ORM (*Object-Relational Mapping*), es una técnica que permite la interacción entre bases de datos relacionales y lenguajes de programación orientados a objetos. En esencia, un ORM facilita la comunicación entre el mundo de los objetos en la programación y las tablas de una base de datos relacional.

En el paradigma orientado a objetos, los datos y la lógica del programa se organizan en clases y objetos. Sin embargo, las bases de datos relacionales almacenan la información en tablas con relaciones entre ellas. Aquí es donde entra en juego el ORM, actuando como un puente que traduce entre estos dos mundos.

Las ventajas del ORM:

- **Abstracción de la base de datos**: ORM proporciona una capa de abstracción que permite a los desarrolladores trabajar con objetos en lugar de consultar directamente la base de datos. Esto simplifica el código y facilita el desarrollo al ocultar la complejidad del modelo de datos subyacente.
- **Portabilidad del código**: Al utilizar ORM, el código de la aplicación se vuelve más independiente del sistema de gestión de bases de datos subyacente. Esto facilita la migración de una base de datos a otra sin tener que reescribir gran parte del código.
- **Productividad mejorada**: ORM simplifica las operaciones CRUD (Crear, Leer, Actualizar, Eliminar), reduciendo la cantidad de código necesario para interactuar con la base de datos. Esto permite a los desarrolladores centrarse más en la lógica de la aplicación en lugar de preocuparse por detalles de bajo nivel.
- **Mantenibilidad**: El código generado por un ORM tiende a ser más mantenible y escalable, ya que sigue patrones de diseño comunes y buenas prácticas. Además, los cambios en el modelo de datos pueden reflejarse de manera más sencilla en el código de la aplicación.
- **Reducción de errores**: Al utilizar ORM, se minimiza la posibilidad de errores relacionados con consultas SQL mal construidas o problemas de mapeo entre objetos y tablas. Esto contribuye a un desarrollo más robusto y a la prevención de vulnerabilidades de seguridad.

2. Especificación JPA

La Java Persistence API (JPA) es una especificación de Java que proporciona un estándar para el mapeo objeto-relacional (ORM). Surgió con el objetivo de simplificar y unificar el desarrollo de aplicaciones Java que interactúan con bases de datos relacionales. JPA permite a los desarrolladores trabajar con objetos Java en lugar de consultas SQL directas, facilitando así la persistencia de datos de manera más natural y orientada a objetos.

Características principales:

- **Mapeo ORM.** JPA proporciona anotaciones y configuraciones XML para establecer la correspondencia entre las clases de Java y las tablas de la base de datos. Esto permite que los objetos Java se almacenen y recuperen de manera transparente en una base de datos relacional, simplificando el desarrollo.

- **Consulta de objetos**. JPA introduce el lenguaje de consulta JPQL (*Java Persistence Query Language*), que es similar a SQL, pero se centra en objetos en lugar de en tablas. Esto facilita la escritura de consultas orientadas a objetos que son independientes del proveedor de persistencia subyacente.
- **Gestión de Transacciones**: JPA gestiona automáticamente las transacciones, simplificando el manejo de la concurrencia y garantizando la integridad de los datos en operaciones CRUD (Crear, Leer, Actualizar, Eliminar).
- **Eventos y Ciclo de Vida de Objetos**. JPA define un ciclo de vida claro para los objetos persistentes, permitiendo a los desarrolladores gestionar eventos como la creación, modificación o eliminación de entidades.

