PROGRAMACIÓN ORIENTADA A OBJETOS CON ACCESO A BBDD

IMPLEMENTACIÓN MEDIANTE ORM

MIEMBROS

Katiane Coutinho Rosa
Nadia Igual Bravo
Xavier Melo Pardo
Daniel Boj

Grupo - Cirice

Producto 4. Implementación mediante ORM

Enlace a GitHub

Para realizar el seguimiento de las tareas que vamos realizando, tanto a nivel individual como grupal, compartimos el enlace a nuestro repositorio de GitHub:
https://github.com/DanielBoj/UOC-POO-con-acceso-a-BBDD

Creación de la BD con docker-compose

Para facilitar la prueba de la aplicación en local, hemos creado un archivo *docker-compose* relacionado con un *Dockerfile* adjuntados a nuestra solución. De este modo, en el directorio raíz vienen incluidos los archivos necesarios para montar rápidamente la BD en un contenedor Docker y el ejecutable punto JAR ya viene listo para conectarse a esta.

Para evitar colisiones de puertos, el contenedor muestra el puerto 3307 en lugar del 3306. Por lo tanto, las variables de entorno del JAR escuchan a este puerto. Si se prefiere generar una base de datos en un servidor local, hay que cambiar el puerto en el archivo *.env*. Hay dos archivos a modificar, uno en el directorio raíz y otro en el directorio *target*.

Comando Docker – Hay que ejecutarlo en el directorio raíz de la solución, donde están situados los dos archivos de referencia:
$ docker-compose up --build -d

ORM

Responde a las siglas de ***Object Relational Mappings*** y es un modelo de programación en el que se facilita el trabajo con BBDD manteniendo el paradigma de POO mediante el mapeo de las estructuras de las BBDD, resultando muy útil en las de tipo relacional. De esta manera, se simplifica y agiliza el desarrollo de la aplicación.

ORM establece un vínculo entre las estructuras de la BD y las entidades lógicas que usa el programa, convirtiéndose en el intermediario para ejecutar las acciones CRUD sobre la BD de forma indirecta. De este modo, el mapeo de los objetos evita que haya que escribir todas las consultaS SQL y la lógica de gestión de persistencia de forma manual en el código.

Así, los **frameworks** ORM permiten simplificar el código, aumentando su legibilidad y facilitando su escalabilidad y mantenimiento. Este hecho hace que su uso sea muy habitual en la programación moderna, en realidad son un requerimiento de casi todas las ofertas de empleo, y se utilizan en una gran variedad de lenguajes de programación cómo Java, Python, Ruby, C# o JavaScript. En el caso particular de Java, las opciones más populares son JPA, JDO e Hibernate.

De este modo, un ORM permite que interactuemos con la BD mediante nuestros Objetos de Java sin tener que realizar modificaciones o adaptaciones manuales en nuestro código y sin tener que codificar las sentencias SQL de forma manual. Otra de las ventajas de usar este paradigma, es que simplifica mucho la ejecución de las acciones CRUD sobre la BD.

Es decir, la interacción con la BD se realiza mediante los métodos y clases que nos ofrecen las interfaces y APIS del ORM sin tener que escribir ninguna sentencia SQL en nuestro código para manejar las acciones CRUD.

En conclusión, el uso de APIs ORM es muy común en la programación actual, es una habilidad técnica muy demandada y nos ayuda a simplificar el código y mejorar su escalabilidad y legibilidad.

Java dispone de varias herramientas ORM, siendo las más usada JPA, JDO e Hibernate. Cada uno tiene sus particularidades que se presentan a continuación:

JPA (Java Persistence API)

Es el ***framework*** nativo de Java, no siendo una implementación persé, sino un grupo de interfaces que deben ser implementadas en el código. Es decir, en nuestro código usaremos las interfaces de JPA que luego tendrán que ser implementadas por la librería que escojamos. Se desarrolló para simplificar la creación de aplicaciones empresariales con Java EE.

Como hemos comentado, es únicamente una especificación, una serie de interfaces, siendo la base para implementaciones muy populares como EclipseLink e Hibernate. Además, dispone de características avanzadas:

- Consultas de criterios
- Cache de segundo nivel
- Soporte para múltiples proveedores de BBDD
- Flexibilidad en la configuración para distintos entornos de desarrollo

El funcionamiento de JPA se basa en:

- Un conjunto de anotaciones para añadir a nuestros objetos e indicar la forma en qué se realiza el mapeo con la BD → javax.persistence
- Un lenguaje de consultas, **Java Persistence Qury Language**, que se usa para acceder a las entidades de la BD.

