Curso de Java SE: SQL y Bases de Datos JDBC

[¿Qué es Java Database Connectivity (JDBC)? 2](#_4d9rqsv2k1kc)

[Arquitectura de la API JDBC 2](#_rdh11ero5gkk)

[Métodos y componentes de JDBC 2](#_85jakuhbjd9v)

[Conectando Java con bases de datos 2](#_t5z8iqimzor6)

[Leer y crear datos con JDBC 2](#_116kh1q73qli)

[Actualizar y eliminar datos con JDBC 2](#_3vq0nqsrd5de)

[CRUD 2](#_dl2f9fz9myh8)

[Clases y estructura del proyecto con IntelliJ y Maven 2](#_jzsqyhvvk2hn)

[Optimización de recursos 2](#_ksqqneubqqsm)

[Patrón DAO y Repository 2](#_7ig3kf2ra7f6)

[Implementando el Patrón Repository 2](#_4uszn4o5gh28)

[Leer Datos desde Java 2](#_5fzowsbqk2w5)

[Insertar Datos en Java 2](#_sb4rexav3o0v)

[Eliminar Datos en Java 2](#_78sn9c8tyuk5)

[Visualizando bases de datos con Java Swing 2](#_ishdkqmodpmz)

[Transacciones 2](#_y217s7zfaqdd)

[¿Qué es una transacción? 2](#_4z0km7pn31tj)

[Implementando transacciones con JDBC 2](#_tpkv97zhtk8k)

[Conexiones PooL 3](#_a8s5qu3m23sv)

[¿Qué es una conexión pool? 3](#_xj3nx0ywpmkr)

[Implementando conexiones pool 3](#_uyfzxfmbrnfr)

[JPA y ORM 3](#_fwpapn38euh)

[¿Qué es JPA y ORM? 3](#_xggkqdht31c4)

[Instalación y configuración de Hibernate 3](#_mq7ezo32g3cg)

[Implementando JPA con Hibernete 3](#_mf30bmjf0ri6)

[Próximos pasos 3](#_oldhpp16u6c1)

[Reto: crea un clon de Netflix con Java 3](#_hqnp9sa1ejmu)

# ¿Qué es Java Database Connectivity (JDBC)?

Es un estándar de conectividad que nos permite conectar programas en java con base de datos.

JDBC (Java Database Connectivity) es una API (Application Programming Interface) de Java que proporciona un conjunto de clases e interfaces para interactuar con bases de datos relacionales, permitiendo a las aplicaciones Java acceder, manipular y gestionar los datos de manera eficiente.

Realiza conexión a una base de datos , funciona como intermediario , al usar la API de JDBC , tendremos unos beneficios establecidos que son 👍.

* Creación de consultas y declaraciones SQL personalizadas.
* Soporte a múltiples bases de datos.
* Portabilidad, se puede usar en cualquier plataforma siempre y cuando soporte la API de JDBC.

¿Qué nos permite la API de JDBC ?

* Establecer conexión con la base de datos.
* Enviar consultas
* Recibir resultados

Nos permite:

Creación de consultas y declaraciones SQL personalizadas:

JDBC (Java Database Connectivity) permite a los desarrolladores de aplicaciones Java crear consultas y declaraciones SQL personalizadas de manera eficiente y flexible.

Soporte a múltiples bases de datos:

JDBC es una especificación estándar de Java que proporciona un conjunto común de interfaces para interactuar con diferentes bases de datos. Esto significa que puedes usar JDBC para conectarte y comunicarte con una amplia gama de bases de datos, como Oracle, MySQL, PostgreSQL, SQL Server y más. JDBC ofrece controladores específicos para cada base de datos, que actúan como una capa de abstracción y manejan las diferencias específicas de cada sistema de gestión de bases de datos.

Portabilidad:

Uno de los principales beneficios de JDBC es su portabilidad. Al seguir la especificación estándar de JDBC, puedes escribir código Java que se conecte y acceda a bases de datos utilizando JDBC, y este código se puede ejecutar en cualquier plataforma que admita JDBC. Esto incluye sistemas operativos como Windows, macOS y Linux, así como diferentes servidores de aplicaciones y entornos de desarrollo.

Lecturas recomendadas

<https://docs.oracle.com/javase/tutorial/jdbc/basics/index.html>

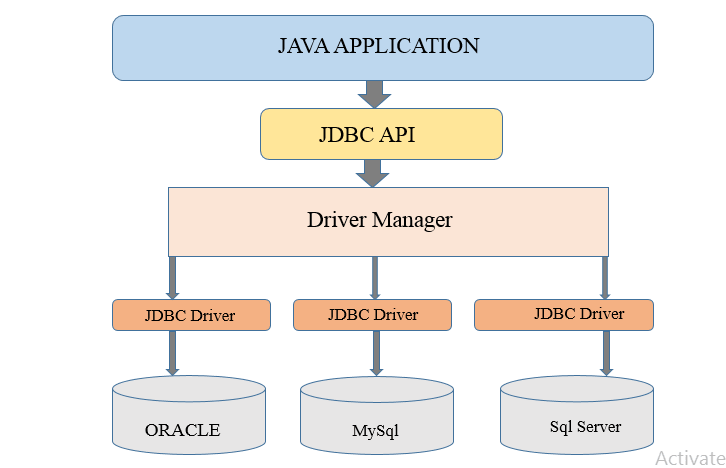
# Arquitectura de la API JDBC

JDBC Driver: El JDBC Driver es un componente de software que permite la conexión y la comunicación entre una aplicación Java y una base de datos. Actúa como un puente entre la aplicación y el sistema de gestión de bases de datos (SGBD). Cada SGBD tiene su propio controlador JDBC específico que debe ser utilizado para establecer la conexión y realizar operaciones en la base de datos. El JDBC Driver es responsable de traducir las llamadas de la API JDBC a instrucciones SQL comprensibles por el SGBD, así como de enviar y recibir resultados de las consultas.

JDBC Driver Manager: El JDBC Driver Manager es una clase proporcionada por el JDK (Java Development Kit) que actúa como un administrador de controladores JDBC. Su función principal es administrar y seleccionar el controlador adecuado para establecer la conexión con la base de datos. El Driver Manager carga dinámicamente los controladores disponibles en tiempo de ejecución y permite a la aplicación seleccionar el controlador apropiado según la URL de conexión proporcionada. También se encarga de abrir y cerrar la conexión con la base de datos.

Origen de datos (Data Source): El origen de datos representa la base de datos con la que se comunica la aplicación Java a través de JDBC. Puede ser una instancia específica de un SGBD o un conjunto de conexiones configuradas para acceder a una base de datos.

En resumen, el JDBC Driver es responsable de la comunicación directa con la base de datos, el Driver Manager administra los controladores y selecciona el adecuado, y el origen de datos representa la base de datos.



Lecturas recomendadas

IBM Documentation

<https://www.ibm.com/docs/es/psfa/7.1.0?topic=db-architecture-odbc-jdbc>

# Métodos y componentes de JDBC

Comprender cómo utilizar adecuadamente los Statements, Result Sets, Connections y el Driver Manager es fundamental para el manejo de bases de datos en Java al utilizar estos conceptos de manera conjunta, puedes establecer una conexión con una base de datos, crear Statements para ejecutar consultas SQL y obtener Result Sets con los datos resultantes. Esto te permite realizar operaciones como recuperar, insertar, actualizar o eliminar datos.

JDBC Connection: Carga lo necesario para establecer conexión: URL de la BD, usuario y contraseña.

JDBC Statement: Se utiliza para enviar consultas SQL simples sin parámetros.

JDBC PreparedStatement: Nos permite crear consultas SQL con parámetros variables.

JDBC ResultSet: Para obtener los resultados de una consulta SQL una BD.

Estos métodos nos brindan las herramientas necesarias para interactuar con una base de datos mediante JDBC, desde la ejecución de consultas hasta la manipulación y obtención de datos.

ExecuteQuery( ): Ejecuta una consulta SQL en la base de datos y devuelve un objeto ResultSet con los resultados.

next( ): Avanzar al siguiente registro en un conjunto de resultados.

close( ): Cerrar una conexión con la BD.

executedUpdate( ): Ejecutar sentencia SQL que modifica los datos en la BD y devuelve el número de filas afectadas.

get( ): Para obtener valores de los registros recuperados de la BD, existen varios métodos.

getString( ): para recuperar cadenas de texto.

getInt( ): para recuperar números enteros.

getDouble( ): para recuperar números con decimales de doble precisión.

getFloat( ): para recuperar números con decimales de precisión simple.

getBoolean( ): para recuperar valores booleanos.

getDate( ): para recuperar valores de fecha.

getTime( ): para recuperar valores de tiempo.

getTimestamp( ): para recuperar valores de fecha y tiempo.

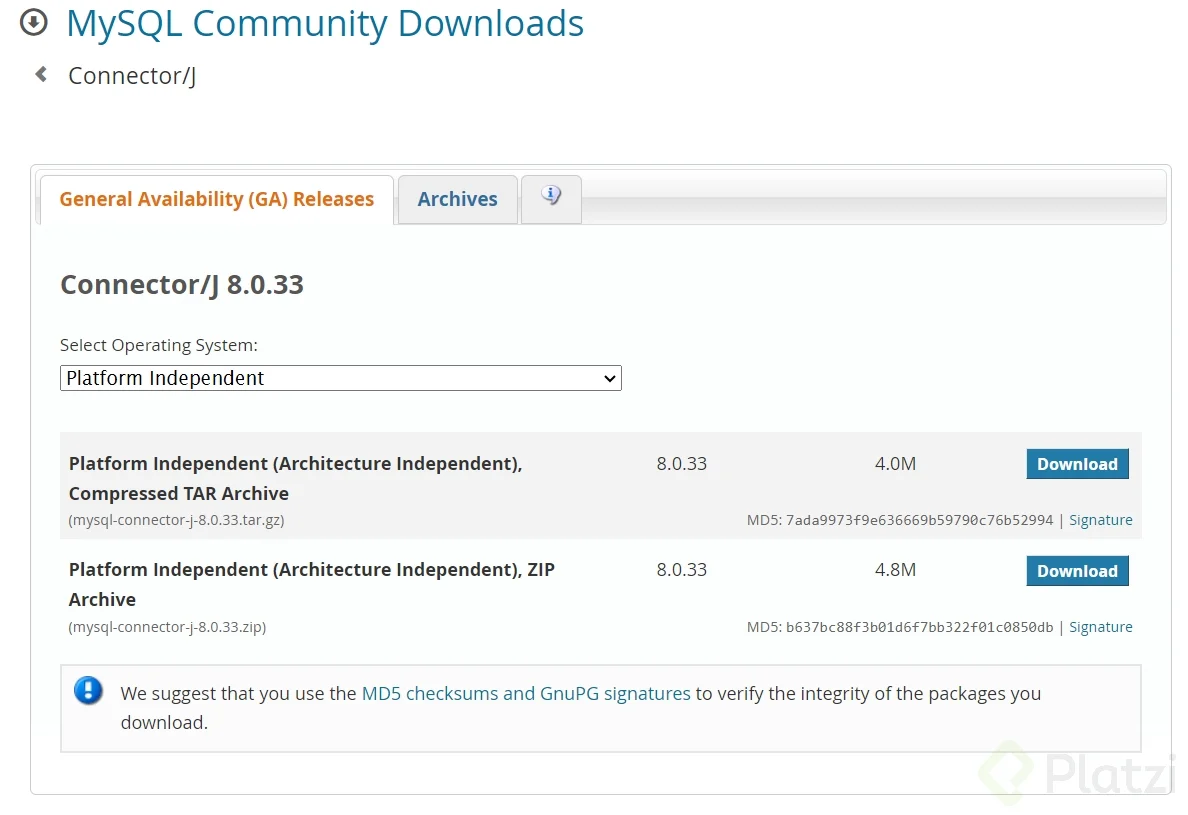
JDBC Connection: Carga lo necesario para establecer conexión: URL de la BD, usuario y contraseña.

JDBC Statement: Se utiliza para enviar consultas SQL simples sin parámetros.

JDBC PreparedStatement: Nos permite crear consultas SQL con parámetros variables.

JDBC ResultSet: Para obtener los resultados de una consulta SQL una BD

Conectando Java con bases de datos



Vamos a descargar el .jar para realizar la conexión de nuestra base de datos y luego agregaremos la ruta del driver a nuestro IDE.

