PROGRAMACIÓN ORIENTADA A OBJETOS CON ACCESO A BBDD

PERSISTENCIA EN BASE DE DATOS

MIEMBROS

Katiane Coutinho Rosa
Nadia Igual Bravo
Xavier Melo Pardo
Daniel Boj

Grupo - Cirice

Producto 3. Persistencia en Base de Datos

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

Interacción entre Java y MySQL mediante JDBC

JDBC es la API de Java para manipular bases de datos SQL. SQL es el lenguaje que se usa para administrar y realizar consultas sobre los datos de un sistema de gestión de bases de datos relacionales (RDBMS). La base de este tipo de BBDD son las **tablas**, que contienen los datos, y las **relaciones** que se establecen entre ellas.
SQL en un lenguaje descriptivo que se divide en distintos tipos de sentencias:

- DDL → Para la creación de las bases de datos y sus entidades.
- DML → Gestiona los datos que introducimos en la base de datos en las tablas que ya hemos creado e implementa las acciones CRUD.
- DCL → Gestiona el control de usuarios y roles.
- DTL → Gestiona el manejo de las transacciones.

Una tabla es como una matriz, formada por filas y columnas, las columnas contienen los atributos y las filas son los registros con datos para cada uno de los campos. Las tablas se relacionan e interactúan entre ellas, estas acciones las controlamos mediante la **integridad relacional.** Podemos entender las tablas, pensando en Java, como un array de arrays.
Basaremos las acciones sobre la base de datos en las **acciones CRUD**: Create, Read, Update y Delete.

Traducción del Modelo de Clases a una Base de Datos Relacional

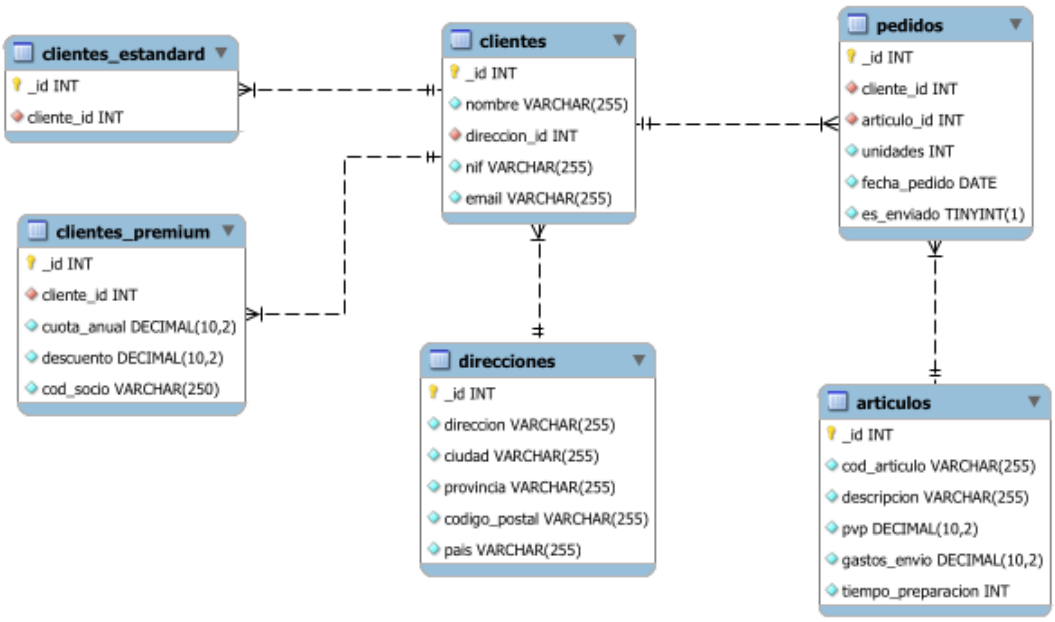
La primera tarea que hemos de realizar, ya que partimos de dos paradigmas distintos, es diseñar nuestra base de datos para que sea capaz de implementar el modelo de clases que hemos diseñado en nuestros productos anteriores. Para ello, hemos de tener en cuenta que deberemos ser capaces de **llevar las asociaciones y herencias entre nuestras clases a un diseño relacional**.
De forma general, hemos de convertir nuestras clases en tablas, las propiedades en columnas y las asociaciones formarán las relaciones, así que deberemos gestionar las FK cuando una clase aparezca dentro de otra clase. Debemos tener especial cuidado si aparece una asociación N:M y en usar las FK correctas para gestionar las herencias.
En cuanto a la implementación del script, tras la creación de la BD, la primera parte, genera las distintas tablas y sus restricciones. Además, creamos un *trigger* para manejar la correcta eliminación de clientes ya que, recordemos, se trata de una relación de herencia. Tras esto,

creamos los distintos *Stored Procedures* que ejecutarán las acciones sobre la BD. Usamos procesos por las siguientes ventajas:

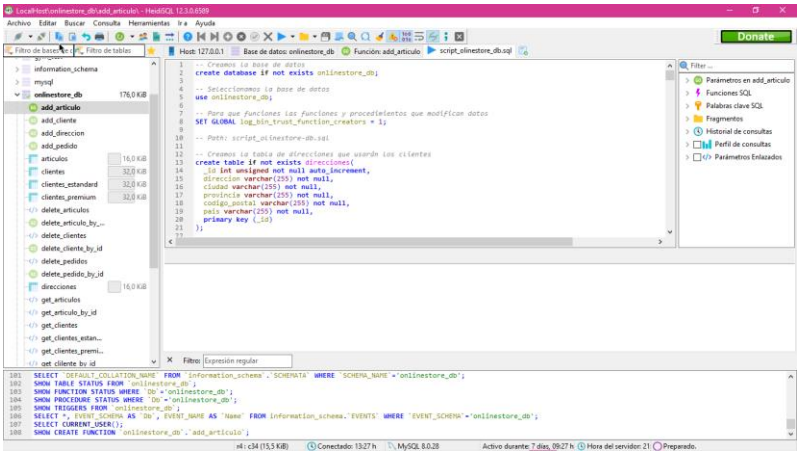
- Ayudan a evitar ataques por inyección SQL.
- Facilitan el manejo de las políticas de acceso y seguridad ya que podemos establecer que usuarios o roles únicamente puedan usar procesos o sets de procesos concretos y cancelar el acceso directo al uso de cualquier tipo de sentencia SQL.
- Permiten implementar las reglas de negocio complejas que establezcan los requisitos de nuestra aplicación de una forma mucho más concreta que con el uso de *constraints.*

Por último, realizamos la creación de roles y usuarios, que se usarán en la versión de producción de la APP e implementamos los dos procesos extras que manejan la carga de datos de test al realizar una instalación limpia de nuestra APP.

Esquema de la Base de Datos

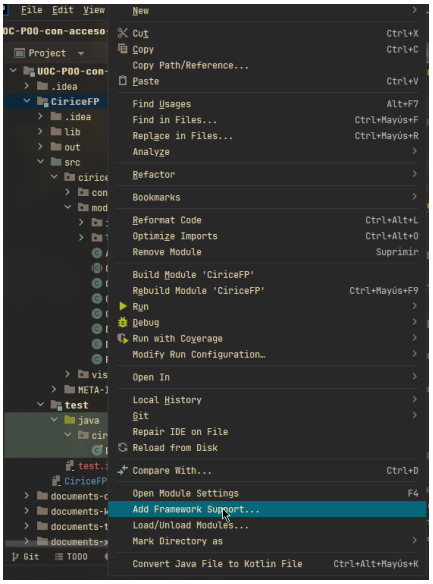


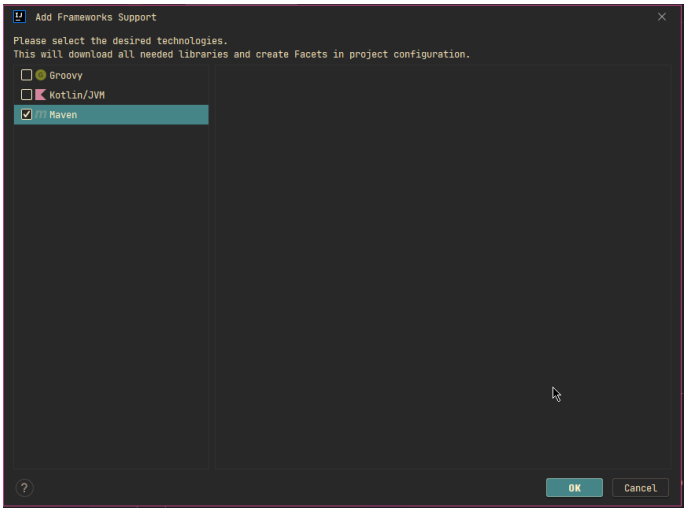
Ejecución del script DDL



Añadir Maven para la gestión de librerías

Maven es una herramienta de Java que nos permite realizar la **gestión de la configuración del proyecto y de las librerías Java** de forma sencilla y, además, facilita enormemente los procesos de despliegue de una app y nos **permite crear ejecutables con las dependencias incluidas**.
Para añadir *Maven* a nuestro proyecto, debemos abrir la ventana de *Project tool* y seleccionar *Add Framework Support*. Una vez dentro del menú contextual, escogemos *Maven* y damos a *Ok*.