JDO (Java Data Objects)

Es una API de persistencia de objetos de Java que permite el trabajo tanto con herramientas relaciones como con no-relacionales. Como en el caso anterior, JDO es únicamente una especificación, independiente de la librería que realice la implementación, con lo que puede usarse con distintos proveedores de BBDD y está implementado por diferentes librerías.

JDO es muy parecido a JPA en sus aspectos funcionales, pero presenta diferencias en la forma de usarse y configurarse. Igualmente, presenta una serie de características avanzadas muy interesantes:

- Soporte para objetos anidados
- Soporte para clases abstractas

Comparativa entre JPA y JDO

- JPA es una especificación de Java EE soportada por la mayoría de proveedores de BBDD. JDO está más enfocado en la persistencia de objetos y no está soportado por tantos proveedores.
- JDO tiene una mayor complejidad que JPA ya que tiene funcionalidades más avanzadas.
- JDO es más rápido que JPA y ofrece mayor flexibilidad en el manejo de los objetos.

Hibernate

Hibernate es un ***framework*** que especifica la implementación JPA y es uan de las herramientas ORM más populares, usadas y requeridas. Es un proyecto **Open Source** y, además, incluye funcionalidades no cubiertas por JPA:

- Soporte para consultas SQL nativas además del uso del lenguaje JPQL.
- Uso de tipos de datos personalizados.
- Configuración de propiedades de mapeo personalizadas.
- Integración con APIs de caché de segundo nivel.
- Mapeo de herencia de entidades y objetos anidados.

De este modo, es un ***framework*** muy flexible que proporciona una gran variedad de opciones de configuración y características avanzadas.

Hibernate es una de las implementaciones más populares de JPA y es una de las herramientas de ORM más utilizadas en Java. Hibernate es una herramienta de código abierto y se utiliza para mapear objetos Java a una base de datos relacional. Hibernate es muy flexible y proporciona muchas opciones de configuración para adaptarse a diferentes entornos de desarrollo. Hibernate también proporciona características avanzadas como caché de segundo nivel, consultas nativas y consultas de criterios.

MyBatis

MyBatis es un *framework* de persistencia de datos para Java que permite trabajar con bases de datos de manera más sencilla y eficiente. MyBatis utiliza archivos XML o anotaciones en Java para realizar el mapeo objeto-relacional (ORM) y manejar las consultas SQL. Ayudando a simplificar el código y mejorar su mantenimiento y escalabilidad.

MyBatis es una herramienta muy flexible y se puede utilizar con diferentes proveedores de bases de datos. Además, proporciona una API de consultas dinámicas que permite construir consultas

de base de datos de manera programática. Las consultas dinámicas son más fáciles de entender que las consultas SQL nativas y se pueden refactorizar más fácilmente.

EclipseLink

EclipseLink es una herramienta ***Open Source*** que implementa la API JPA que presenta una serie de características adicionales:

- Soporte para la persistencia avanzada.
- Acceso a datos sin conexión.
- Administración de la caché.

Ventajas de Hibernate

Hibernate es una herramienta de ORM muy popular en el mundo de Java debido a sus ventajas y características avanzadas que la hacen muy flexible y potente. A continuación, se detallan las ventajas de Hibernate sobre otras herramientas de ORM.

- Flexibilidad: Proporciona distintas opciones de configuración y personalización para diferentes entornos de desarrollo, adaptándose a las necesidades específicas de los desarrolladores. Permite definir, entre otras:
    - Estrategias de carga.
    - Asociaciones entre objetos.
    - Almacenamiento en caché.
- Caché de segundo nivel: Permite almacenar objetos persistentes en caché reduciendo el número de consultas a la BD, con la mejora que supone del rendimiento de la aplicación.
- Consultas nativas: Permite el uso de consultas escritas en SQL cuando sea necesario, además, proporciona una API nativa que facilita su escritura.
- Consultas de criterios: Hibernate proporciona un API de criterios que permite a los desarrolladores construir consultas de base de datos de manera programática.
- Popularidad: Como hemos comentado, Hibernate es seguramente la implementación más popular, usada y demandada de JPA.
- Open Source: Como hemos comentado, Hibernate es una herramienta de código abierto y descarga libre.

Por qué decidimos trabajar con Hibernate

Tras analizar las ventajas de este ***framework***, nos ha quedado claro de que es una de las mejores opciones para escoger un ORM. Además de todas las ventajas funcionales que presenta, nos hemos centrado en algunos aspectos que también consideramos muy positivos:

- Tras una búsqueda en plataformas de empleo, hemos podido comprobar que la mayor parte de ofertas para Java demandan Hibernate y Spring.
- Hibernate funciona muy bien junto a Jakarta EE, siendo un combo muy útil y una de las mejores opciones para profundizar en los conocimientos de Java.
- Hibernate usa y expande JPA, con lo que estaremos aprendiendo también el funcionamiento del ***framework*** nativo.