File > Project Structure > Dependencies > JARs

Error en zona horaria: En la conexión agregar ?serverTimezone=UTC

Declaración de variables: Se declara una variable myConn del tipo Connection y se inicializa con el valor null. Esta variable se utilizará para establecer y mantener la conexión con la base de datos.

Bloque try-catch: El código envuelve el bloque de código que intenta establecer la conexión con la base de datos dentro de un bloque try-catch. Esto permite capturar y manejar posibles excepciones que puedan ocurrir durante la conexión.

Establecimiento de la conexión: Dentro del bloque try, se utiliza el método getConnection de la clase DriverManager para establecer la conexión a la base de datos MySQL. Los argumentos proporcionados al método son la URL de conexión (jdbc:mysql://localhost:3306/nombre de tu proyecto), el nombre de usuario (root) y la contraseña.

MySQL es un sistema de gestión de bases de datos muy utilizado y es relativamente fácil de instalar. Sigue estos pasos:

Descarga MySQL Installer:

Visita el sitio web oficial de MySQL para descargar el instalador: Descargar MySQL Installer.

<https://downloads.mysql.com/archives/community/>

Selecciona la versión adecuada:

Descarga la última versión de MySQL Installer para Windows (por lo general, la que está en la parte superior de la lista).

Puedes optar por la versión de instalador web (Online) o la versión completa (Offline). La versión web descarga los componentes a medida que los necesites, mientras que la versión completa ya los incluirá en la descarga.

Ejecuta el Instalador:

Una vez que se haya descargado el instalador, ejecútalo haciendo doble clic en el archivo descargado (por ejemplo, mysql-installer-web-community-xxx.xxx.xxx.xxx.msi).

Selección de Productos:

Aparecerá una ventana que te permitirá seleccionar los productos a instalar. Para la mayoría de los casos, elige “MySQL Server” y, si lo deseas, puedes seleccionar otros componentes como MySQL Workbench (una herramienta de administración).

Descargar Componentes:

Si elegiste la versión web del instalador, comenzará a descargar los componentes seleccionados. Esto puede llevar un tiempo, así que ten paciencia.

Configuración de MySQL Server:

Durante la instalación, se te pedirá que configures MySQL Server. Puedes elegir la configuración por defecto, que suele ser suficiente para la mayoría de los usuarios, o personalizarla según tus necesidades.

Establecer Contraseña de Root:

Durante la instalación, se te pedirá que establezcas una contraseña para el usuario “root” de MySQL. Asegúrate de recordar esta contraseña, ya que la necesitarás para administrar MySQL.

Completar la Instalación:

Una vez que se completen todos los pasos de configuración, haz clic en “Execute” (Ejecutar) para que la instalación continúe. Luego, haz clic en “Finish” (Finalizar) para completar la instalación.

Verifica la Instalación:

Puedes verificar si MySQL se instaló correctamente abriendo MySQL Workbench o usando la línea de comandos de MySQL.

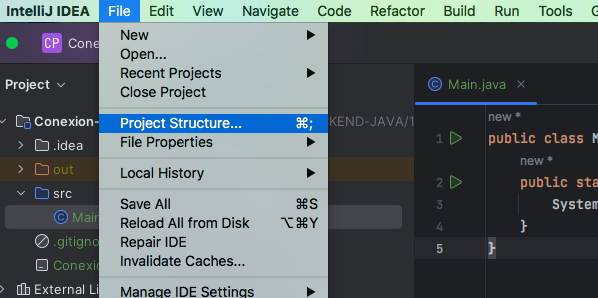
Para abrir MySQL Workbench, búscalo en tu menú de inicio o ejecuta el programa si se creó un acceso directo en tu escritorio durante la instalación.

Para usar la línea de comandos, abre una ventana de comandos (cmd) y ejecuta mysql -u root -p. Te pedirá la contraseña que estableciste durante la instalación.

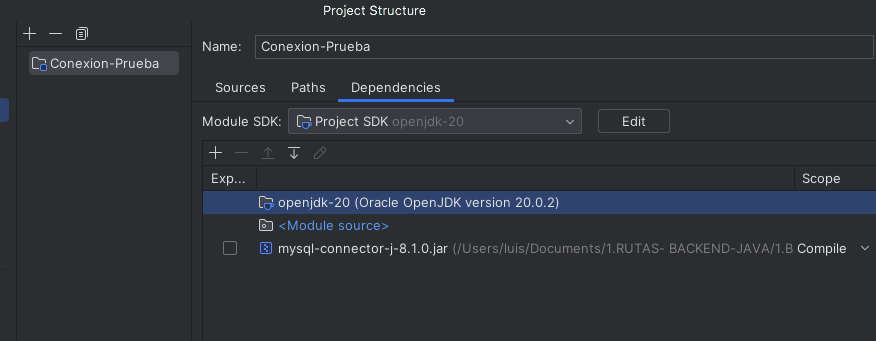
Eso es todo, has instalado MySQL en tu PC con Windows 10. Ahora puedes comenzar a trabajar con bases de datos MySQL.

MACOS

<https://dev.mysql.com/downloads/file/?id=520816>



vamos a realizar la configuración de la estructura.



Instalador MYSQL COMMUNITY HIGH SIERRA

<https://downloads.mysql.com/archives/community/>

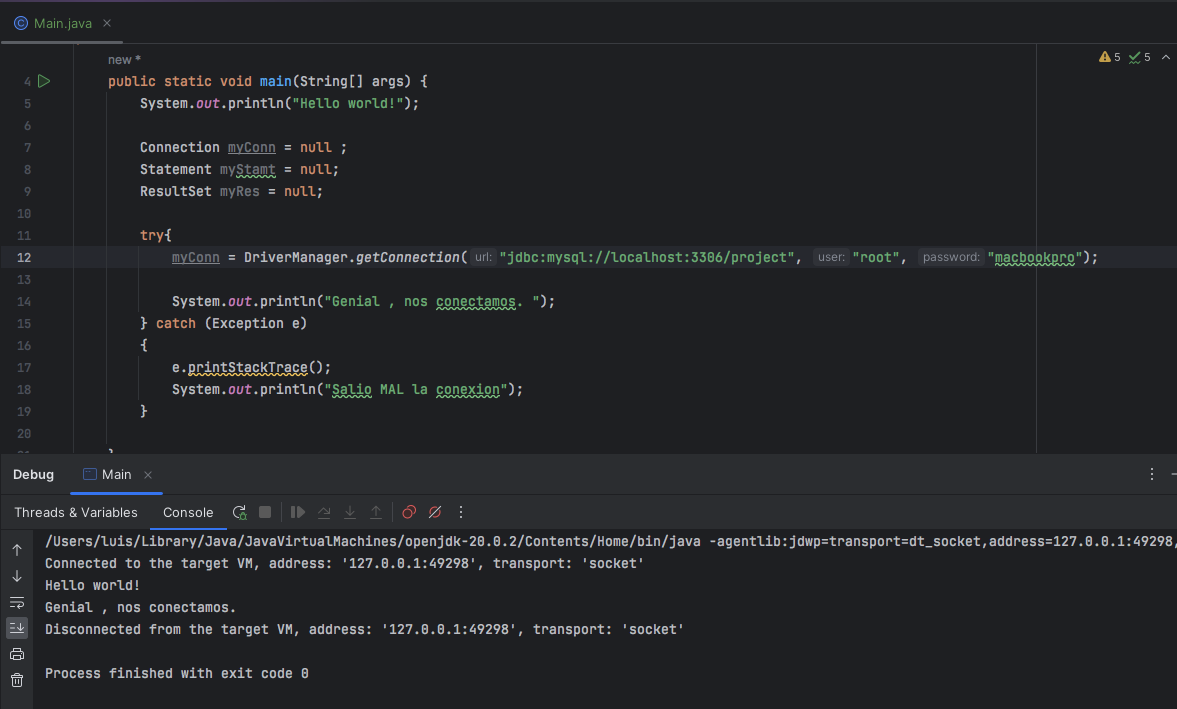
root:user

pass:macbookpro

workbench para sql Instalador MYSQL COMMUNITY HIGH SIERRA

<https://downloads.mysql.com/archives/workbench/>

CREATE SCHEMA `project` DEFAULT CHARACTER SET DEFAULT ;



CREATE TABLE `project`.`employees` (

`nombre` INT NOT NULL,

PRIMARY KEY (`nombre`));

ALTER TABLE `project`.`employees`

CHANGE COLUMN `nombre` `nombre` VARCHAR(50) NOT NULL;

-- Insertar 5 nombres ficticios

INSERT INTO `project`.`employees` (`nombre`)

VALUES

('John Doe'),

('Jane Smith'),

('Michael Johnson'),

('Emily Davis'),

('Robert Brown');



# Leer y crear datos con JDBC

En este paso crearemos nuestras primeras consultas .

* Declaro mis variables de tipo connection, statement, resultset.
* establecer conexión.
* cerrar mi objeto statement y ejecutar mi consulta SQL
* procesar los resultados.

vamos a crear un preparedstatement para insertar datos a nuestra base de datos .





# Actualizar y eliminar datos con JDBC

Para eliminar datos usaremos la siguiente configuración de nuestro programa.

Actualizar

// Crear nuestros clientes con nombres

String sql = "INSERT INTO employees(nombre) VALUES (?)";

myPrep = myConn.prepareStatement(sql);

myPrep.setString(1, "Luis");

myPrep.addBatch(); // Agregar el primer cliente (Luis)

myPrep.setString(1, "Maria");

myPrep.addBatch(); // Agregar el segundo cliente (Maria)

int[] rowsAffected = myPrep.executeBatch();

if (rowsAffected.length > 0) {

System.out.println("Empleado(s) creado(s) correctamente.");

}

Eliminar un EMPLEADO



# CRUD

# Clases y estructura del proyecto con IntelliJ y Maven

Una vez que se haya creado el proyecto Maven, agrega el conector JDBC de MySQL:

Manualmente:

Abre el archivo “pom.xml” en el proyecto Maven.

Busca el enlace al conector JDBC de MySQL en el repositorio Maven. Puedes utilizar el siguiente enlace: https://mvnrepository.com/artifact/com.mysql/mysql-connector-j \*\*\*\*En este repositorio, se encuentran alojados una gran cantidad de artefactos, como librerías, frameworks y plugins de Maven, que pueden ser utilizados en proyectos de desarrollo de software Java.

Copia la dependencia XML del conector JDBC de MySQL desde el enlace.

Pega la dependencia dentro del bloque de dependencias en el archivo “pom.xml”.

Guarda el archivo “pom.xml”.

Después de seguir estos pasos, tu proyecto Maven estará configurado con el conector JDBC de MySQL y podrás utilizarlo en tu aplicación para interactuar con una base de datos MySQL.

Desde el IDE:

IntelliJ IDEA proporciona una forma más sencilla de agregar dependencias utilizando el atajo de teclado “Alt + Insert” en proyectos de Maven.

Abre tu proyecto en IntelliJ IDEA.

Navega hasta el archivo pom.xml en la estructura del proyecto.

Haz clic con el botón derecho del ratón en el archivo pom.xml y selecciona “Open ‘pom.xml’” en el menú contextual.

En el editor de pom.xml, coloca el cursor en el lugar donde deseas agregar la dependencia. Por lo general, se coloca dentro del elemento <dependencies>.

Luego, presiona el atajo de teclado “Alt + Insert” (o “Ctrl + Enter” en macOS).

En el menú emergente, selecciona “Dependency” o “Add Dependency” según la versión de IntelliJ IDEA que estés utilizando.

Aparecerá un cuadro de diálogo llamado “Add Dependency”. En este cuadro de diálogo, puedes buscar la dependencia que deseas agregar escribiendo su nombre en el campo de búsqueda.

A medida que escribas, IntelliJ IDEA buscará en el repositorio Maven y mostrará una lista de coincidencias. Selecciona la dependencia adecuada de la lista.

Una vez que hayas seleccionado la dependencia, haz clic en el botón “OK” para agregarla al archivo pom.xml.

IntelliJ IDEA descargará automáticamente la dependencia y actualizará el archivo pom.xml con la nueva entrada de dependencia. Además, podrás ver el progreso de la descarga en la barra de estado de IntelliJ IDEA.