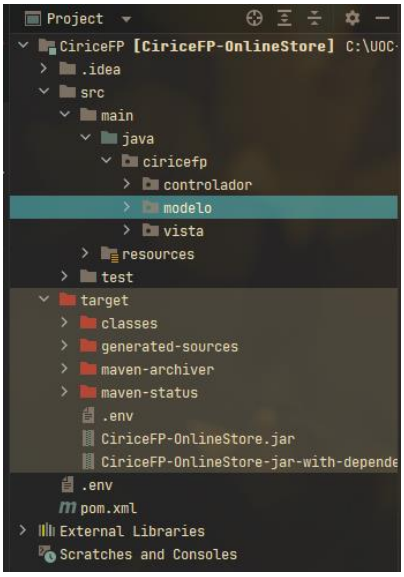




A continuación, se generará el archivo *pom.xml* desde donde podremos editar las caracteríasticas de nuestro proyecto.

```
<groupId>CiriceFP</groupId>
<artifactId>CiriceFP-OnlineStore</artifactId>
<version>1.0</version>
```

Para seguir la jerarquía de directorios de Maven hemos de cambiar ligeramente nuestra estructura de proyecto:



Agregación de dependencias

A través del archivo *pom.xml* podemos agregar las dependencias que vayamos a usar en nuestro proyecto. En este caso, agregaremos las dependencias necesarias para trabajar con JDBC a través del repositorio central de *Maven*. Aunque sea un cambio al modo de trabajar que hemos seguido hasta el momento, una vez adaptado nuestro proyecto, resulta muy sencillo gestionar las dependencias con esta metodología.

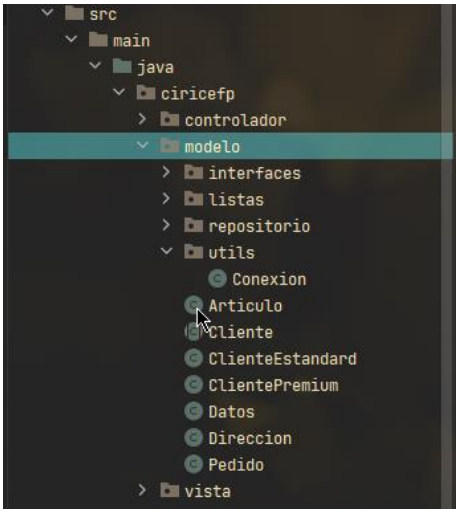
```
<dependencies>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.32</version>
    </dependency>
</dependencies>
```

Tras agregar el código, nos aparecerá un incono para realizar la recarga de *Maven* y todo el proceso de actualización de librerías se realizará de forma automática.

Componente JDBC

Java DataBase Connectivity permite conectar e interactuar con una BD desde Java. JDBC es una API, un conjunto de clases abstractas e interfaces para la gestión y conexión hacia un motor específico de base de datos; esto significa que hemos de escoger un driver en concreto para implementar un conector en relación con el motor de nuestra BD, en este caso escogeremos el motor MySQL. Una vez escogido el driver, deberemos implementar la conexión a la BD y las operaciones CRUD.

Es importante tener en cuenta que, siguiendo la filosofía de Java, esta API también es transparente, con lo que, si en un futuro quisiéramos cambiar de motor, la implementación a realizar sería mínima, cambiar el *driver* y el *string* de conexión bastaría. Para asegurar esta característica, seguimos nuestro patrón de diseño implementado un *package* llamada *utils* donde **implementaremos la clase que gestiona la conexión a la BD**, en nuestro caso, *Conexión*. Así, nuestra app usará una instancia de esa clase para manejar cualquier llamada a la BD. Como comentamos más adelante, aplicamos también el patrón de diseño *Singleton* en el manejo de este recurso.



JDBC implementa una sería de interfaces y clases genéricas para realizar la conexión y las consultas:

- Driver → Permite conectarse a la BD.
- DriverManager → Permite gestionar los drivers.
- Connection → Representa una conexión a la BD.
- DatabaseMetadata → Proporciona información sobre la BD.
- Statement → Permite ejecutar sentencias SQL.
- PreparedStatement → Permite ejecutar sentencias SQL con parámetros de entrada.
- CallableStatement → Permtie ejecutar procedimientos almacenados.
- ResultSet → Contiene las filas obtenidas al ejecutar una sentencia Select.
- ResultSetMetadata → Permite obtener la información sobre la consulta select.

JDBC crea cursores para poder movernos por los resultados obtenidos de la BD. Además, es muy importante, que recordemos que debemos abrir y cerrar los recursos para poderlos usar correctamente. Mediante nuestro patrón de diseño, implementamos una única conexión que se abre al inicio de la aplicación y se cierra en el cierre de esta. Cada llamada realizará un acceso a esta conexión y gestionaremos su cierre con métodos *auto-close* con un *try-catch with resources*.