Características del uso de Hibernate

Hibernate nos permitirá realizar un mapeo de los objetos relacionales desde Java para facilitar la interacción entre nuestra app y la BD. Todas las consultas y operaciones en la BD se realizarán mediante estas entidades ORM que, en definitiva, son clases de Java, con lo que el modelo se amoldará a nuestro paradigma de POO, para ello, trabajaremos mediante ***anotaciones*** que se encargarán de mapear nuestras clases con las tablas correspondientes de la BD. Cada atributo se mapeará a un campo de la tabla, con ello, nos liberamos de las restricciones de JDBC y el código queda mucho más ligero y limpio. Todo esto, manteniendo nuestro modelo Repositorio/DAO.

Podemos trabajar con Hibernate de dos formas, mediante JPA, el API de persistencia de Java que ya hemos comentado, o mediante HNA, ***Hibernate Native API,*** el API nativo de Hibernate. Cada uno se basa en un objeto de trabajo distinto:

- JPA → EntityManager.
- HNA → Session.

Por detrás, el motor de Hibernate estará usando JDBC. Cada operación o consulta pasará por nuestro objetos **entity** e Hibernate sabrá qué consultas deberá realizar en la BD. Los datos se mapean a los objetos **entity** y se pasan a los objetos DAO.

Recordemos que JPA es únicamente una especificación, contiene los contratos abstractos, un conjunto de interfaces, que debe de implementar un proveedor, e Hibernate realiza la implementación de todos ellos. JPA es una capa de abstracción por sobre los proveedores, ello significa que podemos cambiar de proveedor fácilmente cambiando la configuración de nuestra app como ya vimos que podíamos hacer con los motores que usa JDBC.

Hibernate permite distintos tipos de consultas y operaciones:

- Hibernate Query Language → Proviene del JPA Query Language: Es un lenguaje de tipo consulta pero orientado a objetos, es decir, la consulta se realiza al objeto **entity** no a la tabla.
- Criteria API → Permite construir SQL de modo programático.
- SQL nativo → Consultas directamente a la tabla, pero Hibernate lo mapea directamente a la clase.

Como hemos avanzado, para trabajar con Hibernate tendremos que usar las anotaciones:

```
@Entity //Indica que es uan clase entity d eHibernate/JPA y se asigna a una tabla
@Table // Indica el nombre de la tabla a la que se asigna la clase entity
@Id // Indica que el atributo se corresponde con la columna PK de nuestra tabla.
@Column // PErmite indicar el nombre del campo en la tabla.
```

Si los atributos y las columnas tienen el mismo nombre, no es necesario indicar el nombre mediante anotaciones.

Las anotaciones también sirven para indicar la cardinalidad de las asociaciones entre las clases **entity**:

```
// Representan las relaciones mediante FK:
@ManyToMany
@OneToMany
@OneToOne
@ManyToMany
```

Uso de Jakarta EE

Además del uso de Hibernate, vamos a trabajar también con el **framework** Jakarta, la nueva implementación de Java EE, una opción muy recomendable y siendo también un requisito común en las ofertas de trabajo.

Implementaciones

Importar las librería en Maven

Comenzamos por añadir la dependencia de Hibernate a nuestro proyecto en nuestro archivo POM. Desde IntelliJ podemos acelerar el proceso colocándonos en la declaración de dependencias y usando ALT+INSERT y seleccionando 'Add dependency'.

A continuación, buscamos la dependencia por el ***artifact id***, escogeremos hibernate-core-jakarta ya que vamos a trabajar con Jakarta EE de base. Una vez localizada la librería, pulsamos 'add' e IntelliJ la añadirá automáticamente a nuestro archivo POM.

```
        <version>5.2.2</version>
    </dependency>
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-core-jakarta</artifactId>
        <version>5.6.15.Final</version>
    </dependency>
</dependencies>
```

Lo usamos a través de Jakarta, la nueva denominación para Java Enterprise Edition, porque es la forma más recomendada de hacerlo en los ámbitos profesionales.

Configuración de JPA

Para ello tenemos que configurar un archivo XML llamado **persistence.xml** que tenemos que guardad en el directorio ***resources/META-INF***. Es muy importante que esté todo en mayúsculas.

IntelliJ nos ayudará con el autcompletado de este archivo.

En la etiqueta ***persistence*** seleccionaremos la version 3.0 ya que es la de Hibernate Jakarta.

En este archivo es donde podemos realizar el cambio de los proveedores que queramos usar, en nuestro caso Hibernate, pero permite realizar fácilmente cambios de un proveedor a otro. Por otro lado, también definiremos las clases *entity* de nuestro programa, aunque se autoregistra, es mejor realizar una anotación manual. También incluiremos las propiedades de configuración para realizar el acceso a nuestra BD.