2.1. Configuración del fichero persistence.xml y dependencias

El fichero persistence.xml debe estar en la ruta: ***src/main/resources/META-INF/persistence.xml***

Añadir las clases a persistir y la configuración de la conexión a la BD (driver, URL, user, password, lenguaje de BD y acción de la BD). Ejemplo:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="https://jakarta.ee/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence

https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd"
             version="3.0">
<persistence-unit name="default">
    <class>org.example.modelo.Cliente</class>
    <class>org.example.modelo.Coche</class>
    <properties>
        <property name="jakarta.persistence.jdbc.driver"
value="com.mysql.cj.jdbc.Driver"/>
        <property name="jakarta.persistence.jdbc.url"
value="jdbc:mysql://localhost:3306/concesionario-jpa"/>
        <property name="jakarta.persistence.jdbc.user" value="root"/>
        <property name="jakarta.persistence.jdbc.password"
value="admin"/>
        <property name="hibernate.dialect"
value="org.hibernate.dialect.MySQLDialect" />
        <property name="jakarta.persistence.schema-
generation.database.action" value="drop-and-create"/>
    </properties>
</persistence-unit>
</persistence>
```

Importante:

- Si la acción de la BD es "**create**", solo se creará una vez la BD, mientras que "**drop-and-create**", lo que hace es borrar la BD y volver a crearla cada vez que lo ejecutamos. Esto último es muy útil si estamos haciendo modificaciones DDL, es decir, en los tipos de datos, añadiendo-modificando atributos de las clases, etc.…
- El **orden de las clases que se añaden en el fichero influye en cómo creará Hibernate las tablas**, es decir, que se debe seguir el mismo orden en que nosotros ejecutaríamos las sentencias CREATE.
- El **nombre de persistence-unit** ("en nuestro caso **default**") **debe coincidir** posteriormente cuando se **construya el EntityManagerFactory**.

Las dependencias Maven incluyen las librerías JDBC, Hibernate y JPA:

```
<!-- JDBC MySQL Connector -->
<dependency>
    <groupId>com.mysql</groupId>
    <artifactId>mysql-connector-j</artifactId>
    <version>8.4.0</version>
</dependency>
<!-- Hibernate Core -->
<dependency>
    <groupId>org.hibernate.orm</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>6.0.2.Final</version>
</dependency>
<!-- Jakarta Persistence API -->
<dependency>
    <groupId>jakarta.persistence</groupId>
    <artifactId>jakarta.persistence-api</artifactId>
    <version>3.1.0</version>
</dependency>
```

Si se desea visualizar las sentencias SQL que se están lanzando internamente podemos añadir estas 2 líneas al persistence.xml:

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

2.2. Anotaciones JPA para tablas

Las anotaciones en Java son una característica fundamental que permite agregar metadatos, para poder marcar o etiquetar el código fuente con información adicional que puede ser procesada por herramientas externas. Estas pueden ser predefinidas por el lenguaje (como *@Override* o *@Deprecated*), o provenir de bibliotecas de terceros, como en este caso de JPA.

Las anotaciones más importantes de JPA para la creación de tablas son:

- **@Entity**. Indica que la clase está asociada con una tabla de la base de datos.
- **@Table** (opcional). Permite especificar detalles sobre la tabla asociada a la entidad, como el nombre de la tabla, el esquema, etc.
- **@Id**. Marca el campo que actúa como Primary Key de la entidad.
- **@GeneratedValue**. Se generará automáticamente el valor de la clave primaria, la cual será única a nivel de base datos. Hay varias estrategias para generar la Primary Key en JPA:
  - **IDENTITY**. Son auto-incrementados y equivaldría a la opción de "AUTO_INCREMENT" de MySQL.
  - **SEQUENCE**. Utiliza "sequences" (objetos de la BD) que genera valores únicos de manera incremental para dicha secuencia.
  - **AUTO (por defecto)**. Utiliza la estrategia más adecuada de acuerdo con la base de datos subyacente.

```java
@Entity
@Table(name = "nombre_tabla_personalizado")
public class Entidad {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO) //por defecto
    private long id;
}
```

Ejemplo de clase JPA con estrategia de generación de PK automática.

IMPORTANTE. Para que la persistencia se realice correctamente, siguiendo las especificaciones JPA, debe haber un **constructor por defecto (vacío) y getter/setter de los atributos**.

2.3 Anotaciones JPA para las columnas

2.3.1. Propiedades de columnas

@Column

Permite cambiar algunas propiedades de la columna:

- Nullable. Permite almacenar nulos
- Name. Cambiar el nombre de la columna (por defecto coge el nombre del atributo).
- Length. Tamaño del campo.
- Unique. Estable la columna como Unique.
- Insertable, updatable. Permite que la entidad sea insertable o actualizable.

Ejemplo para establecer el atributo teléfono como Unique en SQL:

```java
@Column(unique = true)
private String telefono;
```

2.3.2. Mapeo de fechas

Para mapear las fechas, hay que elegir si se desea almacenar únicamente la fecha:

```java
@Temporal(TemporalType.DATE)
private Date registDate;
```

O si se desea almacenar la fecha y la hora (es la que utiliza por defecto si no se especifica):

```java
@Temporal(TemporalType.TIMESTAMP)
private Date registDate;
```

2.3.3. Relaciones con entidades

Asociaciones Muchos a uno (ManyToOne)

Para establecer una relación con una lista de entidades de forma unidireccional, lo haríamos como en una BD: estableciendo una FK desde la entidad de “muchos” a la de “uno”. En JPA se hace estableciendo una ManyToOne:

```
@ManyToOne
private Trabajador propietario;
```

En la tabla Coches se crea la columna “propietario” con una restricción FK hacia la PK de la tabla Trabajadores. Esto solo tiene sentido si no queremos tener la lista de coches en la tabla Trabajadores.

Asociaciones Uno a Muchos (OneToMany)

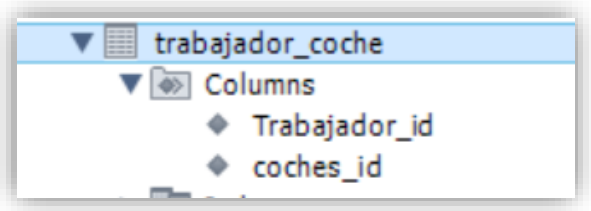
Relaciones unidireccionales

Únicamente se puede navegar del 1 al muchos unidireccional

En la tabla Trabajadores establecemos OneToMany hacia la tabla Coches

```
@OneToMany
private List<Coche> coches;
```

Como no hay ManyToOne del muchos al uno, la forma de implementarlo en SQL es creando una tabla intermedia, con FK hacia la tabla Trabajador y Coche:



La razón por la cual Hibernate crea esta tabla intermedia es porque, en una relación unidireccional, la entidad Coche no tiene conocimiento de la entidad Trabajador. Sin esa información, Hibernate necesita una tabla adicional para almacenar las relaciones.

Relaciones bidireccionales

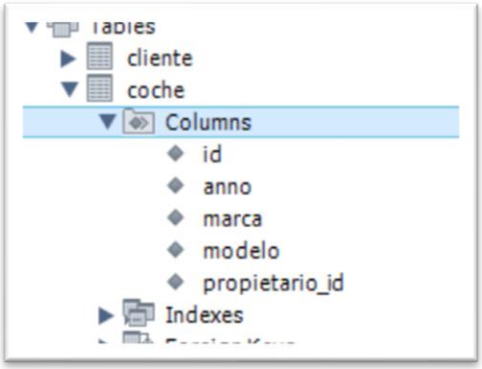
Se debe especificar la relación del 1 a muchos en la clase Trabajador:

```
@OneToMany(mappedBy = "propietario")
List<Coche> coches;
```

Y la relación muchos a uno en la clase Coche:

```
@ManyToOne
private Trabajador propietario;
```

Esta **estrategia es la más correcta** si no hay requisito de ocultar la navegación hacia alguna dirección, ya que genera las tablas SQL de forma **eficiente** (con una FK de la relación muchos hacia la del uno):



IMPORTANTE: Al ser bidireccional, no solo hay que añadir el coche a la lista de coches de Trabajador, sino también habrá que añadir el propietario en la ManyToOne de Coches. Para realizar ambas operaciones, se recomienda hacer métodos *helper* que hagan toda la gestión en la clase Trabajador:

```
public void addCoche(Coche coche){
    coches.add(coche);
    coche.setPropietario(this);
}
public void removeCoche(Coche coche){
    coches.remove(coche);
    coche.setPropietario(null);
}
```

Nota: En el método removeCoche establecer el propietario a null sería innecesario si tenemos orphanRemoval a true porque se borraría el coche de la tabla Coches al no ser referenciado por ningún propietario.

Anotaciones Cascade y orphanRemoval

Cuando en SQL, queríamos persistir un objeto que contuviera otro objeto (Many To One), teníamos que hacer otro INSERT, esa inserción en cascada se especifica de esta manera:

```
@ManyToOne(cascade = CascadeType.PERSIST)
```

Y si queremos que cuando se persista un objeto, se persista la lista de objetos relacionada, se especificaría de la misma manera:

```
@OneToMany(mappedBy = "propietario", cascade = CascadeType.PERSIST)
```

Por otro lado, cuando en SQL creábamos una relación OneToMany, podíamos especificar en la FK si la actualización y el borrado serían en Cascade, No action…En JPA habrá que especificarlo también, dando lugar a los siguientes tipos de acciones en cascada:

- **CascadeType.PERSIST**. Persiste las entidades relacionadas cuando se persiste la entidad.
- **CascadeType.MERGE**. Realiza un merge de las entidades relacionadas cuando se hace un merge en la entidad principal.
- **CascadeType.REMOVE**. Elimina las entidades relacionadas cuando se elimina la entidad principal (equivalente en SQL a ON DELETE CASCADE). Cabe destacar que el *Delete No action* sería el valor por defecto.

- **CascadeType.REFRES.** Cuando se ejecuta refresh de la entidad padre, se actualizan tanto el padre como las entidades relacionadas.
- **CascadeType.DETACH**: Desasocia las entidades relacionadas cuando se hace un detach de la entidad principal.

Ejemplo que implicaría borrado/inserción en cascada y merge:

```
@OneToMany(mappedBy = "propietario", cascade = {CascadeType.PERSIST,
CascadeType.REMOVE, CascadeType.MERGE})
private List<Coche> coches;
```

La propiedad **orphanRemoval** en la relación @OneToMany o @OneToOne en JPA especifica si las entidades secundarias (hijas) deben ser eliminadas de la base de datos cuando se eliminan de la colección de entidades primarias (padres). En nuestro ejemplo,

```
@OneToMany(mappedBy = "propietario", cascade = CascadeType.PERSIST,
orphanRemoval = true)
private List<Coche> coches;
```

Si eliminas un Coche de la lista de coches de un Trabajador, Hibernate también eliminará ese Coche de la base de datos si no está siendo referenciado por otro Trabajador. Dicho de otra manera, evitará que esa fila Coche de la tabla se quede "huérfana", pues no hay ni un trabajador que la referencie.

Asociaciones Uno a Uno (OneToOne)

La anotación OneToOne se utiliza para establecer una relación Uno a Uno. En otras palabras, una instancia de una entidad está relacionada con exactamente una instancia de otra entidad, y viceversa.

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

    private String nombre;

    @OneToOne(mappedBy = "persona")
    private DetallesPersona detalles;
}
@Entity
public class DetallesPersona {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @OneToOne
    private Persona persona;
}
```

Habría que decidir cuál de las dos tablas va a ser la que tenga la FK hacia la otra tabla. Será aquella tabla, donde tengamos establecido el mapeo. En el ejemplo mostrado, la tabla DetallesPersona es la que tendrá una columna "persona", que será FK hacia la tabla Persona.

Asociaciones Muchos a Muchos (ManyToMany)

La anotación ManyToMany sirve para establecer una relación muchos a muchos, por ejemplo, un profesor tiene una lista de estudiantes y cada estudiante tiene una lista de profesores.