Evitar ataques por SQL injection - Ventajas de PreparedStatement y CallableSatatement

Mediante el uso del tipo PreparedStatement o CallableStatement podemos generar plantillas de consulta parametrizadas que nos ahorrarán tener que reescribir código. La idea es crear la plantilla de consulta de manera previa y únicamente pasarle los parámetros que necesitemos.
El uso de este tipo de métodos representa un valor añadido a la capa de seguridad de nuestra aplicación **ya que evita que se produzcan ataques por inyección**.

- Paso de parámetros: Tenemos que usar el métodos .set*TipoDato*().
- Preparación de parámetros de salida Tenemos que usar el método .registerOutParameter).

Ejecución:

- Mediante .executeQuery() ejecutamos las sentencias SELECT.
- Mediante .executeUpdate() ejecutamos el restos de sentencias.

Conexión a MySQL

Como comentamos en los anteriores Productos, **centralizamos la conexión a la Base de Datos en una clase independiente, siguiendo el patrón Singleton**, que asegura la reusabilidad del código y mejora su legibilidad, mantenimiento y escalabilidad. Esta clase se encuentra dentro del módulo Modelo y será instanciada por nuestra app. Hemos realizado una pequeña mejora de estructura y ahora la clase se localizará dentro del package *utils* de *model*.
Debemos crear un string de conexión específico para la BD. Para realizarlo de forma parametrizada, primero crearemos las propiedades *url, username* y *password* y lo uniremos todo en la propiedad *conn* del tipor Connection que ejecutará el string de conexión.

Variables de entorno

Para asegurar las buenas prácticas en cuanto a seguridad en nuestro código, en lugar de generar las sentencias de conexión mediante *hard code*, hemos implementado el uso de variables de entorno para nuestro proyecto. Además, como queremos facilitar el despliegue y distribución de la aplicación, en lugar de tener que crear las variables de entorno en el sistema, y siguiendo el modelo que ya usan muchos otros lenguajes de programación, hemos configurado el uso de un **archivo de texto plano .env** que contine las variables de entorno.
Para poder usar estos archivos en nuestro desarrollo, hacemos uso de la librería *io.github.cdimascio* que consigue que Java reconozco y lea estos archivos como si fueran variable env del sistema y crea la clase Dotenv para realizar el acceso a cualquier tipo de variable env, del sistema o de nuestro archivo.
Debemos localizar una copia del archivo .env tanto en la raíz de nuestro proyecto como en la carpeta donde se encuentre el JAR, en nuestro caso, dentro del directorio *tarjets.*

```
ENV=dev

DB_LOCAL_NAME=onlinestore_db
DB_LOCAL_PASS=batty
DB_LOCAL_URL=localhost:3306
DB_LOCAL_USER=root

DB_PROD_NAME=onlinestore_db
DB_PROD_PASS=ciricefp
DB_PROD_URL=uoc-ciricefp-db.c3jhdyiljohu.eu-west-3.rds.amazonaws.com
DB_PROD_USER=admin
```

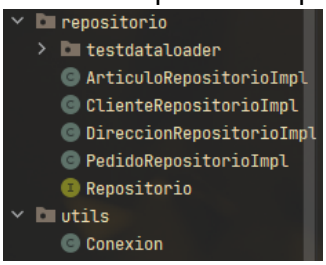
Factory DAO - Creación del *Repositorio*

Para facilitar el consumo de los recursos de la base de datos, debemos implementar los modelos que representen los datos de la tabla y crear el *package repositorio* para manejarlo de forma modularizada. Los modelos deben de tener los mismos atributos que los campos de nuestras

tablas SQL. Es decir, **tenemos que mapear nuestros datos a las clases POO realizando el patrón de diseño Factory DAO, o Repository.**

DAO se refiere a una abstracción de los datos de la persistencia para poder relacionar nuestros modelos relacionales con nuestros modelos de POO. Al trabajar con Java, tenemos que acabar manejando nuestros datos con una orientación 100% POO. La aplicación de estos patrones de diseño permiten traducir de forma efectiva nuestros modelos ERM y usarlos en una aplicación diseñada mediante el paradigma POO. Transformaremos nuestras tablas en colecciones de objetos.

Para seguir el patrón Factory, todos los modelos DAO implementarán una interfaz que prototipará el modelo de acciones que podrá realizar. Dada que la complejidad de nuestras entidades no es muy elevada y todas van a implementar las mismas acciones, **para simplificar el diseño de nuestra app usamos una única interfaz Factory creada usando Java Generics** que implementarán todas nuestras clases DAO ya que podrá adaptarse a cada tipo. Hay diversas opciones de nomenclatura para seguir este patrón, en nuestro caso, creamos todas las clases e interfaces dentro del *package repositorio*, la interfaz se llama *Repositorio*, si hubiera más de una deberíamos especificar también el nombre del modelo de datos de la entidad, otra nomenclatura posible es I*Modelo*Factory. Por último, usamos el modelo de nomenclatura *Entidad*RespositorioImpl para las clases DAO.