Es importante que tomemos nota del nombre que asignamos al *entity* ya que lo vamos a necesitar.

Directorio de entidades

Como ya tenemos creadas todas las clases en nuestro modelo, vamos a refactorizarlas para convertirlas en clases entity. Recordemos que estas clases siempre tendrán que asociarse a una tabla de nuestra BD.

Comenzamos por referenciar nuestras clases en el xml de configuración, para ello seguimos la plantilla:

```
<class>package.Clase</class>
```

Configuración programática

Actualmente, es cada vez más común realizar una configuración programática del Entity JPA. Esto tiene una serie de ventajas:

- Permite detectar errores en tiempo de ejecución.
- Nos evita tener que codificar un archivo XML.
- Nos permitirá añadir variables de entorno a nuestra configuración y mantener la confidencialidad de los datos.
- Facilita el cambio dinámico de datos cuando sea necesario.
- Usa Java con lo que nos sentiremos más naturales.

Para ello, dentro de nuestra clase **util**, deberemos añadir un método que devuelva un mapeado estático con todas las configuraciones y pasarlo como argumento a nuestro método de creación del Entity Factory.

En nuestro caso ha sido la opción que hemos escogido, puede verse el código implementado dentro de la clase *ConexionJpa*.

Refactorizando los modelos

Para trabajar con Hibernate vamos a hacer uso de decoradores en nuestras clases, de esta manera, el ***framework*** sabrá qué datos tiene que tratar y cómo tiene que hacerlo. Comenzamos por añadir los decoradores necesarios a nuestros modelos de entidades para que Hibernate sepa que son **entity classes** y son clases de persistencia que se van a guardar en el contexto de JPA. En definitiva, lo que hace Hibernate o JPA es levantar una BD virtual con un paradigma orientado a objetos, estos objetos se sincronizan con la BD y están alojados dentro de contenedores que maneja Hibernate. Cuando se hace un commit, los datos se actualizan entre las dos BD, la real y la virtual.

Para empezar, importamos la librería en nuestras entidades:

```
import jakarta.persistence.Entity;
```

Decoradores:

- @Entity → Se coloca antes de la definición de la clase e indica que es una **entity class**.
- @Table → Indicamos explícitamente el nombre de la table de correspondencia en el modelo relacional.
- @Id → Señala el campo que usaremos como PK.
- @GeneratedValue() → Indicamos que el valor PK es generado y autoincremental. Como argumento pasaremos la estrategia de creación, en nuestro caso, autoincremental: (strategy = GenerationType.*IDENTITY)*
- @Column(name = nombre) → Indicamos el nombre de la columna de referencia de modo explícito.

Los atributos que se llamen del mismo modo se mapean de forma automática.

Dentro de nuestras clases es importante que siempre tengamos un constructor vacío para que lo pueda usar JPA.

Uso de FK

Para indicar que uno de nuestros campos se traduce como una FK en nuestro modelo relacional, usaremos la cardinalidad del atributo:

- @OneToOne
- @ManyToOne
- @OneToMany
- @ManyToManny

Refactorizando la clase Utility

Para poder crear un **entity manager**, el administrador de clases **entity** que permite implementar el CRUD, debe generarse una clase que maneje la conexión con nuestra BD. Como hicimos con JDBC, implementaremos la clase dentro del package ***utils,*** de momento, mantendremos nuestra clase ***Conexión*** así que crearemos una nueva: ***ConexionJpa***. Esta clase irá importando los elementos necesarios de ***jakarta.persistence***.

Implementamos Factory y Singleton

Mantendremos los patrones de diseño que hemos ido usando aplicando los conceptos de Factory y Singleton.

Comenzaremos por crear un objeto Factory que nos permita crear ***entity managers***. Cada objeto es para un cliente o petición en particular, igual que cuando trabajábamos con el objeto **Connection** en JDBC. Si estuviéramos en un ambiente Web, necesitaríamos un **entity manager** por cada request. Recordemos, el objeto Factory es único, pero puede crear tantas entidades como sea necesario.

Para esto, crearemos el atributo ***entityManagerFactory*** del tipo ***EntityManagerFactory*** y un método estático que devuelve un administrador que obtenemos a través de la unidad de persistencia. Como argumento, pasaremos el nombre que hemos dado a nuestra entidad de persistencia:

```
// Creamos el atributo para instanciar los objetos de tipo EntityManagerFactory.

private static final EntityManagerFactory entityManagerFactory = buildEntityManagerFactory();



// Generamos el método Factory para la conexión a la base de datos, además, aplicamos el patrón Singleton.

private static EntityManagerFactory buildEntityManagerFactory() { return Persistence.createEntityManagerFactory("onlinestoreJPA"); }
```

Realizando esta implementación, estamos aplicando también el patrón Singleton.