Lecturas recomendadas

Just a moment...

https://mvnrepository.com/artifact/com.mysql/mysql-connector-j

https://www.arquitecturajava.com/que-es-un-java-maven-artifact/

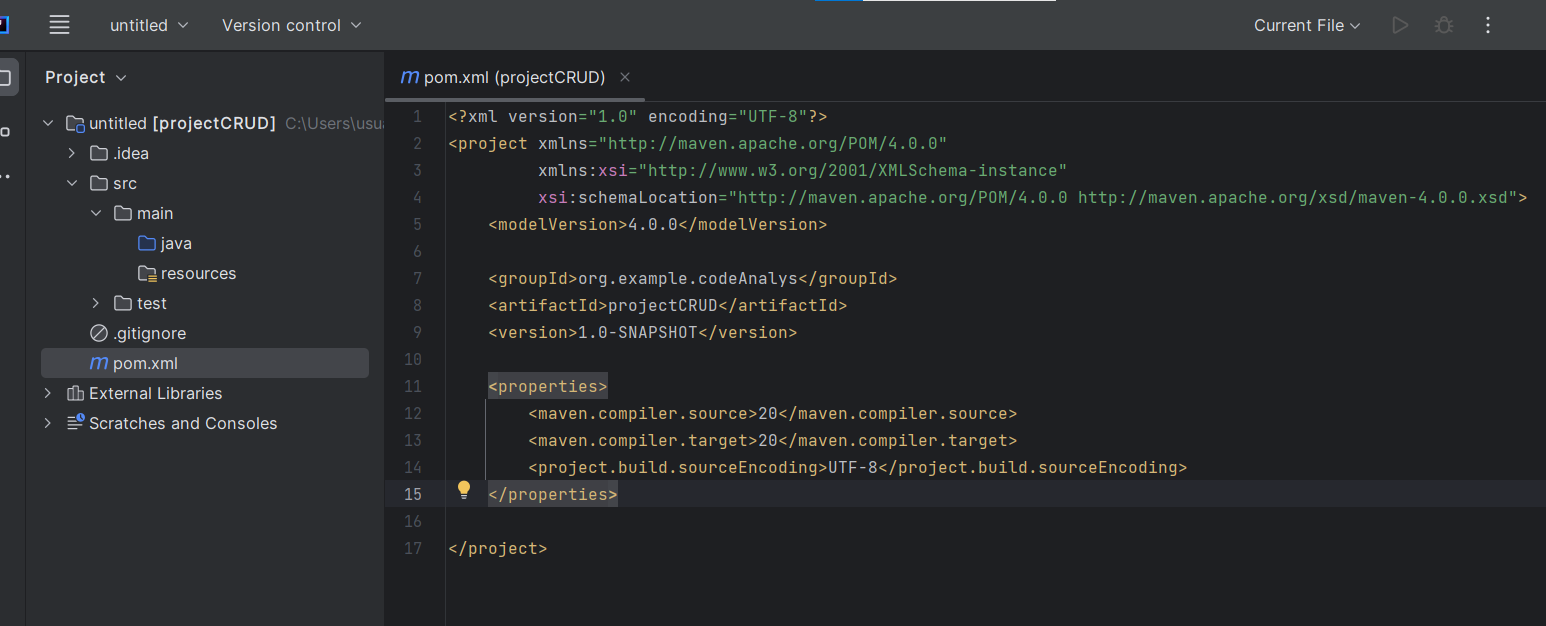
https://www.arquitecturajava.com/que-es-un-java-maven-artifact/

¿Qué es un Java Maven Artifact ? - Arquitectura Java

https://www.arquitecturajava.com/que-es-un-java-maven-artifact/

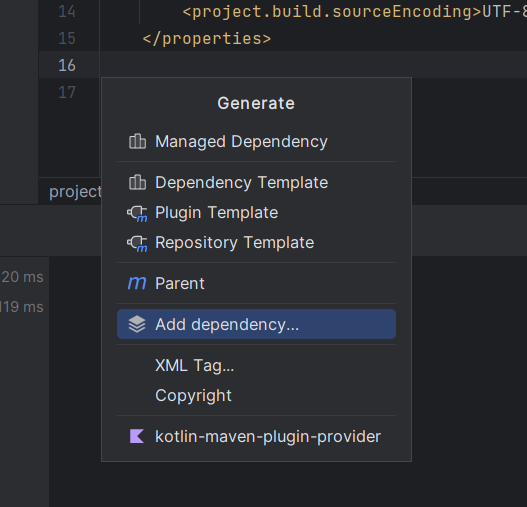
Vamos a crear nuestra conexión y a crear el proyecto usando maven.

Lo primero que vamos a realizar es crear el projectoCRUD con maven.

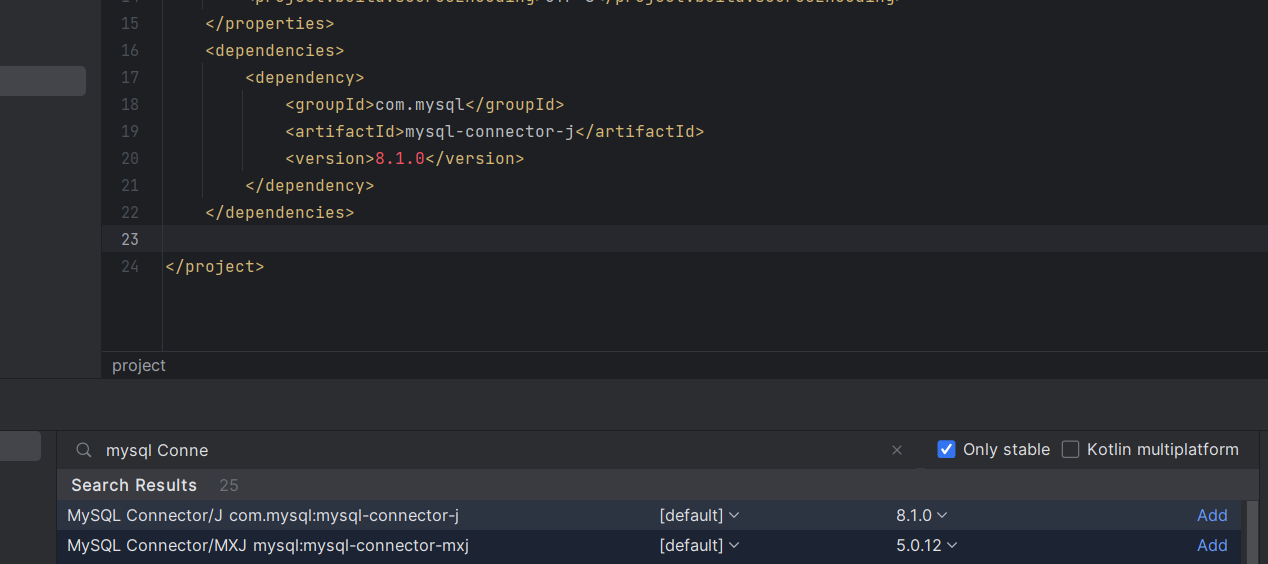


segundo nos vamos al archivo pom.xml y es aquí donde cargaremos las dependencias de nuestro proyecto.

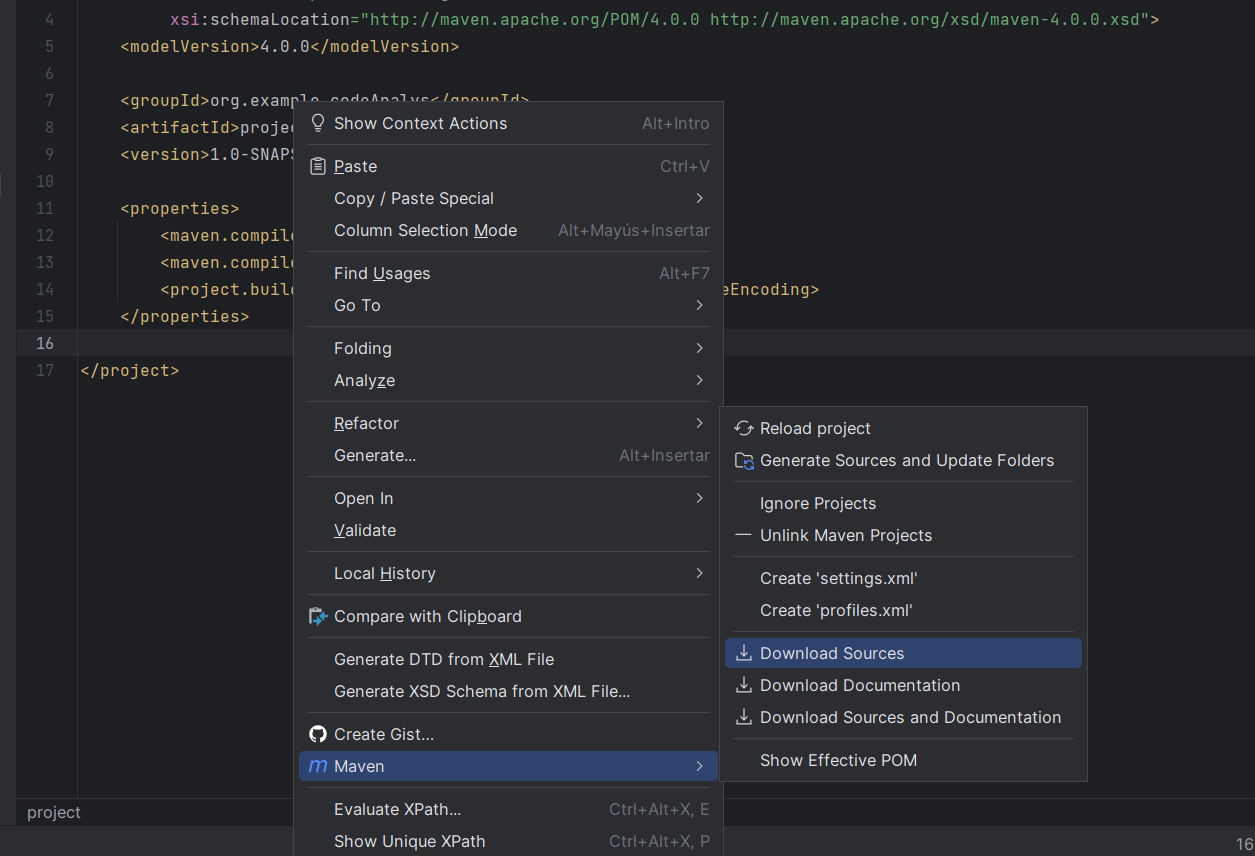
Clic derecho , Genereate y luego add Dependency