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

    @ManyToMany(cascade = { CascadeType.PERSIST, CascadeType.MERGE })
    private List<Profesor> profesores = new ArrayList<>();
}

@Entity
public class Profesor {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToMany(mappedBy = "profesores")
    private List<Estudiante> estudiantes = new ArrayList<>();

}
```

Al igual que con el OneToMany bidireccional, hay que mapear en una de las dos tablas. Y también se recomienda utilizar métodos helpers para añadir y eliminar elementos.

- **CascadeType.PERSIST**: Cuando persistes una entidad (Estudiante o Profesor), las entidades relacionadas también se persistirán automáticamente. Si creas un nuevo Estudiante y lo persistes, los profesores en los que está inscrito también se persistirán en la base de datos.
- **CascadeType.MERGE**: Supongamos que un Estudiante está inscrito en varios profesores y quieres vaciar la lista de profesores y añadir un profe nuevo. Todos esos cambios se fusionarían (merge) si está habilitada esta opción.

Personalización de FK

Se puede especificar el nombre de la columna FK (*name*) y el nombre de la restricción (*foreignKey*):

```
@ManyToOne
@JoinColumn(name="trabajador_id",
            foreignKey=@ForeignKey(name="TRABAJADOR_ID_FK"))
private Trabajador propietario;
```

Anotaciones lazy y eager

Cuando se utiliza la **anotación Lazy** (*perezosa*), se establece la carga perezosa para la relación. La carga perezosa significa que los objetos relacionados no se cargarán automáticamente cuando se recupera el objeto principal de la base de datos. En su lugar, se cargarán solo cuando se acceda explícitamente a ellos, en este ejemplo, solo cuando se invoque el método getCoches() se traería los coches de la BD.

```
@OneToMany(mappedBy = "propietario", fetch = FetchType.LAZY ,cascade =
CascadeType.ALL, orphanRemoval = true)
private List<Coche> coches;
```

¡DANGER! Si usamos Lazy, el problema es que solo cargará la lista de coches si se hace uso de los coches dentro de la transacción, por tanto si nuestro dao devuelve el cliente, lo devolverá sin los coches y crasheará al intentar acceder a ellos. Si queremos que una colección de tipo lazy esté cargada hay que usar el método initialize:

```
@Override
public Cliente findByIdWithCars(int id) {
    EntityManager entityManager =
entityManagerFactory.createEntityManager();
    Cliente cliente = entityManager.find(Cliente.class, id);
    Hibernate.initialize(cliente.getCoches());
    entityManager.close();
    return cliente;
}
```

```
@OneToMany(mappedBy = "propietario", fetch = FetchType.EAGER ,cascade
= CascadeType.ALL, orphanRemoval = true)
private List<Coche> coches;
```

Por otro lado, cuando se estable la **anotación Eager** (*carga ansiosa*) significa que los objetos relacionados se cargarán automáticamente junto con el objeto principal. En otras palabras, se recuperarán de la base de datos en el mismo momento que se recupera el objeto principal. Por tanto, en la lista de trabajadores en este ejemplo, tendría cada trabajador la lista de coches automáticamente.

2.3.4. Mapeo de colecciones de tipos primitivos

Se utiliza @ElementCollection para mapear colecciones de tipos primitivos (Integer, String, etc.) sin necesidad de crear una tabla adicional.

```
@Entity
public class Persona {

    @ElementCollection
    @CollectionTable(name = "telefonos_personas")
    private List<String> telefonos = new ArrayList<>();

}
```

- **@ElementCollection** se aplica a la propiedad telefonos, indicando que se trata de una colección de elementos primitivos.

- **@CollectionTable** (opcional) permite especificar el nombre de la tabla intermedia que se creará para almacenar la colección. En este caso, se crea una tabla llamada " telefonos_personas".

Con esta configuración, cuando se persiste una instancia de Persona, Hibernate creará automáticamente la tabla "telefonos" y almacenará los números de teléfono en esa tabla, asociados a la entidad Persona.

Si no se quiere especificar el nombre de la tabla intermedia ni su columna se pueden obviar esas anotaciones:

```
@ElementCollection
private List<String> telefonos = new ArrayList<>();
```

2.4. Anotaciones JPA para la persistencia de jerarquías

Ya vimos en la Unidad anterior que para persistir una jerarquía había 3 estrategias diferentes: “tabla única”, “tabla por subclase” y “tabla por clase concreta”. Hibernate implementa las 3 estrategias especificándolo en la clase padre:

Tabla única (“Single table”)

Solo hay una única tabla, ya que los atributos de los hijos suben al padre. Para discernir entre una subclase u otra se utiliza un atributo “dtype”, que indica qué subclase está almacenando. Es la **estrategia por defecto** de Hibernate, si no especificamos ninguna estrategia de forma explícita.

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
public class MyProduct {
    //..

@Entity
public class Book extends MyProduct {
    //..
}
```

Por defecto la columna es un String y se llama “dtype” y contendrá el nombre que hayamos especificado a la entidad. Sin embargo, se puede cambiar para que en lugar de “dtype” se llame de otra manera e incluso sea otro tipo de dato:

```
@Entity(name="products")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="product_type",
  discriminatorType = DiscriminatorType.INTEGER)
public class MyProduct {
    // ...
}
```

En este ejemplo, lo hemos renombrado a “producto_type” y el tipo de dato es un Integer.

Si se hace de esta manera hay que especificar en las subclases, qué Integer va a contener cada subclase:

```
@Entity
@DiscriminatorValue("1")
public class Book extends MyProduct {
    // ...
}

@Entity
@DiscriminatorValue("2")
public class Pen extends MyProduct {
    // ...
}
```

Tabla por subclase ("Joined table")

Para cada clase existe una tabla en la base de datos. Consiste en que la clase Padre y las clases hijas tienen su tabla correspondiente. Para poder implementarlo, las clases hijas tienen como PK, una FK hacia la PK de la clase padre.

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public class Animal {
    //..

@Entity
public class Pet extends Animal {
    //..
}
```

Esto implica que cada vez que se haga una consulta requerirá hacer un JOIN en SQL, lo cual implica una pérdida de rendimiento.

Tabla por clase concreta ("Mapped superclass")

Consiste en que las subclases contienen los atributos de la clase padre, por tanto, habrá tantas tablas como subclases, es decir, se implementa eliminando la tabla padre y añadiendo sus atributos a las tablas hijas.

```
@MappedSuperclass
public class Person {
    //...

@Entity
public class MyEmployee extends Person {
    //..
}
```

Es importante notar que la clase padre “Persona” no tiene la anotación @Entity, ya que no se va a persistir en la BD.

2.5. Ciclo de vida de una entidad en JPA

En JPA, las entidades pasan por varios estados durante su ciclo de vida. Estos estados incluyen:

Estado 1. *Transient*. Se ha instanciado el objeto en Java pero no está gestionado por el EntityManager (contexto de persistencia), por tanto, no está persistido en la BD. Ej:

```
Cliente cliente = new Cliente("11111Z", "Jose", "Gomez",
"+34123123123");
```

Estado 2. *Managed*. La entidad está siendo gestionada por el EntityManager. Esto significa que Hibernate detectará cualquier cambio en la entidad y generará las sentencias INSERT/UPDATE correspondientes. Hay diferentes maneras de conseguir que un objeto en estado managed:

- Persistiéndolo: em.persist(cliente)
- Leyendo el objeto: em.find(Cliente.class, 1)
- Merge. Puedes hacer merge de una entidad detached para actualizarlo. em.merge(cliente)

Estado 3. *Detached*. La entidad estaba gestionada por el EntityManager, pero ha sido desconectada (por ejemplo, cuando el EntityManager hace detach() o se cierra).

Por regla general, **no se recomienda hacer detach() de un objeto**.

Ejemplo de ciclo de vida:

```
entityTransaction.begin();
// Se crea el cliente en estado Transient
Cliente cliente = new Cliente("12345X", "Juan", "Pérez",
"+34123456789");
entityManager.persist(cliente); //El cliente pasa a managed al
persistirlo
entityTransaction.commit();
// Tras el commit, el cliente está en estado detached

// Modificamos el coche fuera del contexto de persistencia
coche.setModelo("Yaris");