Cuando se usa la clase por primera vez, este código carga una constante. Solo se inicializa una sola vez, luego obtendremos la constante mediante un método getter, con lo que mantendremos el patrón Singleton, una conexión por cliente.

Refactorizando el programa main

Recordemos que en nuestro método **main** realizábamos la conexión a la BD, así que tenemos que adaptar el código y generar nuestro **entity manager**.

También tenemos que cerrar la conexión una vez finalizada la App.

Reimplementación de los controladores

Ahora, nuestros controladores trabajarán con Hibernate/JPA, así que debemos refactorizar todo el código de los controladores del repositorio. Hay que tener en cuenta algunos puntos:

- Trabajaremos con métodos sobre la clase Entity Manager.
- No podemos implementar **auto-close** como en LDBC, así que tenemos que cerrar las conexiones de forma manual.

En general, vamos a ver que la generación de peticiones y captura de resultados se va a simplificar mucho debido al uso tanto de los métodos del Entity Manager como de la sintaxis embebida para crear las consultasa. Cómo no podemos usar *auto-close*, implementaremos los bloques *try-catch* con ***finally*** para ejecutar el cierre del Entity Manager.

Una de las ventajas de Hibernate es que, como trabajamos en un contexto donde tenemos una bd virtual basada en POO, no necesitamos mapear manualmente nuestros objetos para convertirlos en modelos relacionales, el ***framework*** se encarga de todo. Esto nos aporta mucha agilidad y reduce enormemente la complejidad de nuestro código.

Una ventaja muy importante de trabajar con un ***framework*** como Hibernate es que simplifica enormemente la codificación del repositorio DAO, aligera muchas líneas de código, lo deja muy ordenado y legible y muy fácil de refactorizar.

Interfaz

Así, la base de trabajo con el repositorio DAO es la misma, tenemos que crear los distintos **interfaces** para las entidades de nuestro repositorio y las clases DAO que implementan los contratos con estas, para diseñar nuestros controladores para la transferencia de datos con la BD. Estas clases suelen incluirse dentro de un ***package*** llamado **repositories**.

Por otro lado, un repositorio sencillo como el nuestro, en lugar de una interfaz por cada una de las entidades, puede implementar una única interfaz del tipo ***java generics*** con la que realizaremos el contrato en cada uno de los DAO. AL implementarla como una clase genérica nos aseguramos su reusabilidad. Esta clase deberá declarar todos los prototipos de métodos que tienen que implementar nuestras clases CRUD. Esta es la opción que hemos seguido nosotros, a continuación, se muestra una versión resumida sin los comentarios:

```java
package ciricefp.modelo.repositorio;
import ciricefp.modelo.listas.Listas;

public interface Repositorio<T> {
    Listas<T> findAll();
    T findById(Long id);
    T findOne(String key);
    boolean save(T t);
    boolean delete(Long id);
    int count();
    T getLast();
    boolean isEmpty();
    boolean resetId();
}
```

Clases DAO

A continuación, refactorizamos todas las clases DAO, asegurándonos de cumplir con los contratos de implementación y cambiando todos los métodos para usar Hibernate en lugar de JDBC. Los métodos van a ser mucho más simples y concisos ya que el ***framework*** realizar de forma automática la mayor parte de los procesos (manejar la consulta, poblar los resultados, mapear los objetos…).

El primer cambio importante es que deberemos obtener la instancia de **entity manager** que generamos al inicio de la aplicación. A continuación, eliminaremos los bloques try-catch de nuestro código, ya que se manejarán en las clases *services* y, por último, refactorizaremos todo el código para usar los métodos JPA y las sentencias en HQL/JPQL propios del *framework* Hibernate.

Al final del proceso veremos cómo hemos obtenido un código mucho más simple, legible y propio de la POO.

Generando Services

Para mejorar el patrón de diseño de nuestro código, manejaremos toda la lógica de las clases DAO mediante servicios, es una buena práctica desacoplarlo del DAO. De este modo, las clases ***service*** implementarán todo el manejo de las transacciones de nuestros DAO.

Un *service* es un patrón de diseño que permite:

- Desacoplar toda la lógica de negocio de nuestros DAO y de nuestros contraladores y la aislamos en una clase especializada.
- Podemos trabajar con varios DAO en un **service** y estos pueden colaborar para llevar a cabo tareas complejas.

- Aísla el manejo de transacciones para todas las acciones de escritura.

Tendremos que colocar toda esta lógica en un ***package*** nuevo dentro de nuestro proyecto. Proseguiremos creando las **interfaces** para nuestros DAO. Cada servicio contendrá los métodos que queramos implementar, normalmente muy parecido a nuestra ***interface*** para los DAO.