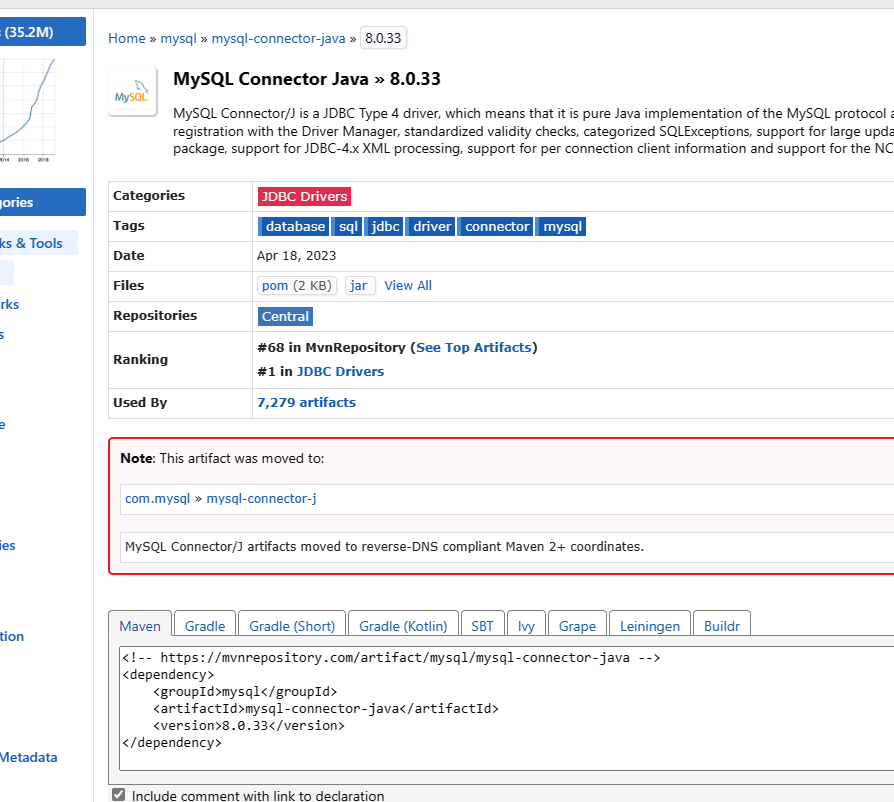


Agregaremos la dependencia MySQL connection

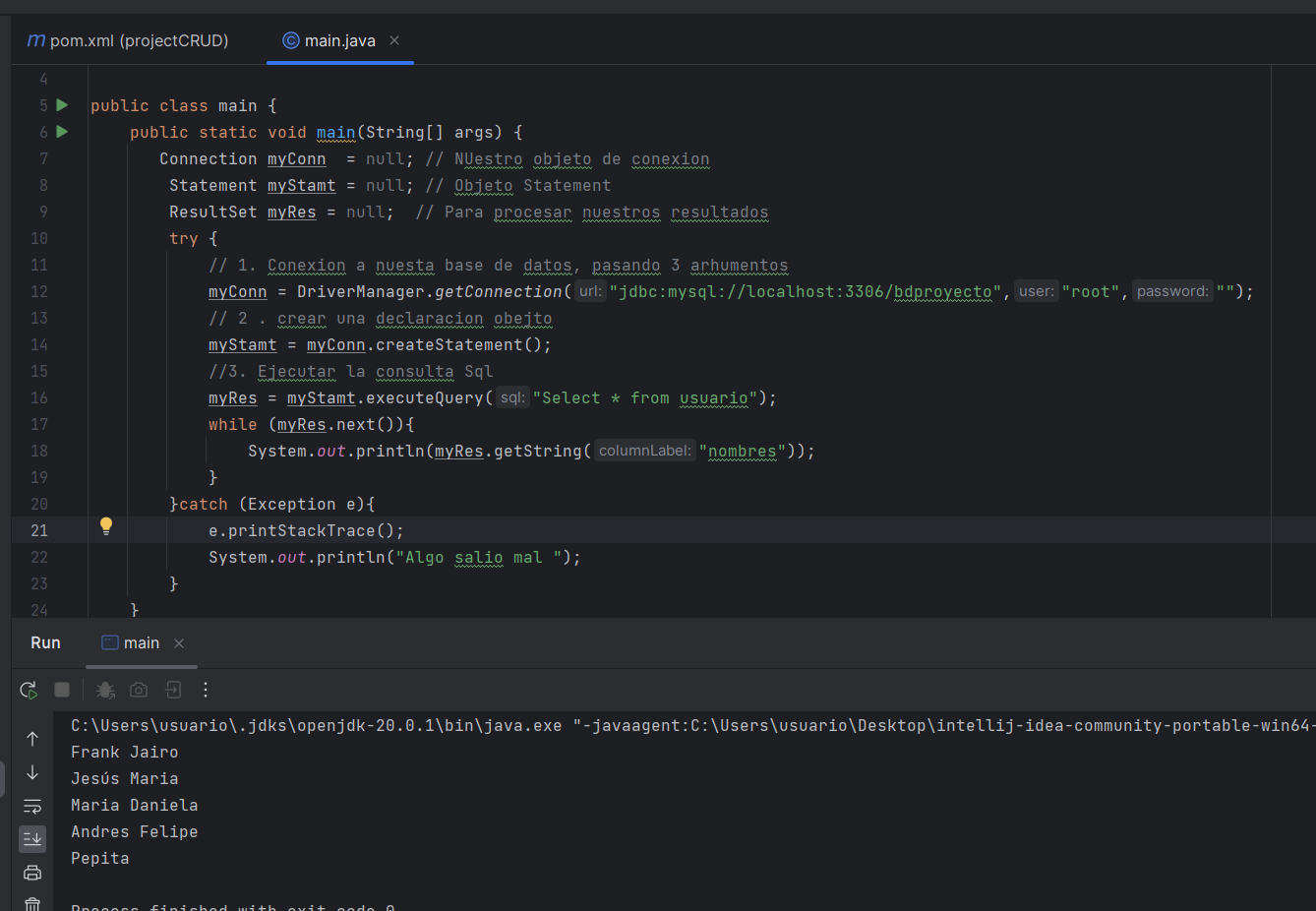


La otra forma es ir directamente al navegador y asignar la dependencia directamente desde la página.





Después de crear la conexión con las dependencias vamos a crear una clase llamada main , la cual se encargará de revisar que nuestra conexión esté bien establecida.



| Connection myConn = null; // NUestro objeto de conexion Statement myStamt = null; // Objeto Statement ResultSet myRes = null; // Para procesar nuestros resultados try { // 1. Conexion a nuesta base de datos, pasando 3 arhumentos myConn = DriverManager.getConnection("jdbc:mysql://localhost:3306/bdproyecto","root",""); // 2 . crear una declaracion obejto myStamt = myConn.createStatement(); //3. Ejecutar la consulta Sql myRes = myStamt.executeQuery("Select \* from usuario"); while (myRes.next()){ System.out.println(myRes.getString("nombres")); } }catch (Exception e){ e.printStackTrace(); System.out.println("Algo salio mal "); } |
| --- |

# Optimización de recursos

Es una buena práctica cerrar los recursos con el método close() después de usarlos para liberar los recursos utilizados. Esto se aplica a la programación JDBC con Java, así como a cualquier otra situación en la que se utilicen recursos del sistema, como archivos o sockets. Una forma común de hacerlo es usando el bloque finally para asegurarnos de que los recursos se cierren incluso si se produce una excepción en el código.

Otra forma introducida en Java 7 es usar la característica “Try with resources” que implementan la interfaz AutoCloseable o su subinterfaz Closeable. Antes de la introducción del “try con recursos”, era necesario cerrar manualmente los recursos abiertos utilizando bloques finally para asegurarse de que se liberaran correctamente, incluso en caso de excepciones. Sin embargo, esto puede ser propenso a errores y aumenta la cantidad de código necesario. Al utilizar esta estructura, los recursos declarados dentro del bloque try se cierran automáticamente al finalizar el bloque, ya sea que se haya producido una excepción o no.

Si tienes una clase que implementa la interfaz AutoCloseable o Closeable, no es necesario agregar manualmente la interfaz AutoCloseable en el código.

Lecturas recomendadas

The try-with-resources Statement (The Java™ Tutorials > Essential Java Classes > Exceptions) body { margin-left:10px; margin-right:10px; line-height: 1.5; FONT-FAMILY: Arial, Helvetica, sans-serif; font-size: 0.8em; } a:link{text-decoration:none; col

https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html

Chapter 14. Blocks and Statements

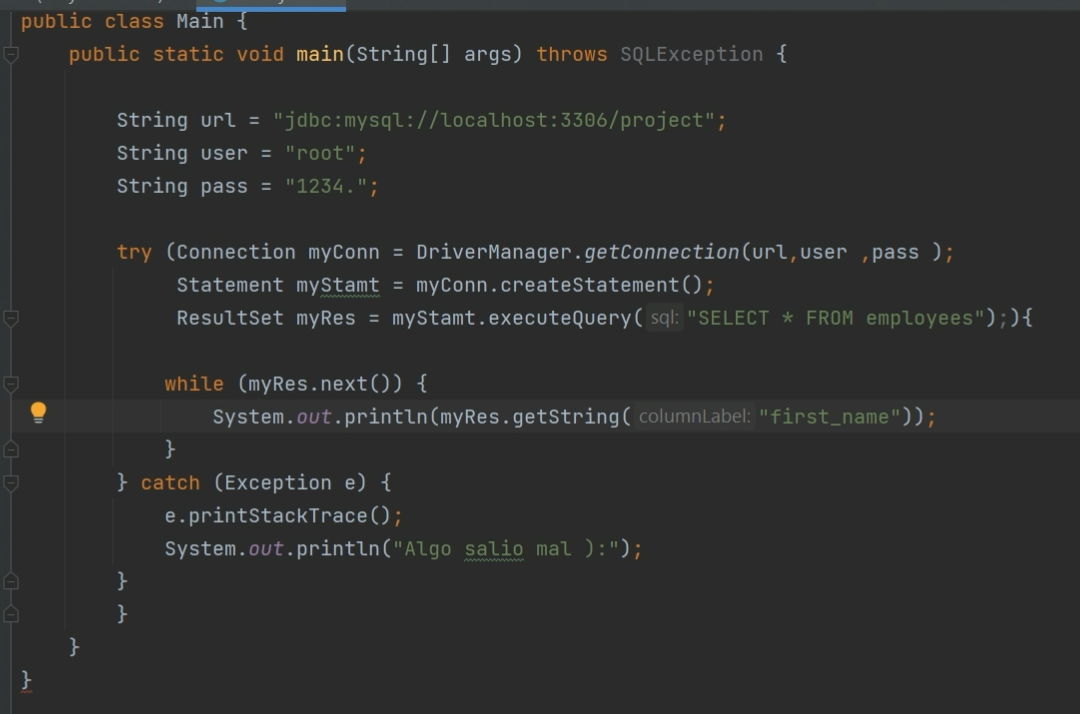
https://docs.oracle.com/javase/specs/jls/se8/html/jls-14.html#jls-14.20.3

Método close()

cerrar los recursos con el método close () después de usarlos para liberar los recursos utilizados.

Necesitamos utilizar de forma más eficiente nuestros recursos al aplicar las variables close()





# Patrón DAO y Repository

El patrón DAO (Data Access Object) y el patrón Repository son dos patrones de diseño utilizados en el desarrollo de aplicaciones para separar la lógica de acceso a datos de la lógica de negocio.

Patrón DAO:

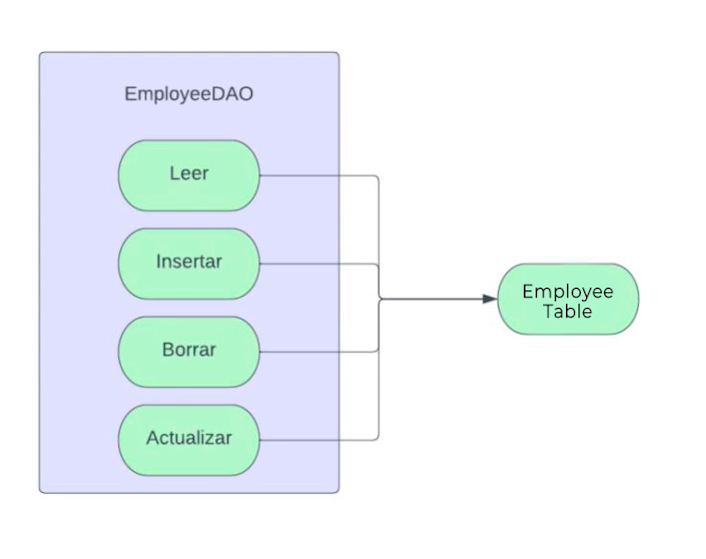
Se centra en proporcionar una capa de abstracción entre la capa de negocio y la capa de acceso a datos.

El DAO define una interfaz o una clase abstracta que encapsula la lógica de acceso a datos.

Proporciona métodos para crear, leer, actualizar y eliminar (CRUD).

El DAO oculta los detalles específicos de implementación del almacenamiento de datos y proporciona una interfaz coherente para que la capa de negocio interactúe con los datos.

Permite centralizar y reutilizar la lógica de acceso a datos en toda la aplicación.



Patrón Repository:

Se centra en separar la lógica de acceso a datos (capa de persistencia de datos) de la lógica de negocio.

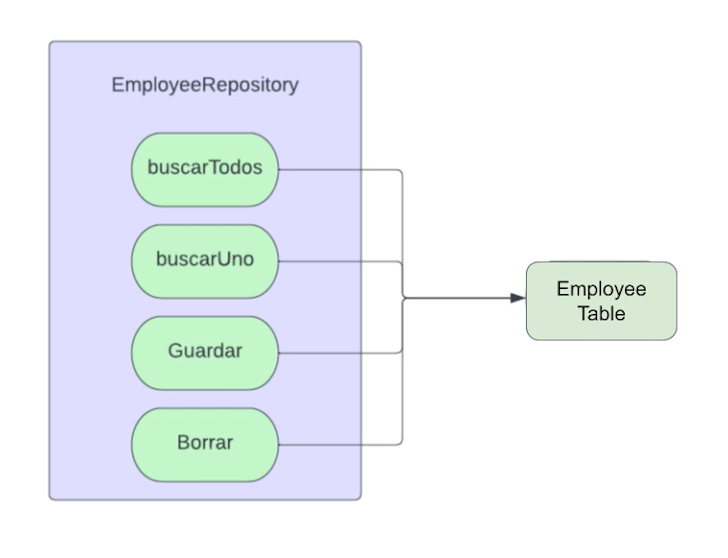
El Repository define una interfaz o una clase abstracta que proporciona métodos para recuperar y almacenar.