Primero, crearemos las distintas clases base necesarias para manejar los modelos de las entidades de nuestra aplicación y poder comunicarnos con la BD. Como en nuestro caso ya tenemos los distintos modelos de entidades creados, deberemos adaptarlos para respetar los datos de nuestra BD añadiendo la propiedad *id* que se relaciona con el campo *_id* que tienen todos las entidades de nuestro modelo relacional y nos aseguramos que la clase sobrecargue un constructor que reciba todos los atributos por parámetro que usaremos para recibir los datos de la BD.

Luego, implementaremos la **interfaz del repositorio**, que servirá para controlar todos los métodos CRUD que vamos a usar. Como hemos comentado, implementa el patrón Factory. Todo repositorio debe tener un CRUD para poder manejar las acciones sobre la base de datos, estas acciones básicas son; crear, leer, actualizar y borrar. En nuestro caso, y siguiendo patrones como por ejemplo los que usa Hibernate, usaremos los siguientes métodos:

- FindAll: Para obtener una lista con todos los objetos de una entidad de la BD.
- FindById: Para obtener un objeto de una entidad de la BD identificado por su identificador único.
- Save: Que reunirá las funciones de creación (insert) y modificación (update). Para ello, deberemos comprobar si el objeto que recibimos tiene id o se trata de un nuevo objeto.
- Delete: Para eliminar un objeto de una entidad de la BD identificado por su identificador único.

De este modo, implementamos los métodos básicos para las acciones CRUD, siguiendo de esta manera el **patrón de diseño Factory DAO** básico implementando un **DTO** (*Data Transfer Object)*.
Continuamos implementando las distintas clases que implementarán los modelos para nuestro repositorio, uno para cada una de las entidades de nuestra aplicación. Colocaremos estas clases dentro del package *repositorio* siguiendo el siguiente modelo de nombre: EntidadRepositotioImpl.
Para cada uno de estos modelos, **implementaremos la interfaz** recordando que la hemos **creado como tipo genérico** y podremos pasarle como tipo la clase en la que se basa nuestro modelo, es decir, la entidad dentro de nuestra aplicación. Además, cada una de estas clases deberá implementar el método de conexión a nuestra BD mediante un **método privado**, así no hemos de codificar la conexión en cada una de las acciones. Además, como estamos usando una clase independiente para gestionar la conexión, lo que hacemos es llamar a una instancia de esa clase en lugar de tener que reimplementar de nuevo el código de conexión. Así, aseguramos reusabilidad, mantenimiento y escalabilidad.
Otro aspecto importante de las implementaciones es que **creamos métodos para mapear los resultados obtenidos de la BD**, con lo cuál aseguramos la reusabilidad del código.

Modificaciones en la capa de datos

Al implementar el acceso e interacción con la BD, podemos realizar una serie de cambio sobre cómo estábamos manejando los datos de la aplicación.

- Eliminamos las Listas que usábamos para almacenar Artículos, Clientes y Pedidos ya que podemos trabajar directamente sobre los datos persistentes. De este modo, todas las acciones CRUD se ejecutarán directamente en la BD y así eliminamos un recurso intermedio que significa un aumento del consumo de recursos de nuestra app.
- **Implementación de Clientes mediante el diseño Factory**

Para seguir el modelo recomendado por la actividad y **trasladarlo al uso de la herencia**, hemos aplicado el patrón de diseño Factory a la clase *Cliente*. Así, hemos cambiado bastante la interfaz original y hemos **creado una nueva interfaz IClienteFactory**, en la que mostramos el otro modelo de nomenclatura. En esta, se implementan los métodos estáticos que generalizarán el uso de la clase y sus hijas, por ello, hemos cambiado toda la implementación de la interfaz en las clases originales y hemos pasado a usar los métodos de IClienteFactory. **Desde Java8 podemos implementar métodos estáticos dentro de una interfaz**, con lo que será más sencillo aún generar el patrón.
Tras crear la nueva Interfaz Factory, deberemos implementarla en nuestra clase Cliente.

```
try {
    // Producto 3 -> Mediante Factory podemos crear el cliente directamente.
    Cliente cliente = (Cliente) IClienteFactory.createCliente(nombre, domicilio, nif, email, tipo);
```

```
try (PreparedStatement stmtPremium = getConnection(System.getenv( name: "ENV")).prepare
    // Generamos el modelo para la inserción del Cliente Premium.
    stmtPremium.setLong( parameterIndex: 1, findOne(cliente.getNif()).getId());
    stmtPremium.setDouble( parameterIndex: 2, (IClienteFactory.getCuota(cliente)));
    stmtPremium.setDouble( parameterIndex: 3, (IClienteFactory.getDescuento(cliente)));
    stmtPremium.setString( parameterIndex: 4, (IClienteFactory.getCodSocio(cliente)));
```