Si queremos añadir métodos con retorno opcional, para evitar errores de NullPointerException, podemos usar la clase **Optional** de Java Utils.

```java
package ciricefp.modelo.services;
import ciricefp.modelo.Articulo;
import ciricefp.modelo.listas.Listas;
import java.util.Optional;

public interface ClienteService {
    Listas<Articulo> findAll();
    Articulo findById(Long id);
    Optional<Articulo> findOne(String key);
    boolean save(Articulo articulo);
    boolean delete(Long id);
    int count();
    Optional<Articulo> getLast();
    boolean isEmpty();
    boolean resetId();
}
```

A continuación, procedemos a crear las clases **services** donde implementaremos la lógica de nuestros interfaces. Seguimos la nomenclatura **ClaseServiceImpl**. Los servicios no trabajan directamente con el repositorio en el sentido más estricto, ya que las operaciones las realizan nuestras clases DAO, pero implementan el acceso al **entity manager** para poder cargarlo en nuestras clases DAO y para poder manejar las transacciones.

El cierre de las conexiones se tiene que realizar fuera del *service*. Si lo cerramos dentro del método, no podremos invocar el resto de los métodos.

Refactorizaciónd el controlador interno *Datos*

Como último paso, hemos de refactorizar la lógica del controlador Datos para que use el ***entity manager*** y los servicios que hemos creado.

Parametrización

Como con JDBC, podemos pasar parámetros a nuestras peticiones, para hacerlo, usamos el signo de interrogación '?' para indicar que vamos a introducir un parámetro y, a continuación, el número de parámetro que se corresponde a esa posición:

```
Query query = em.CreateQuery("select c from Cliente c where c.nif = ?1", Cliente.class);
```

Funciona muy parecido a las ***Prepared Statement.***

Para pasar un parámetro, usamos:

```
query.setParameter(1, "valor");
```

Puede ser necesario que realicemos un cast del resultado obtenido.

Otra forma muy intuitiva de pasar los parámetros es usando un nombre:

```
articulo = em.createQuery("select a from Articulo a where a.codigo = :codigo", Articulo.cl
                    .setParameter("codigo", key)
                    .getSingleResult();
```

Métodos más comunes

- **createQuery**() → Nos permite crear una petición personalizada.
- **find**() → Devuelve un registro buscando por su PK.
- **merge**() → Sirve para modificar registros.
- **persist**() → Sirve para crear registros.
- **createStoredProcedureQuery**() → Permite ejecutar un procedimiento almacenado.

Transacciones

Todas las acciones de escritura deben de ejecutarse dentro de una transacción para asegurar la integridad de los datos de nuestra BD.

Para ello, usaremos el método **getTransaction()** de nuestra clase EntityManager.

```
em.getTransaction().begin(); // Inicia la transacción.
... Acciones CRUD de escritura.
em.getTransaction().commit(); // Realiza el commit y guarda los datos en la BD.
```

Dentro del boque **catch** deberemos implementar el **rollback**:

```
} catch (Exception e) {
  if (em.getTransaction().isActive()) {
    em.getTransaction().rollback();
  }
}
```

Cada vez que realizamos un **commit**, nuestro contexto virtual se sincroniza con nuestra BD relacional.

Manejo de asociaciones

Al usar Hibernate u otros **framewroks** basados en JPA, podemos usar un modelo orientado a objetos en lugar de un modelo relacional para manejar las asociaciones entre nuestras clases. De este modo, podemos mapear las relaciones mediante llaves FK usando los decoradores de JPA en los atributos de nuestras entidades. Hibernate transforma de forma automática el campo en una FK siguiendo el patrón clase_id.

1. Decoradores: Como hemos visto más arriba, tenemos un decorador para cada tipo de asociación relacional. La primera cardinalidad del decorador siempre se refiere a la clase en la que estamos trabajando, la segunda, a la clase asociada:

```
// Un pedido (origen) solo puede tener un cliente (asociada), pero un
// cliente (asociada) puede tener varios pedidos (origen).
@ManyToOne
private Cliente cliente;
```

2. Si queremos asegurarnos de estar haciendo referencia a la columna FK correcta, podemos usar el decorador @JoinColumn():

```java
@ManyToOne
    // Nos aseguramos de la correcta ref de columna
    @JoinColumn(name = "cliente_id")
    private Cliente cliente;
```

3. Podemos especificar cómo Hiebrnate irá a buscar los datos asociados la BD mediante ***Fetch.***
    1. FetchType.LAZY → Carga el recurso sobre demanda, solo cuando se necesita. Por ejemplo, cuando se usa el método Get. Con este método mejora la optimización. Siempre que podamos, usaremos el LAZY.