Proporciona métodos más específicos para leer, guardar y borrar.

El Repository oculta los detalles específicos de almacenamiento y consulta, y proporciona una abstracción para interactuar con los objetos.

Permite un enfoque más orientado a objetos para trabajar con los datos, tratando los objetos como colecciones en lugar de registros aislados.

Podemos concluir que ambos patrones tienen como objetivo separar la lógica de acceso a datos de la lógica de negocio, nos proporcionan una abstracción para interactuar con los datos y ocultan los detalles específicos de almacenamiento y consulta, además de que nos facilitan la reutilización y centralización de la lógica de acceso a datos.



# Implementando el Patrón Repository

En la implementación del patrón, creamos varias clases principales en nuestro proyecto.

En nuestro paquete util, tenemos la clase DatabaseConnection, que se encarga de manejar la lógica de la conexión a la base de datos. Utilizamos el patrón Singleton para garantizar que solo tengamos una instancia de conexión en todo el programa. La clase DatabaseConnection expone un método getInstance() que nos permite obtener la instancia de la conexión en cualquier parte del programa.

En nuestro paquete model, tenemos la clase Employee, que representa el modelo de datos para los empleados. En esta clase, definimos los atributos de los empleados, así como los métodos getter, setter, toString y los constructores (tanto el constructor vacío como el constructor que recibe los datos del empleado).