// Intentamos sincronizar nuevamente la entidad con la base de datos
entityTransaction.begin();
entityManager.merge(cliente);  // El cliente pasa a estado attached
entityTransaction.commit();
```

Si no hiciéramos merge, los cambios (setModelo(“Yaris”)), no se sincronizarían con la BD.

Eliminado (Removed): Cuando se utiliza el método remove, no se borra de la BD, sino que pasa a estado “Removed” y cuando se haga commit se llevará a cabo.

Ejemplo de error por culpa de tener un objeto en estado “detached”:

En el Dao de Coche, la persistencia, en principio debería funcionar persistiendo el coche simplemente:

```
@Override
public void save(Coche coche) {
    EntityManager entityManager =
entityManagerFactory.createEntityManager();
    entityManager.getTransaction().begin();
    entityManager.persist(coche);
    entityManager.getTransaction().commit();
    entityManager.close();
}
```

Sin embargo, este error nos dará error cuando el coche contenga un objeto cliente (propietario) ya persistido previamente. El error que dará es este:

```
jakarta.persistence.PersistenceException: Converting
org.hibernate.PersistentObjectException to JPA PersistenceException :
detached entity passed to persist: org.example.modelo.Cliente
```

Lo que está ocurriendo es que el objeto cliente que ya se persistió previamente no está en estado managed, es decir, no está en el contexto de persistencia. Para solucionarlo, antes de hacer commit, hay que meterlo en el contexto de persistencia mediante el método merge:

```
@Override
public void save(Coche coche) {
    EntityManager entityManager =
entityManagerFactory.createEntityManager();
    entityManager.getTransaction().begin();
    if (coche.getPropietario()!=null){

coche.setPropietario(entityManager.merge(coche.getPropietario()));
    }
    entityManager.persist(coche);
    entityManager.getTransaction().commit();
    entityManager.close();
}
```

3. Implementación Hibernate

Si JPA tenía como objetivo proporcionar un conjunto estándar de interfaces y anotaciones para mapear objetos Java a tablas en bases de datos, Hibernate se va a encargar de implementar todas esas interfaces y anotaciones. Por tanto, Hibernate es el framework que se encarga de hacer realidad ese estándar teórico que define JPA. Otra implementación de JPA es EclipseLink.

3.1. EntityManager: operaciones CRUD

```
EntityManagerFactory entityManagerFactory =
Persistence.createEntityManagerFactory("default");
EntityManager entityManager = entityManagerFactory.createEntityManager();
EntityTransaction entityTransaction = entityManager.getTransaction();
entityTransaction.begin();
entityManager.persist(objeto); //INSERT
entityManager.merge(objeto); //UPDATE
entityManager.find(Clase.class, objeto.getId()); //SELECT
entityManager.remove(objeto); //DELETE
entityTransaction.commit();
```

```
entityManager.close();
entityManagerFactory.close();
```

Características:

- **Excepciones**. Las excepciones que lanza son no comprobadas, por lo que no hay obligación de declararlas.
- **Transaccionalidad**. En JPA la transacción se gestiona automáticamente, lanzando el rollback si se lanza una excepción Runtime o de Perisistencia.

Ejemplo de método save del dao aplicando JPA:

```
@Override
public void save(Cliente cliente) {
    EntityManager entityManager =
entityManagerFactory.createEntityManager();
    entityManager.getTransaction().begin();
    entityManager.persist(cliente);
    entityManager.getTransaction().commit();
    entityManager.close();
}
```

Ejemplo de método findById del Dao:

```
@Override
public Cliente findById(int id) {
    EntityManager entityManager =
entityManagerFactory.createEntityManager();
    Cliente cliente = entityManager.find(Cliente.class, id);
    entityManager.close();
    return cliente;
}
```

Nota: no hace falta iniciar una transacción para realizar una consulta.

3.2. Lenguaje JPQL ("Java Persistence Query Language")

Se trata de un lenguaje similar a SQL, pero diseñado para trabajar con entidades JPA, y soportando, además, el paradigma de la Programación Prientado a Objetos.

Documentación oficial de JPQL

3.2.1. Sintaxis básica

Partimos de la consulta más simple: leer de BD todos los trabajadores que se llamen Juan:

```
// Consulta JPQL:Seleccionar todos los clientes que se llamen Juan
List<Cliente> clientes = entityManager.createQuery(
"SELECT c FROM Cliente c WHERE c.nombre = :nombre", Cliente.class)
   .setParameter("nombre", "Juan")
   .getResultList();
```

- **SELECT** c: Indica que queremos seleccionar objetos de tipo Trabajador (alias t). El alias es obligatorio.
- **FROM** cliente c: Especifica que estamos seleccionando objetos de la clase de entidad Cliente y les asignamos el alias c para facilitar la referencia en la consulta.

- **WHERE** c.nombre =:nombre. Aplica una condición, donde :nombre es un parámetro nombrado, es decir, su valor se asignará posteriormente con el método setParameter.
- getResultList. Le pasamos en createQuery la **clase** a la que queremos que haga el **mapeo** (Cliente.class) para que getResultList devuelva una lista de dicha clase.
- **DANGER**: A diferencia del find, el método **createQuery lanza una excepción** en caso de no encontrar ninguna fila.

Ejemplo, teniendo en cuenta que se lanza una excepción y que hay que cerrar el entityManager:

```
public List<Cliente> findByNombre(String nombre) {
    EntityManager entityManager =
entityManagerFactory.createEntityManager();
    List<Cliente> clientes = new ArrayList<>();
    try {
        clientes = entityManager.createQuery(
                        "SELECT c FROM Cliente c WHERE c.nombre =
:nombre", Cliente.class)
                .setParameter("nombre", nombre)
                .getResultList();
    } catch (Exception e) {
        return clientes;
    } finally {
        entityManager.close();
    }
    return clientes;
}
```

Pertenencia a conjuntos (IN)

Esta consulta busca coches que pertenezcan a una lista de matrículas:

```
List<Coche> coches = entityManager.createQuery(
        "SELECT co FROM Coche co WHERE co.matricula IN :matriculas",
Coche.class)
    .setParameter("matriculas", List.of("1123HAR", "999HAR"))
    .getResultList();
```

3.2.2. Consultas JOIN

A diferencia de SQL, aquí podemos hacer uso los atributos del objeto, sin tener que lidiar con las FK.

Ejemplo. Consultar clientes con coches de un modelo específico

```
List<Cliente> clientes = entityManager.createQuery(
        "SELECT c FROM Cliente c JOIN c.coches co WHERE co.modelo =
:modelo", Cliente.class)
    .setParameter("modelo", "Fiesta")
    .getResultList();
```

- Realiza un Join entre la clase Cliente (“c”) y la clase Coche accediendo a su lista de coches (“co”).
- El alias “c” referencia a cliente y el alias “co” a coche.
- Se filtra el modelo con un parámetro nombrado (“:nombre”).

3.2.3. Consultas de agregación

COUNT

Podemos usar la función COUNT, como en SQL, pero con la ventaja de que podemos navegar al campo que necesitamos (nombre) como si fuera una variable de Java, sin necesidad de hacer JOIN:

```
// Ejemplo 3 de consulta para contar el número de coches que tienen
los que se llaman Juan
Long count = entityManager.createQuery("SELECT COUNT(co) FROM Coche co
WHERE co.propietario.nombre = :nombre", Long.class)

    .setParameter("nombre", "Juan")
    .getSingleResult();
}
```

COUNT Y GROUP BY:

Si quisiéramos la cuenta de coches agrupada por nombres (Group By) se haría así;

```
List<Object[]> resultados = entityManager.createQuery(
    "SELECT co.propietario.nombre, COUNT(co) FROM Coche co GROUP BY
co.propietario.nombre", Object[].class)
    .getResultList();

for (Object[] resultado : resultados) {
    System.out.println("Propietario: " + resultado[0] + ", Número de
coches: " + resultado[1]);
}
```