       Por defecto, se usa en las asociaciones @xToMany.
    2. FetchType.EAGER → Se cargan todos los datos de forma inmediata, por detrás, usa los JOIN necesarios. Con este método empeora la optimización, pudiendo encadenarse varias consultas en cascada.

       Por defecto, se usa en las asociaciones @xToOne.

Manejo de la herencia

Hibernate ofrece varias opciones para trabajar con la herencia de nuestras entidades orientadas a objetos y trasladarlas a la lógica relacional. Para automatizar y simplificar el proceso, que ya vimos que era bastante denso de codificar mediante JDBC, se hace uso de los ***decoradores*** en las clases padre e hijas para indicar a Hibernate qué comportamiento deberá tener para reflejar la herencia en el modelo relacional.

Trabajar con Hibernate nos va a ofrecer una enorme ventaja frente a hacerlo mediante **raw** JDBC:

- No tenemos que manejar las FK de forma manual.
- Podemos implementar de forma muy simple las distintas formas de manejar la herencia en el modelo relación: ***Tabla única, tablas relacionadas mediante FK, tabla completa por clase…*** En nuestro caso, usamos la estrategia de tablas relacionadas o **joined tables** que, aunque puede decrementar el rendimiento de una BD relacional, ofrece una solución consistente y sólida con un manejo muy lógico a nivel relacional.
- No tenemos que crear nuevos campos en las clases ya que se automatizará la creación de las columnas necesarias mediante los decoradores.
- Podemos ser tan específicos como queramos en nuestros decoradores para declarar el comportamiento de nuestra BD.

Estrategia joined tables

Este método genera una tabla padre con los campos comunes y las tablas hijas con los campos específicos de cada especificación de la super clase. Para manejar la herencia, relaciona las

tablas hijas mediante FK con la tabla padre. Así, las consultas sobre las herencias se realizan mediante JOINS, lo que puede disminuir algo el rendimiento si hay mucha jerarquía de niveles de asociaciones, pero en nuestra opinión, es una solución simple y consistente.

1. Declaramos la clase padre en la lógica relacional mediante el decorador @Inheritance(strategy = InheritanceType.JOINED), con esta anotación indicamos que todas las clases hijas tendrán su propia tabla.
2. EL campo indicado como PK de nuestra clase se usará como FK en las clases hijas, este proceso se realizará de forma automática e Hibernate usará el atributo con el decorador @Id.
3. En las entidades de nuestro modelo orientado a objetos que ***extiendan*** la clase padre, deberemos usar el decorador @PrimaryKeyJoinColumn(name = "nombre_campo_id_padre") para indicar el nombre de la columna donde relacionaremos la FK. Normalmente, se usa 'clase_padre_id'.

Si tuviéramos que especificar un comportamiento distinto a **delete on cascade** y **update on cascade** podemos usar las anotaciones @OnDelete y @OnUpdate.

Observaciones sobre el código

- Hemos implementado los patrones Factory DAO y Singleton.
- Hemos usados programación funcional siempre que ha sido posibles para mejorar la legibilidad y facilitar el mantenimiento y la escalabilidad de nuestro código.
- Hemos realizado una configuración programática de nuestro Entity Factory y hemos aplicado restricciones de seguridad en nuestra configuración, además de usar una configuración explícita para controlar todos los aspectos que nos interesan.
- Manejo de excepciones y errores y control de argumentos *null*.
- Seguimiento de la parte del código donde puede surgir un problema mediante el uso del valor de salida en el programa main.
- Modularización del código atendiendo a la reusabilidad, legibilidad y escalabilidad.
- Uso de variables de entorno para no revelar datos sensibles.
- Uso de Procedimientos Almacenados.
- Uso del patrón Factory también para la implementación de la herencia.
- Uso de Maven para la gestión de dependencias y creación de un JAR con las dependencias incluidas.

Uso de Querys y uso de Procedimientos Almacenados

La clase **Articulo, Direccion y Pedido**, la hemos implementado mediante métodos de creación de queries o mediante métodos propios de Hibernate JPA para ver cómo se implementa el servicio de forma más manual.

La clase Pedido la implementamos usando llamadas a los procedimientos almacenados de nuestra BD, ya que los generamos teniendo en mente asegurar la seguridad de nuestros datos y poder gestionarla de forma sencilla a tavés de roles. Para vernos abligados a funcionar

únicamente con procedimientos, tendríamos que configurar el usuario de la BD en nuestra APP diferente a root o admin.

Hemos usado consultas en HQL/SPQL.