Por último, tenemos una interfaz Repository que define los métodos comunes para realizar las operaciones CRUD (Create, Read, Update, Delete) y que vamos a desarrollar en nuestra clase EmployeeRepository. Esta interfaz utiliza un tipo genérico para permitir que pueda ser implementada por diferentes clases y trabajar con diferentes tipos de datos. Los métodos comunes que se definen en esta interfaz son findAll(), getById(), save() y delete(), que representan las operaciones de búsqueda de todos los elementos, búsqueda por ID, guardado y eliminación, respectivamente.

```````findAll\*\*(): Este método devuelve una lista de elementos de tipo \*\*T`. Es utilizado para obtener todos los elementos almacenados.

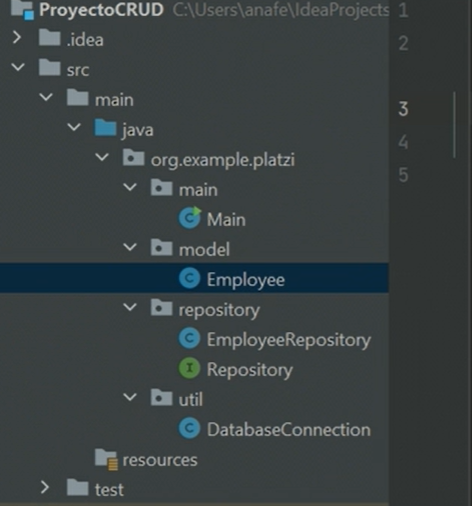
getById(Integer id): Este método recibe un identificador (id) y devuelve un elemento de tipo T que corresponde a ese identificador.

save(T t): Este método recibe un objeto de tipo T y lo guarda. Es utilizado para insertar o actualizar un elemento.

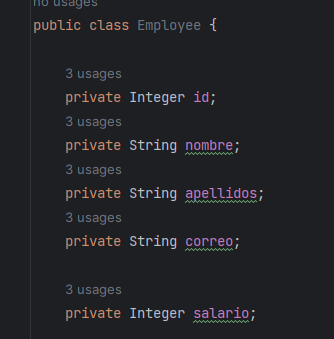
delete(Integer id): Este método recibe un identificador (id) y elimina el elemento correspondiente a ese identificador.

La letra T es una convención comúnmente utilizada en la programación genérica para representar un tipo genérico. Permite una mayor flexibilidad y reutilización del código, ya que la interfaz Repository puede ser implementada para trabajar con diferentes tipos de objetos sin tener que especificar un tipo concreto en el momento de su definición.

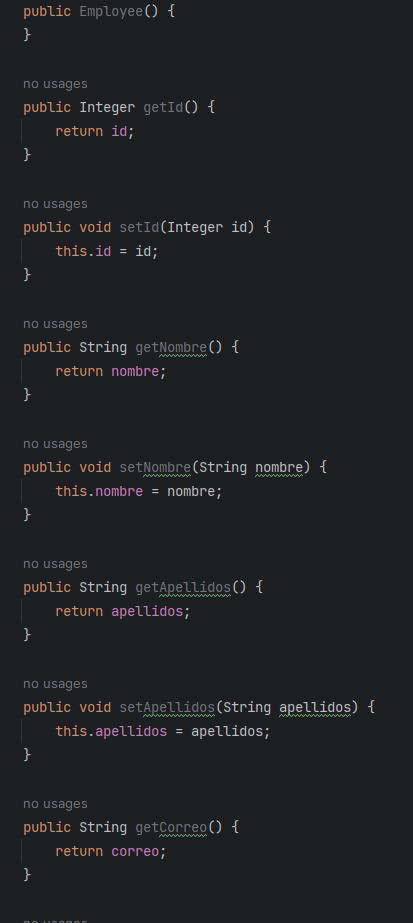
Crearemos la estructura de nuestro programa.



Vamos a crear la estructura y luego los atributos de la clase empleado



Posterior a esto crearemos los métodos de esta clase



Después de crear los métodos Getter y setter crearemos la conexión a nuestra base de datos, esto lo realizaremos en database connection.

| package code.util;  import java.sql.Connection;  import java.sql.DriverManager;  import java.sql.SQLException;  public class DatabaseConnection {  private static String *url* = "jdbc:mysql://localhost:3306/bdproyecto2";  private static String *user* = "root";  private static String *pass* = "";  private static Connection *myConn* ;  public static Connection getInstance() throws SQLException{  if (*myConn* == null){  *myConn* = DriverManager.*getConnection*(*url*,*user*,*pass*);  }  return *myConn*;  }  } |
| --- |

CREAMOS LA CONEXIÓN

Vamos a modificar el main.

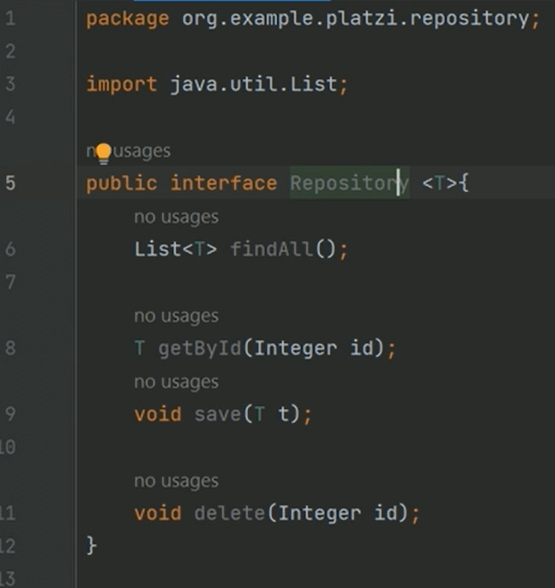


| public class main {  public static void main(String[] args) throws SQLException {  try ( Connection myConn = DatabaseConnection.*getInstance*();  Statement myStamt = myConn.createStatement();  ResultSet myRes = myStamt.executeQuery("SELECT \* FROM usuarios");){  while (myRes.next()){  System.*out*.println(myRes.getString("nombre"));  }  }catch (Exception e){  e.printStackTrace();  System.*out*.println("Algo salio mal ");  }  }  } |
| --- |

# **Leer Datos desde Java**

En el paquete repository, creamos la clase EmployeeRepository que implementa la interfaz Repository. En esta clase, implementamos todos los métodos definidos en la interfaz y comenzamos a codificar cada uno de ellos.

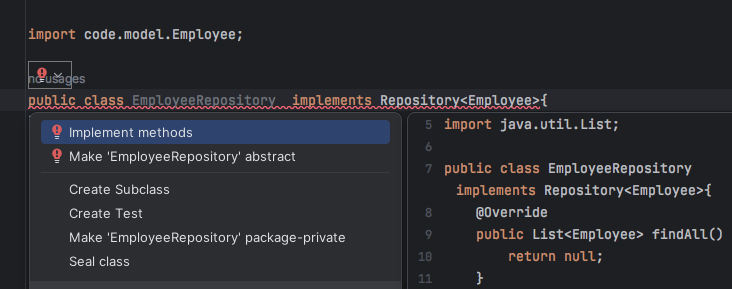
Comenzamos con los métodos findAll() y getById(), que corresponden a las operaciones de lectura en el CRUD. Aprendimos a extraer un método utilizando la opción “Refactor” > “Extract Method” en nuestro IDE. Esta técnica nos permitió reutilizar código y evitar repetir la lógica de asignación de valores a través de los métodos set de la clase Employee. En su lugar, creamos el método createEmployee, que nos ayuda a encapsular esa lógica y mejorar la legibilidad y mantenibilidad del código.

Crearemos estos métodos en la implementación 

estamos creando los métodos del CRUD que posteriormente los instanciamos para que se cargue a nuestra base de datos.

| List<T> findAll();  T getById(Integer id);  void save(T t);  void delete (Integer id); |
| --- |

Vamos a implementar los métodos , esto lo realizaremos al momento de cargar los datos a de nuestro employrepository .

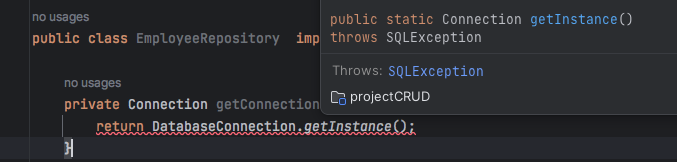


implementamos los metodos de employee

| package code.repository;  import code.model.Employee;  import java.util.List;  public class EmployeeRepository implements Repository<Employee>{  @Override  public List<Employee> findAll() {  return null;  }  @Override  public Employee getById(Integer id) {  return null;  }  @Override  public void save(Employee employee) {  }  @Override  public void delete(Integer id) {  }  } |
| --- |

PARAMETRIZACIÓN DEL MÉTODO findALL

creamos la conexion y luego la casteamos en el metodo de conexion

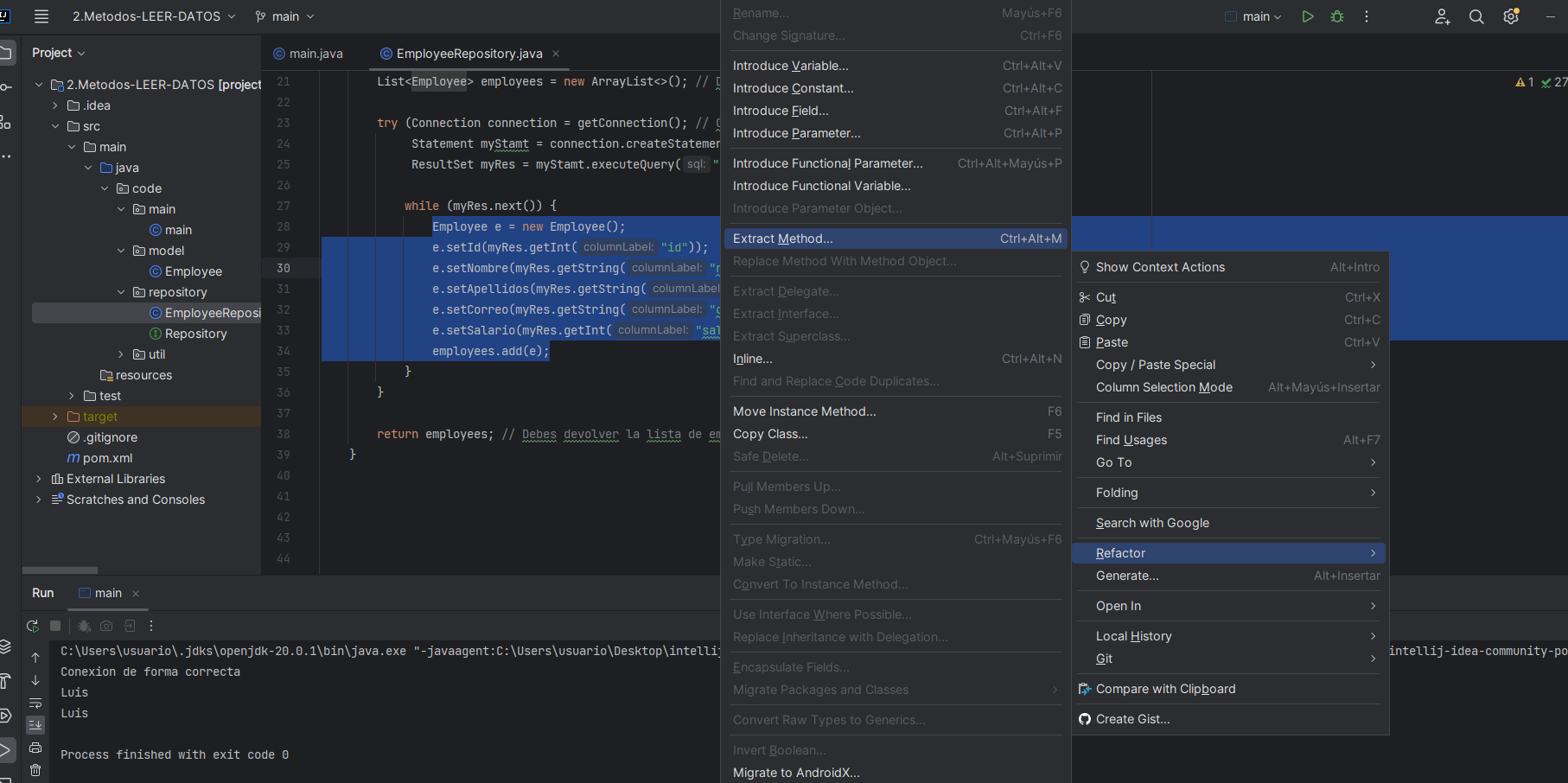


| private Connection getConnection() throws SQLException {  return DatabaseConnection.*getInstance*(); // Agregué ".getConnection()"  } |
| --- |

| @Override  public List<Employee> findAll() throws SQLException {  List<Employee> employees = new ArrayList<>(); // Debes crear una lista para almacenar los empleados.  try (Connection connection = getConnection(); // Obtener la conexión correctamente  Statement myStamt = connection.createStatement();  ResultSet myRes = myStamt.executeQuery("SELECT \* FROM empleados")) {  while (myRes.next()) {  Employee e = new Employee();  e.setId(myRes.getInt("id"));  e.setNombre(myRes.getString("nombre")); // Cambiado a getString para obtener una cadena.  e.setApellidos(myRes.getString("apellidos")); // Cambiado a getString.  e.setCorreo(myRes.getString("correo")); // Cambiado a getString.  e.setSalario(myRes.getInt("salario")); // Cambiado a getDouble para obtener un número decimal.  employees.add(e);  }  }  return employees; // Debes devolver la lista de empleados.  } |
| --- |

Vamos a extraer el método.

Clic derecho, reflector extraer método.



Después de organizar el método de búsqueda vamos a realizar el método de actualización.



| try ( Connection myConn = DatabaseConnection.*getInstance*()){  Repository<Employee> reopository = new EmployeeRepository();  reopository.findAll().forEach(System.*out*::println);  }  try (Connection myConn = DatabaseConnection.getInstance()) {: Este bloque de código inicia una estructura de control try-with-resources.  La parte importante aquí es la creación de una variable llamada myConn que representa una conexión a la base de datos. Esta conexión se obtiene llamando al método getInstance() de la clase DatabaseConnection. El beneficio de usar try-with-resources es que la conexión se cerrará automáticamente al final del bloque, lo que es una buena práctica para asegurarse de que las conexiones se manejen correctamente y se liberen cuando ya no se necesitan.  Repository<Employee> repository = new EmployeeRepository();: Aquí se crea una instancia de un objeto llamado repository, que se supone que es un repositorio genérico. Este repositorio se utiliza para acceder a los datos relacionados con los empleados. Específicamente, parece ser un repositorio de tipo EmployeeRepository, que se usa para realizar operaciones relacionadas con los empleados, como buscar todos los empleados en la base de datos.  repository.findAll().forEach(System.out::println);: Esta línea de código llama al método findAll() en el repositorio para recuperar todos los registros de empleados de la base de datos. Luego, utiliza forEach para iterar sobre estos registros y los imprime en la consola usando System.out::println. En resumen, esta línea muestra en la consola todos los empleados recuperados de la base de datos. |
| --- |

# **Insertar Datos en Java**

Para insertar datos vamos a irnos a la clase employeeRepository la cual es la encargada de llevar la lógica de nuestro proyecto y así mismo modificaremos el método save.

En el método save(Employee employee) implementamos la funcionalidad para insertar un nuevo registro de empleado en la base de datos.

Dentro del método, definimos una consulta SQL de inserción y la almacenamos en una variable de tipo String. Luego, utilizamos un objeto PreparedStatement para preparar la consulta y asignar los valores de los parámetros.

Una vez que todos los valores se han asignado, ejecutamos el método executeUpdate() del PreparedStatement para ejecutar la consulta y realizar la inserción en la base de datos.

@Override

public void save(Employee employee) throws SQLException {

String sql = "INSERT INTO empleados(nombre,apellidos,correo,salario) VALUES(?,?,?,?)";

try(PreparedStatement myStamt = getConnection().prepareStatement(sql)){

myStamt.setString(1,employee.getNombre());

myStamt.setString(2,employee.getApellidos());

myStamt.setString(3,employee.getCorreo());

myStamt.setInt(4,employee.getSalario());

myStamt.executeUpdate();

}

}

Este código es un método llamado `save` que se utiliza para insertar un nuevo registro en una tabla de MySQL. Está diseñado para trabajar con objetos de la clase `Employee` y utiliza una conexión a la base de datos para realizar la inserción. A continuación, te explico cada parte del código:

1. `@Override`: Esto indica que el método `save` está sobrescribiendo un método de una clase padre o interfaz. En este caso, probablemente sea una implementación de una interfaz o una clase base relacionada con la manipulación de datos.

2. `public void save(Employee employee) throws SQLException`: Este es el encabezado del método `save`. Toma un objeto `Employee` como parámetro y puede arrojar una excepción `SQLException` si ocurre un error en la base de datos.

3. `String sql = "INSERT INTO empleados(nombre,apellidos,correo,salario) VALUES(?,?,?,?)";`: Esta línea define una consulta SQL parametrizada para insertar un nuevo registro en la tabla "empleados". Los valores entre los paréntesis son marcadores de posición (`?`) que serán reemplazados por los valores reales más adelante.

4. `try (PreparedStatement myStamt = getConnection().prepareStatement(sql))`: Aquí se inicia un bloque `try-with-resources` para garantizar que los recursos se cierren correctamente después de su uso. Se crea un objeto `PreparedStatement` llamado `myStamt` que se utilizará para ejecutar la consulta SQL. `getConnection()` se supone que es un método que obtiene una conexión a la base de datos.

5. Dentro del bloque `try`, se asignan valores a los marcadores de posición en la consulta SQL utilizando los métodos `setString` y `setInt` del objeto `myStamt`. Estos métodos asignan valores a las posiciones correspondientes de los marcadores de posición en el orden en que aparecen en la consulta SQL.

- `myStamt.setString(1, employee.getNombre());`: Asigna el valor del nombre del empleado al primer marcador de posición.

- `myStamt.setString(2, employee.getApellidos());`: Asigna el valor de los apellidos del empleado al segundo marcador de posición.

- `myStamt.setString(3, employee.getCorreo());`: Asigna el valor del correo del empleado al tercer marcador de posición.

- `myStamt.setInt(4, employee.getSalario());`: Asigna el valor del salario del empleado al cuarto marcador de posición.

6. `myStamt.executeUpdate();`: Finalmente, esta línea ejecuta la consulta SQL con los valores proporcionados y realiza la inserción del nuevo registro en la tabla "empleados". `executeUpdate()` se utiliza para ejecutar consultas que modifican datos en la base de datos.

En resumen, este método `save` toma un objeto `Employee`, crea una consulta SQL parametrizada para insertar un nuevo registro en una tabla de la base de datos y luego asigna los valores del objeto `Employee` a los marcadores de posición en la consulta antes de ejecutarla para realizar la inserción.

En el método MAIN.

public static void main(String[] args) throws SQLException {