Resumen de detalles de la capa de permanencia de datos

A continuación, realizamos un repaso de algunos puntos clave:

- **Lanzamiento de querys mediante CallableStatement**

Las consultas a la BD se lanzan mediante el uso de Procedimientos Almacenados que hemos creado en la lógica relacional. Para ello, se usa el tipo *CallableStatement* del *driver* de JDBC.

```
    sql = "call add_articulo(?, ?, ?, ?, ?, ?)";
}

// Colocamos los recursos como argumentos del try-with-resources para que se cierren automáticamente.
// Creamos la consulta a la BD mediante un PreparedStatement ya que recibimos un parámetro.
try (CallableStatement stmt = getConnection(System.getenv( name: "ENV")).prepareCall(sql)) {
```

- **Mapeado para la traducción de objetos**

Menos en el caso particular de la creación de clientes, que es algo más complejo, se han usado funciones de mapeo para la traducción de objetos POO a relacionales y viceversa.

```
private static void getStatement(Articulo articulo, CallableStatement stmt) throws SQLException {

    // Asignamos los parámetros a la consulta desde el artículo que recibimos por parámetro.
    // La elección del parámetro se hace por su posición.
    // Preparamos el parámetro de salida.
    stmt.registerOutParameter( parameterIndex: 1, Types.NUMERIC);

    stmt.setString( parameterIndex: 2, articulo.getCodArticulo());
    stmt.setString( parameterIndex: 3, articulo.getDescripcion());
    stmt.setDouble( parameterIndex: 4, articulo.getPvp());
    stmt.setDouble( parameterIndex: 5, articulo.getGastosEnvio());
    stmt.setInt( parameterIndex: 6, articulo.getTiempoPreparacion());
}
```

- **Patrón Singleton**

Debido al análisis de los requisitos de negocio, extraemos que los requisitos de concurrencia sobre la BD van a tener una carga muy ligera, por ese motivo, **aplicamos un patrón *Singleton***, una conexión dentro de nuestro contexto, y abrimos una única conexión al iniciar la aplicación que se mantiene hasta el momento del cierre de la aplicación. Las llamadas a la BD irán usando esta conexión que hemos abierto al principio. Se puede comprobar porque los mensajes de conexión y desconexión a la BD por consola solo aparecen una vez.

Si quisiéramos aplicar un patrón con conexiones individuales por cada una de las operaciones, podríamos seguir usando la opción de Java8 para implementar el *auto-close* como argumento del bloque *try*, pero generaríamos la conexión en cada bloque. Además, tenemos que eliminar la conexión inicial a la BD en el programa main y modificar el método *getInstance()* de la clase Conexión para que genere siempre una nueva conexión.
Esta aproximación no es muy recomendable porque puede generar cuellos de botella, sería mejor implementar un pool de conexiones usando la librería de ApacheCommons. Además, a diferencia del caso anterior, el pool de conexiones sigue manteniendo el patron Singleton. Para demostrar como se hace, hemos incluido un bloque en la clase *Conexión* que deja la app lista para pasar a un tipo de conexión por *pool*.

Patrón Singleton

```
// Instanciamos una nueva conexión a la BD siguiendo el patrón Singleton.
Connection db = Conexion.getInstance(System.getenv( name: "ENV"));
```

Ejemplo de conexión múltiple

```
try (Connection db = getConnection(System.getenv( name: "ENV"));
     PreparedStatement stmt = db.prepareStatement(sql);
    ResultSet res = stmt.executeQuery(sql)) {
```

Pool de conexiones

```
public static BasicDataSource getPoolInstance() throws SQLException {
    if (pool == null) {
        pool = new BasicDataSource();

        // Configuramos el pool de conexiones
        pool.setUrl(url);
        pool.setUsername(login);
        pool.setPassword(pass);

        // Configuramos el tamaño del pool
        pool.setInitialSize(3);
        pool.setMaxTotal(10);
        pool.setMinIdle(3);
        pool.setMaxIdle(10);
    }

    return pool;
}

// Obtenemos una sola conexión del pool de conexiones
no usages  new *
public static Connection getPoolConnection() throws SQLException {
    return getPoolInstance().getConnection();
}
```

- **Patrón DAO-Factory**

Se ha seguido este patrón de diseño para manejar la conexión e interacción de nuestra capa de datos con la BD, intentando aplicar todas las normas de código de este para modularizar, independizar y encapsular el manejo de datos.