Uso de Optionals

Como hemos usado el tipo **optional** para manejar las excepciones NullObject, tenemos que reimplementar toda la lógica que comprobaba si el resultado de una acción devolvía **null** y poasarlo a una comprobación **optional**:

```
optionalTipo.ifPResent(element -> // Hacemos algo con el elemento);

// Si lo que queremos es imprimir un optional, la implementación funcional es aún más
// sencilla.
optionalTipo.ifPresent(System.Out::println);

// Cuando tenemos que descriminar entre el retorno de un valor o podemos obtenr un retorno
// null, usamos orElse.
return datos.createArticulo(descripcion, pvp, gastosEnvio, tiempoPreparacion).orElse(null);
```

Refactorización de la redundancia de bloques try-catch

En los casos en los que una clase raíz ya realizaba el manejo de excepciones se ha eliminado la redundancia para aligerar el código. Esto se ve sobre todo en la clase ***Datos*** ya que la mayor parte del manejo de excepciones se está realizando en las clases ***service***.

Análisis del producto

Elementos de mayor dificultad o que han exigido mayor dedicación

Lo cierto es que pasar de la implementación mediante JDBC a Hibernate simplifica mucho todo el código y resulta muy intuitivo una vez se van dominando los principios. Seguramente, la parte más complicada del producto ha sido asegurarnos cómo se implementaba correctamente la herencia, aunque una vez hecho resulta muy sencillo, y cogerle el truco a trabajar con tipos de retorno Optional.

Puntos del enunciado no cumplidos

Por suerte, hemos podido cumplir con todos los puntos de esta actividad e implementar todos los requisitos esperados. El haber preparado un flujo de trabajo controlado, accediendo a unos

recursos muy buenos y siguiendo un orden de implementación para la producción establecido para facilitar el proceso, nos ha permitido poder acabar el producto a tiempo y realizar una entrega de todos los apartados y funcionalidades. También hemos podido revisar y prepara las funcionalidades opcionales y revisar los errores de redundancia de nuestro código anterior.

Mejoras y refactorizaciones

Los aspectos en los que nos gustaría trabajar y que nos quedan pendiente son:

- Validación de los datos introducidos por el usuario: Ahora mismo el usuario puede introducir datos que rompan la consistencia de nuestra aplicación, por ejemplo, una dirección de email inválida.
- Uso de herencia múltiple de *interfaces*: Una de las prácticas recomendadas para asegurar que nuestro código es lo más limpio, reusable y mantenible posible es no amalgamar todas las acciones en una única interfaz si no usar la herencia múltiple de interfaces. Así, creemos que un elemento a mejorar sería modelar más correctamente nuestra interfaz *Repositorio*. Para ello, llevaríamos las acciones CRUD a una interfaz independiente, las comparaciones para el filtrado a otra y las acciones de conteo a otra. Podemos, entonces, refactorizar nuestra interfaz *Repositorio* con un *extends* en el que añadimos las diferentes interfaces que implementa y así tenemos una *interface* con herencia múltiple.
- Implementación de un *pool* de conexiones mediante un servicio externo.

Autoevaluación

Teniendo en cuenta que hemos podido cumplir con todos los requisitos esperados para el producto y que, además, hemos procurado optimizar el código y el diseño pero que, como hemos comentado, queda por solucionar el problema de la validación de datos y el *pool* externo, nuestra autoevaluación sería de un 95.

Otras observaciones sobre nuestro código

Nos gustaría detallar algunos puntos que hemos seguido en la implementación de nuestro Producto y que pensamos que son interesantes para la evaluación:

- Uso de funciones recursivas.
- Uso de programación funcional.
- Manejo de excepciones y errores y control de argumentos *null*.
- Seguimiento de la parte del código donde puede surgir un problema mediante el uso del valor de salida en el programa main.
- Modularización del código atendiendo a la reusabilidad, legibilidad y escalabilidad.
- Uso de variables de entorno para no revelar datos sensibles.
- Uso de PreparedStatemens y CallableStatements para aumentar la seguridad y evitar ataques por inyección SQL.
- Uso del patrón Factory también para la implementación de la herencia.
- Uso de Maven para la gestión de dependencias y creación de un JAR con las dependencias incluidas.

Recursos

Máster Completo en Java de cero a experto 2023 (+127 hrs). (s. f.). Udemy. https://www.udemy.com/course/master-completo-java-de-cero-a-experto

Stripe Developers. (2021, 12 enero). *Environment Variables .env with Java* [Vídeo]. YouTube. https://www.youtube.com/watch?v=o1HsGbTZObQ

Álvarez Caules, Cecilio. (2023, 12 de marzo). *Usando el patrón Factory*. https://www.arquitecturajava.com/usando-el-patron-factory/

Calle, N. R. (2023). Herencia con Hibernate. *Refactorizando*. https://refactorizando.com/herencia-hibernate/