try (Connection myConn = DatabaseConnection.*getInstance*()){

Repository<Employee> reopository = new EmployeeRepository();

// System.out.println("------Listado----------");

// reopository.findAll().forEach(System.out::println);

System.*out*.println("------Insertando un empleado----------");

Employee employee = new Employee();

employee.setNombre("Diego");

employee.setApellidos("Pimentel");

employee.setCorreo("pimentel@ejemplo.com");

employee.setSalario(300000);

reopository.save(employee);

System.*out*.println("------Nuevo empleado Insertado----------");

reopository.findAll().forEach(System.*out*::println);

}catch (SQLException e) {

e.printStackTrace();

}

}

Creamos el objeto para cargarlo en nuestra base de datos.

# **Eliminar Datos en Java**

Vamos a realizar primero la modificación de nuestro método save , para darle una condicional y así poder actualizar nuestros elementos en la base de datos.

MÉTODO ACTUALIZAR

@Override

public void save(Employee employee) throws SQLException {

String sql;

// si contiene un valor numerico mayor que cero , entonces existe un id y es una actualizacion

if(employee.getId()!= null && employee.getId()>0){

sql = "UPDATE empleados SET nombre = ?, apellidos = ?, correo = ?, salario = ? WHERE id = ?";

}else{

sql = "INSERT INTO empleados(nombre,apellidos,correo,salario) VALUES(?,?,?,?)";

}

try(PreparedStatement myStamt = getConnection().prepareStatement(sql)){

myStamt.setString(1,employee.getNombre());

myStamt.setString(2,employee.getApellidos());

myStamt.setString(3,employee.getCorreo());

myStamt.setInt(4,employee.getSalario());

if (employee.getId() != null && employee.getId()>0) {

myStamt.setInt(5,employee.getId());

}

myStamt.executeUpdate();

}catch (SQLException e) {

e.printStackTrace();

}

}

Modificamos el método MAIN.

try (Connection myConn = DatabaseConnection.*getInstance*()){

Repository<Employee> reopository = new EmployeeRepository();

// System.out.println("------Listado----------");

// reopository.findAll().forEach(System.out::println);

System.*out*.println("------Actualizando un empleado----------");

Employee employee = new Employee();

employee.setId(1);

employee.setNombre("Diego");

employee.setApellidos("Soto");

employee.setCorreo("DiegoSoto@ejemplo.com");

employee.setSalario(20000);

reopository.save(employee);

System.*out*.println("------Empleado Actualizado----------");

reopository.findAll().forEach(System.*out*::println);

}catch (SQLException e) {

e.printStackTrace();

}

El método delete(Integer id) está implementando la funcionalidad de eliminar un registro de la base de datos. Utiliza una conexión a la base de datos obtenida mediante getConnection(), luego crea un objeto PreparedStatement con la consulta SQL “DELETE FROM employees WHERE id=?” y establece el valor del parámetro id utilizando myStamt.setInt(1, id). Finalmente, ejecuta la actualización mediante myStamt.executeUpdate() para eliminar el registro correspondiente al ID proporcionado.

public static void main(String[] args) throws SQLException {

try (Connection myConn = DatabaseConnection.*getInstance*()){

Repository<Employee> reopository = new EmployeeRepository();

//System.out.println("------Listado----------");

//reopository.findAll().forEach(System.out::println);

System.*out*.println("------ELIMINAR----------");

reopository.delete(6);

System.*out*.println("------Listado----------");

reopository.findAll().forEach(System.*out*::println);

Modificamos el metodo MAIN

@Override

public void delete(Integer id) throws SQLException {

try(PreparedStatement myStamt = getConnection().prepareStatement("DELETE FROM empleados WHERE id=?")){

myStamt.setInt(1,id);

myStamt.executeUpdate();

}

}

Explicar cómo implementar los cuatro métodos básicos de un CRUD (Crear, Leer, Actualizar y Eliminar) a un reclutador es una excelente oportunidad para demostrar tu conocimiento y habilidades como desarrollador de software. Aquí te proporciono una explicación de cómo implementar estos métodos en el contexto de una aplicación Java utilizando una base de datos MySQL:

**1. \*\*Crear (Create)\*\*:**

- Método: Este método se utiliza para agregar un nuevo registro a la base de datos.

- Implementación:

- Debes tener una clase que represente el objeto que deseas crear. Por ejemplo, si estás trabajando con empleados, puedes tener una clase `Employee` con atributos como nombre, apellidos, correo, salario, etc.

- Utiliza una conexión a la base de datos (por ejemplo, JDBC) para conectarte a la base de datos MySQL.

- Crea una consulta SQL de inserción que incluya los valores necesarios para el nuevo registro.

- Ejecuta la consulta SQL utilizando un `PreparedStatement` y asigna los valores a los marcadores de posición.

- Maneja las excepciones que puedan ocurrir, como SQLException.

- Cierra la conexión a la base de datos después de la inserción.

**2. \*\*Leer (Read)\*\*:**

- Método: Este método se utiliza para recuperar registros de la base de datos.

- Implementación:

- Utiliza una consulta SQL de selección (por ejemplo, `SELECT \* FROM empleados`) para obtener los registros que deseas leer.

- Ejecuta la consulta SQL y obtén los resultados.

- Puedes mapear los resultados a objetos de tu clase (por ejemplo, `Employee`) para facilitar su manejo.

- Itera a través de los resultados y realiza las operaciones necesarias, como mostrarlos en la interfaz de usuario o almacenarlos en una estructura de datos.

- Maneja las excepciones, como SQLException.

- Cierra la conexión a la base de datos cuando hayas terminado de leer los datos.

**3. \*\*Actualizar (Update)\*\*:**

- Método: Este método se utiliza para modificar registros existentes en la base de datos.

- Implementación:

- Al igual que en la creación, debes tener una clase que represente el objeto que deseas actualizar (por ejemplo, `Employee`).

- Utiliza una consulta SQL de actualización que especifique qué registro deseas actualizar y qué valores deseas cambiar.

- Ejecuta la consulta SQL utilizando un `PreparedStatement` y asigna los valores a los marcadores de posición.

- Maneja las excepciones, como SQLException.

- Cierra la conexión a la base de datos después de la actualización.

**4. \*\*Eliminar (Delete)\*\*:**

- Método: Este método se utiliza para eliminar registros de la base de datos.

- Implementación:

- Utiliza una consulta SQL de eliminación (por ejemplo, `DELETE FROM empleados WHERE id = ?`) que especifique qué registro deseas eliminar.

- Ejecuta la consulta SQL utilizando un `PreparedStatement` y asigna el valor necesario al marcador de posición (por ejemplo, el ID del empleado que deseas eliminar).

- Maneja las excepciones, como SQLException.

- Cierra la conexión a la base de datos después de la eliminación.

En tu explicación al reclutador, puedes destacar la importancia de manejar excepciones de manera adecuada, asegurarte de cerrar las conexiones a la base de datos para evitar problemas de recursos y validar los datos de entrada para garantizar la integridad de los registros en la base de datos. También es importante mencionar que estos métodos pueden variar según el marco de trabajo (framework) que estés utilizando, como Spring o Hibernate, que proporcionan abstracciones para simplificar estas operaciones.

# 

# **Visualizando bases de datos con Java Swing**

Java Swing es una biblioteca de interfaz de usuario (UI) para aplicaciones de escritorio en Java. Proporciona un conjunto de componentes gráficos y herramientas que permiten la creación de interfaces gráficas interactivas y visualmente atractivas.

Algunos de los métodos o funciones más utilizados en Java Swing son:

JFrame: Permite crear una ventana principal para la aplicación.

JPanel: Proporciona un contenedor flexible para organizar y agrupar otros componentes.

JButton: Crea un botón interactivo que puede realizar acciones cuando se hace clic.

JLabel: Muestra un texto o una imagen en la interfaz gráfica.

JTextField: Permite al usuario ingresar texto o números.

JTable: Muestra datos en forma de tabla.

JScrollPane: Agrega barras de desplazamiento a componentes que pueden contener más contenido de lo que se muestra inicialmente.

Puedes encontrar documentación detallada y ejemplos de Java Swing en el sitio web oficial de Oracle: https://docs.oracle.com/javase/tutorial/uiswing/

Reto: Personalizar tu vista de Java Swing:

Como parte del curso, se te desafiará a personalizar la apariencia de tu interfaz gráfica en Java Swing. Explora diferentes opciones de diseño, colores, fuentes y tamaños para adaptar la interfaz .

Recurso: codigo de Java Swing.

Lecturas recomendadas

Trail: Creating a GUI With Swing (The Java™ Tutorials) body { margin-left:10px; margin-right:10px; line-height: 1.5; FONT-FAMILY: Arial, Helvetica, sans-serif; font-size: 0.8em; } a:link{text-decoration:none; color:#09569d;} a:visited{text-decoration

<https://docs.oracle.com/javase/tutorial/uiswing/>

# Transacciones

# ¿Qué es una transacción?

Las transacciones representan un conjunto de operaciones o acciones que deben tratarse como una unidad atómica e indivisible. Se utilizan para garantizar la consistencia y la integridad de los datos, lo que implica que se cumplan todas las operaciones o ninguna en caso de fallos.

Commit: El commit (confirmación) es una operación que se utiliza para finalizar exitosamente una transacción.

Rollback: El rollback (reversión) es una operación que se utiliza para deshacer todos los cambios realizados en una transacción y devolver la base de datos a su estado anterior a la transacción.

Las transacciones en Java con JDBC siguen el modelo ACID, que garantiza las siguientes propiedades:

Atomicidad (Atomicity): Una transacción se considera una unidad atómica, lo que significa que todas sus operaciones se ejecutan en su totalidad o ninguna de ellas se ejecuta en absoluto. Si ocurre algún error o fallo durante la transacción, se revierten todas las operaciones realizadas hasta el punto en que se produjo el error.

Consistencia (Consistency): La transacción debe asegurar que la base de datos pase de un estado válido a otro estado válido. Esto implica que todas las reglas y restricciones de integridad de los datos se mantengan antes y después de la transacción.

Aislamiento (Isolation): Las transacciones deben ser aisladas entre sí, lo que significa que los cambios realizados por una transacción no deben ser visibles para otras transacciones hasta que la transacción se haya completado correctamente. Esto evita problemas de concurrencia y garantiza la coherencia de los datos.

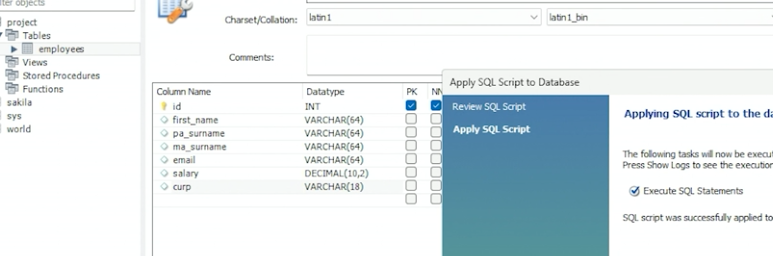
Durabilidad (Durability): Una vez que una transacción se completa correctamente, los cambios realizados en la base de datos deben ser permanentes y persistir incluso en caso de fallos del sistema o reinicios.

# Implementando transacciones con JDBC

Para agrupar varias sentencias en una única transacción, debes cambiar la propiedad “autocommit” de la conexión a “false”. Por defecto, esta propiedad está establecida en “true”. Al desactivar el “autocommit”, las sentencias no se ejecutarán de forma inmediata en la base de datos, lo que nos permite realizar varias operaciones en conjunto.

Si todas las sentencias dentro de la transacción se completan sin errores, se realiza un “commit” al final de la transacción. El “commit” confirma los cambios y los guarda de manera permanente en la base de datos.

Sin embargo, si ocurre alguna falla durante la ejecución de alguna sentencia dentro de la transacción, debes realizar un “rollback”. El “rollback” deshace todos los cambios realizados en la transacción, restaurando el estado anterior de la base de datos. Este paso se realiza dentro de un bloque catch, donde se capturan las excepciones.