1. Se ha creado una clase única e independiente para manejar la conexión a la BD dentro del **package utils**, *Conexión*.
2. Hemos creado el ***package repositorio*** donde implementamos toda la lógica de transferencia de datos con la DB. Dentro de este:

- **Interface Repositorio:** Es la interface Factory, debido a que la aplicación que estamos creando es bastante sencilla, es suficiente crear un único Interface que usaran todos los DAO ya que no vamos a necesitar usar métodos únicos para ninguna entidad. Así, la interfaz usa Java Generics para adaptarse a cada DAO.
- **Clases DAO:** Creamos las distintas clases DAO que implementarán el interfaz Factory y especificarán los métodos para cada una de las entidades de nuestra lógica de negocio.
- **Instanciación:** Seguimos el modelo de instanciación de los objetos DAO que permite ceñirnos a la lógica de nuestra interfaz Factory. Usamos la *interface* DTO como clase genérica y la clase DAO como clase específica. De lo más general a lo más específico.

```
// Producto 3 -> Manejamos la creación de un artículo a través del Repositorio.
Repositorio<Articulo> repositorio = new ArticuloRepositorioImpl();
```

Análisis del producto

Elementos de mayor dificultad o que han exigido mayor dedicación

En nuestro caso, la parte del desarrollo de este producto que ha exigido una mayor dedicación ha sido la implementación de las acciones sobre la Base de Datos en los objetos DAO. En un principio, hicimos los tests de las operaciones mediante *PreparedStatements* y *Statements*, y funcionaba todo correctamente. Pero al pasar al uso de *CallableStatements* empezó a fallar toda la lógica. Como hemos comentado incluso en los foros generales de la asignatura, todo el problema se debía al uso del constraint *Not Null* en la creación de las tablas que interfería en la preparación de las sentencias en nuestro código Java. Además, tuvimos que prestar especial atención al manejo correcto de los argumentos de salida que también nos dieron algún quebradero de cabeza hasta que aprendimos a gestionarlos correctamente.
Por otro lado, otro punto que ha requerido especial atención ha sido la asimilación del patrón Factory y cómo aplicarlo de manera práctica. Al final, en nuestra opinión, es más complicado leerlo que llevarlo a la práctica y se entiende mucho más fácilmente al implementarlo.

Puntos del enunciado no cumplidos

Por suerte, hemos podido cumplir con todos los puntos de esta actividad e implementar todos los requisitos esperados.

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.

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, 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.

Creación del JAR con Maven

Al estar trabajando con un proyecto Maven, la forma de generar el archivo ejecutable Jar difiere de la usada con anterioridad. Para empezar, Maven ubica los archivos generados en el directorio *target.*

Por otro lado, tenemos que generar un archivo de configuración para que nuestro ejecutable se genere de forma correcta. Además, podemos añadir las dependencias al archivo JAR para que pueda distribuirse de forma global. Como comentábamos, hemos de añadir la configuración específica de nuestro artefacto y, para ello, usaremos el archivo *pom.xml*. Dentro de este archivo hemos de introducir el siguiente esquema:

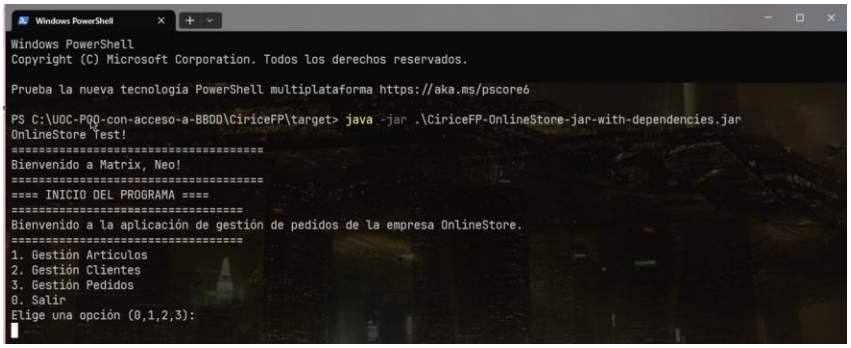
```
<build>
    <finalName>CiriceFP-OnlineStore</finalName>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-assembly-plugin</artifactId>
            <version>3.4.2</version>
            <configuration>
                <descriptorRefs>
                    <descriptorRef>jar-with-dependencies</descriptorRef>
                </descriptorRefs>
                <archive>
                    <manifest>
                        <addClasspath>true</addClasspath>
                        <classpathPrefix>lib/</classpathPrefix>

<mainClass>ciricefp.controlador.OnlineStore</mainClass>
                    </manifest>
                </archive>
            </configuration>
            <executions>
                <execution>
                    <id>make-assembly</id>
                    <phase>package</phase>
                    <goals>
                        <goal>single</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

Mediante este esquema indicaremos el nombre del ejecutable, el plugin de Maven que usaremos para crearlo, la dirección de la clase *main* y que queremos crear el proyecto con las dependencias, por ello creamos el JAR mediante el plugin *Assembly* y no *Jar*. El proceso genera 2 ejecutables y hemos de tener en cuenta que el que va a funcionar va a ser el que cuyo nombre indica *with-dependencies*. Cuando usamos capa de datos persistente, es muy importante que generemos el proyecto con dependencias.
Finalmente, podemos generar el archivo de una forma muy ágil trabajando directamente desde la terminal. Para ello, abriremos un *bash* en el directorio de nuestro proyecto y ejecutaremos:

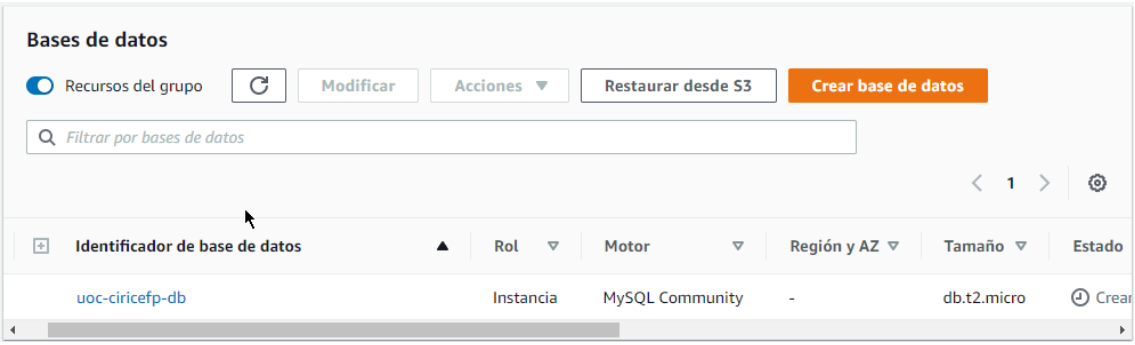
```
$ mvn package
```

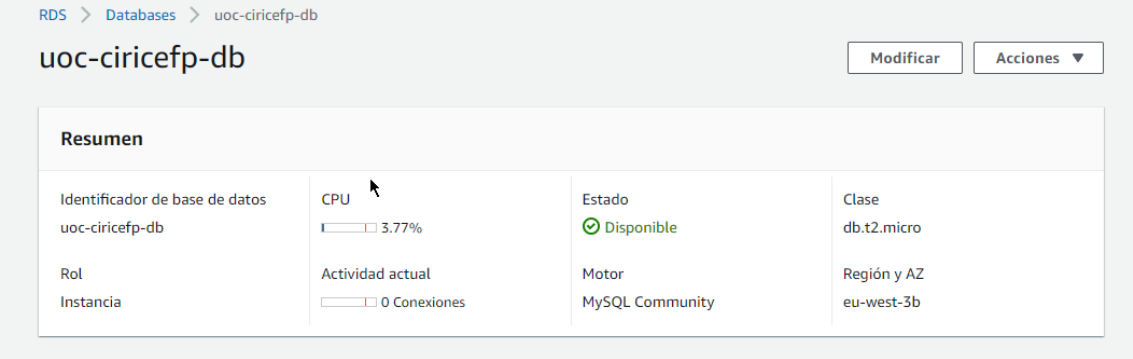
A continuación, podemos ver la ejecución del archivo por consola:



Creación de la BD en AWS

Para simular de forma más realista el despliegue de nuestra APP, hemos creado una BS de la nube gracias al *Free Tier* de AWS. Como el uso de esta capa de servicio tiene límite de horas, realizaremos todo el desarrollo de la aplicación en modo local y, posteriormente, prepararemos el despliegue de la BD en la nube para la demostración del producto final.





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

Baeldung. (2021, 24 diciembre). *How to Create an Executable JAR with Maven*. Baeldung. https://www.baeldung.com/executable-jar-with-maven

Mkyong. (2019, 19 junio). *JDBC CallableStatement – Stored Procedure OUT parameter example - Mkyong.com*. Mkyong.com. https://mkyong.com/jdbc/jdbc-callablestatement-stored-procedure-out-parameter-example/

Java CallableStatement - javatpoint. (s. f.). www.javatpoint.com. https://www.javatpoint.com/CallableStatement-interface

GeeksforGeeks. (2022). How to Call Stored Functions and Stored Procedures using JDBC. *GeeksforGeeks*. https://www.geeksforgeeks.org/how-to-call-stored-functions-and-stored-procedures-using-jdbc/

IBM Documentation. (s. f.). https://www.ibm.com/docs/es/db2-for-zos/12?topic=sql-calling-stored-procedures-in-jdbc-applications

Pankaj. (2022). CallableStatement in Java Example. *DigitalOcean*. https://www.digitalocean.com/community/tutorials/callablestatement-in-java-example

Environment variables and program arguments | AppCode. (s. f.). AppCode Help. https://www.jetbrains.com/help/objc/add-environment-variables-and-program-arguments.html

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/