crearemos un valor en la base de datos llamado curp 

quitaremos los getconection y lo pasaremos al mycon

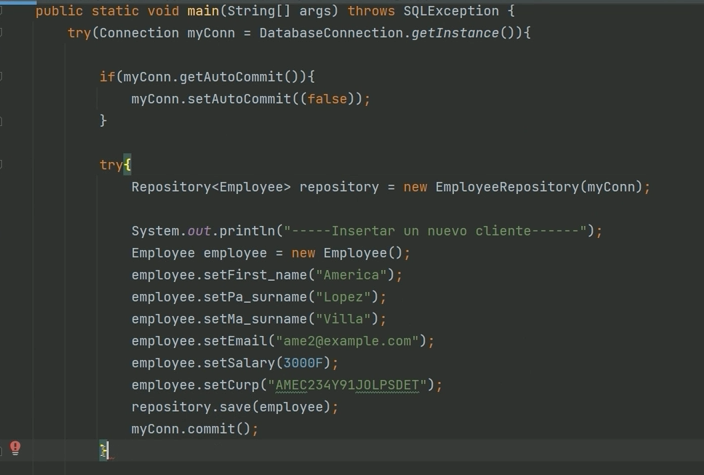


creando una sola conexion a una variable llamada mycon



agregaremos a los metodos de guardar y actualizar el campo nuevo.





creamos el commit si todo sale bien

y el rollback si pasa algo mal

# **Conexiones PooL**

# **¿Qué es una conexión pool?**

# ImplementandoUn pool de conexiones es una técnica para optimizar la gestión de conexiones a bases de datos. En lugar de establecer una nueva conexión cada vez, se solicita una conexión del pool existente, lo que ahorra tiempo y recursos.

# 

# Mejora del rendimiento: Al reutilizar conexiones existentes en lugar de establecer una nueva cada vez, se evita el costo de establecimiento de conexión, lo que resulta en una mejora del rendimiento de la aplicación.

# Ahorro de recursos: Al utilizar un pool de conexiones, se evita el consumo excesivo de recursos computacionales al reducir la carga de establecer y cerrar conexiones repetidamente.

# Tiempo de respuesta más rápido: Al tener conexiones preestablecidas y disponibles en el pool, las solicitudes pueden ser atendidas más rápidamente, lo que se traduce en una mejora del tiempo de respuesta de la aplicación.

# Control y gestión de conexiones: Un pool de conexiones permite tener un control más eficiente sobre el número máximo de conexiones permitidas, evitando así la sobrecarga de la base de datos o el servidor.

# Escalabilidad: Al utilizar un pool de conexiones, la aplicación puede manejar de manera eficiente un mayor número de solicitudes concurrentes sin saturar los recursos disponibles, lo que facilita la escalabilidad de la aplicación. conexiones pool.

Si se piden más conexiones de las que hay esperando en el pool se crearán en ese instante nuevas conexiones hasta el máximo de conexiones que permita el pool

Al devolver una conexión al pool , ésta se queda esperando para que otra petición la pueda usar. Si hay ya demasiadas conexiones esperando a ser usadas se cerrarán para ahorrar recursos en el servidor de base de datos.

La clase DataSource en Java se utiliza para configurar el pool de conexiones, estableciendo parámetros como el tamaño máximo y mínimo del pool.

Apache Commons Pool 2 es una biblioteca que proporciona una implementación genérica de un pool de objetos, incluyendo conexiones a bases de datos, y es comúnmente utilizada en aplicaciones Java.

Métodos de configuración:

setInitialSize(int): Establece el tamaño inicial del pool de conexiones.

setMaxTotal(int): Establece el número máximo de conexiones totales que el pool puede mantener activas simultáneamente.

setMaxIdle(int): Establece el número máximo de conexiones inactivas que el pool puede mantener en todo momento.

setMinIdle(int): Establece el número mínimo de conexiones inactivas que el pool debe mantener en todo momento.

setMaxWaitMillis(long): Establece el tiempo máximo (en milisegundos) que una solicitud de conexión puede esperar antes de lanzar una excepción por tiempo de espera.

La documentación oficial de Apache Commons DBCP2 se encuentra en el sitio web de Apache: <https://commons.apache.org/proper/commons-dbcp/>



# **JPA y ORM**

# **¿Qué es JPA y ORM?**

JPA (Java Persistence API) y JDBC (Java Database Connectivity) son dos tecnologías utilizadas en Java para interactuar con bases de datos. A continuación, te explicaré las diferencias principales entre ambas:

Nivel de abstracción:

JDBC: Es una API de bajo nivel que proporciona un conjunto de clases e interfaces para acceder a bases de datos directamente. Requiere escribir código manualmente para establecer conexiones, enviar consultas SQL y manejar el procesamiento de resultados.

JPA: Es una API de alto nivel que proporciona una capa de abstracción adicional sobre JDBC. Permite trabajar con objetos Java de manera transparente y mapearlos a tablas en la base de datos. JPA se basa en el patrón de diseño ORM (Object-Relational Mapping) para facilitar la persistencia y recuperación de objetos.

Mapeo objeto-relacional:

JDBC: No ofrece soporte nativo para el mapeo objeto-relacional. Los resultados de las consultas se obtienen en forma de conjuntos de filas y columnas de la base de datos, y el desarrollador debe procesar manualmente estos resultados y mapearlos a objetos Java.

JPA: Proporciona un mecanismo de mapeo objeto-relacional automático. Utiliza anotaciones o archivos XML para establecer relaciones entre las clases Java y las tablas de la base de datos. Los objetos se pueden guardar, actualizar, eliminar y consultar directamente sin la necesidad de escribir consultas SQL.

Productividad y mantenibilidad:

JDBC: Al ser una API de bajo nivel, el desarrollo con JDBC puede ser más tedioso y propenso a errores. Requiere escribir una cantidad significativa de código manualmente para realizar operaciones básicas en la base de datos. Además, cualquier cambio en el esquema de la base de datos puede requerir modificaciones en el código JDBC.

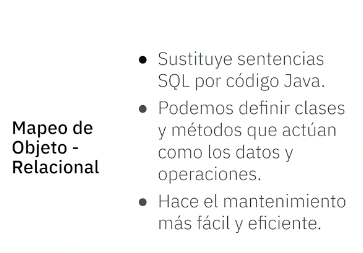
JPA: Al proporcionar una capa de abstracción más alta y un mapeo objeto-relacional automático, JPA simplifica el desarrollo y mejora la productividad. Permite escribir menos código y facilita el mantenimiento, ya que los cambios en el esquema de la base de datos se pueden reflejar en las clases Java a través del mapeo.

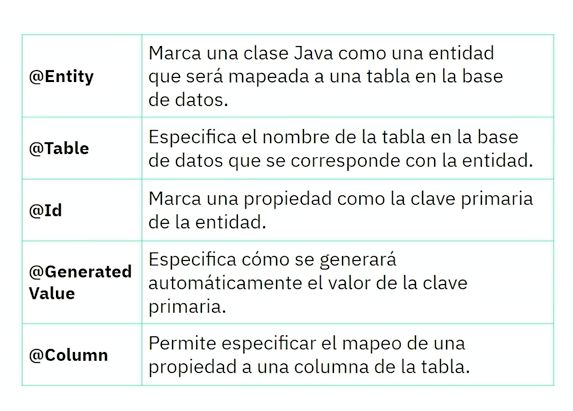
Portabilidad:

JDBC: Al ser una API estándar de Java, JDBC es ampliamente compatible y se puede utilizar con la mayoría de los sistemas de gestión de bases de datos (DBMS) disponibles.

JPA: JPA es una especificación de Java EE y, aunque se basa en JDBC, requiere un proveedor de persistencia específico (como Hibernate o EclipseLink) para interactuar con la base de datos. Aunque muchos proveedores son compatibles con JPA, puede haber diferencias en las características y el rendimiento entre ellos.

En resumen, JPA proporciona una capa de abstracción más alta, un mapeo objeto-relacional automático y una mayor productividad en comparación con JDBC. Sin embargo, si se requiere un mayor control y rendimiento a nivel de base de datos, JDBC puede ser más adecuado.







# **Instalación y configuración de Hibernate**

Hibernate es una herramienta de mapeo objeto-relacional para Java, que proporciona una capa de abstracción entre una base de datos relacional y las entidades de la aplicación. Facilita el desarrollo de aplicaciones persistentes al eliminar la necesidad de escribir consultas SQL directamente, permitiendo a los desarrolladores interactuar con la base de datos a través de objetos y consultas en lenguaje Java.

Además de las anotaciones estándar de JPA (Java Persistence API), Hibernate ofrece sus propias anotaciones para personalizar el mapeo y el comportamiento de las entidades. Estas anotaciones brindan un mayor control sobre cómo se realiza el mapeo de objetos a tablas de base de datos y permiten definir características específicas de Hibernate.

Algunas de las anotaciones proporcionadas por Hibernate incluyen:

@**Cascade**: Se utiliza para especificar el comportamiento de cascada en las operaciones CRUD, permitiendo que las operaciones realizadas en una entidad se propaguen a las entidades asociadas.

@**Fetch**: Permite controlar la estrategia de recuperación de datos, especificando cómo se deben cargar y recuperar los datos asociados a una entidad.

@**Formula**: Permite definir fórmulas SQL personalizadas en atributos calculados, donde el valor del atributo se calcula mediante una expresión SQL.

Estas anotaciones son solo algunos ejemplos de las muchas opciones que Hibernate proporciona para personalizar el mapeo y el comportamiento de las entidades.

Puedes encontrar más información y documentación detallada sobre Hibernate en el sitio web oficial: https://hibernate.org/

Además, aquí hay algunos enlaces útiles para aprender más sobre Hibernate:

Hibernate User Guide: https://docs.jboss.org/hibernate/orm/current/userguide/html\_single/Hibernate\_User\_Guide.html

Hibernate Annotations Reference Guide: https://docs.jboss.org/hibernate/stable/annotations/reference/en/html\_single/

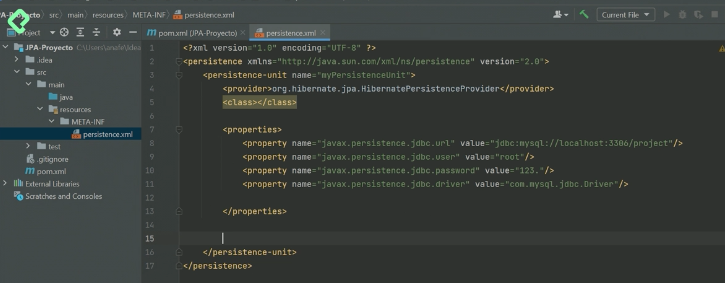
Explorar estos recursos te ayudará a comprender mejor Hibernate y aprovechar al máximo sus capacidades para el desarrollo de aplicaciones persistentes en Java.

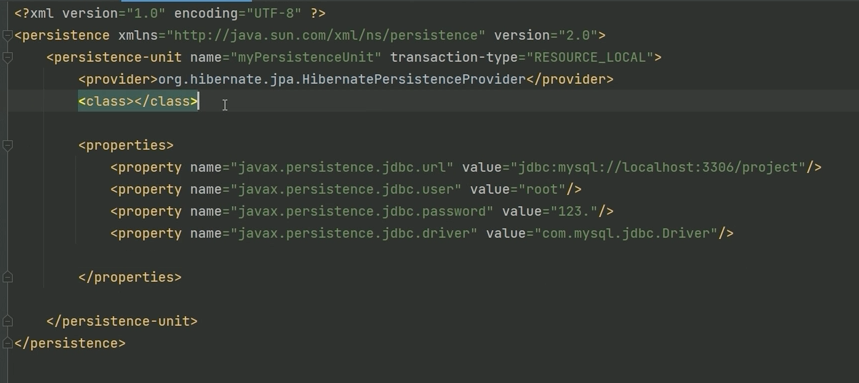
AGREGAREMOS LAS DEPENDENCIAS DE MYSQL CONNECTOR, HIBERNATE CORE Y JAVA PERSISTENCE.



ECLIPSELINK Y OPENJPA TAMBIEN PUEDEN SERVIR COMO DEPENDENCIAS

CREAREMOS VARIOS ARCHIVOS PARA LA CONFIGURACIÓN A CONTINUACIÓN SE DARÁ EL EJEMPLO DE CÓMO DEBE QUEDAR.





# **Implementando JPA con Hibernate**

El EntityManagerFactory es una clase importante en el contexto de Java Persistence API (JPA), que es una especificación para mapear objetos Java a una base de datos relacional. EntityManagerFactory es responsable de crear instancias de EntityManager, el cual se utiliza para interactuar con la base de datos en JPA. EntityManagerFactory configura la conexión a la base de datos y EntityManager representa una “unidad de trabajo”, realiza las operaciones de persistencia, como insertar, actualizar y eliminar datos.

Cuando trabajas con JPA, generalmente configuras un archivo de persistencia (como el persistence.xml) que contiene información sobre cómo establecer la conexión con la base de datos y cómo se deben gestionar las entidades. El archivo “persistent” es un archivo de metadatos y debe estar ubicado en el directorio “META-INF” en la estructura del archivo de la aplicación Java. “META-INF” es una convención utilizada en Java para almacenar metadatos y recursos específicos de la aplicación.

La Java Persistence API (JPA) proporciona una serie de métodos y funcionalidades para interactuar con bases de datos relacionales. Algunos de los métodos más comunes que ofrece JPA son los siguientes:

persist(Object entity): Permite guardar una nueva entidad en la base de datos.

merge(Object entity): Permite actualizar entidades existentes en la base de datos.

remove(Object entity): Permite eliminar una entidad de la base de datos.

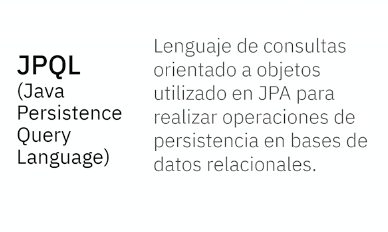
createQuery(String qlString): Permite realizar consultas JPQL (Java Persistence Query Language).

find(Class<T> entityClass, Object primaryKey): Permite obtener una entidad por su clave primaria.

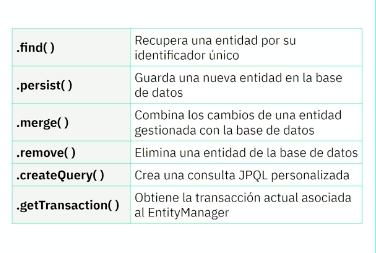
findAll(Class<T> entityClass): Permite obtener todas las entidades de una determinada clase.

begin() y commit(): Permiten administrar transacciones.

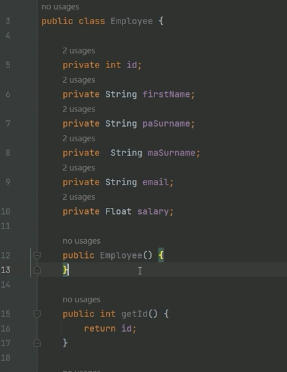
Estos son solo algunos ejemplos de los métodos que proporciona JPA. La API también incluye funciones para manejar consultas complejas, realizar operaciones transaccionales avanzadas, administrar el ciclo de vida de las entidades y realizar mapeos de relaciones entre objetos Java y tablas de la base de datos.







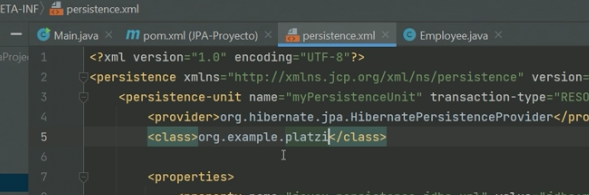
crearemos los getter, setter y los string de nuestros atributos de la clase



Después de tener los atributos y demás vamos a realizar el mapeo de nuestras tablas a nuestros atributos.



al archivo persistence conectaremos la entity class

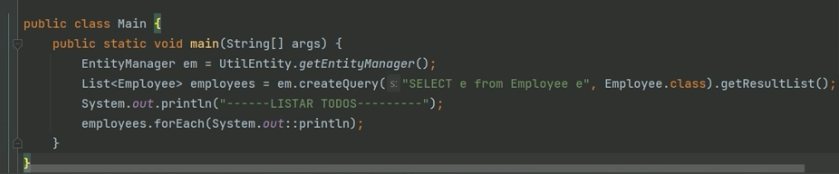


Realizaremos en entity Manager Factory, abre las puertas a nuestra base de datos

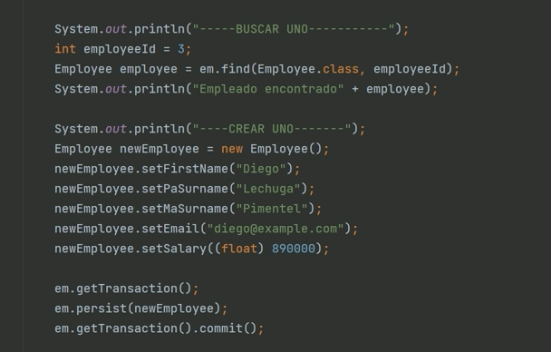
responsable de crear nuestras instancias entity manager o unidades de trabajado para crear nuestra unidad de persistencia de trabajo.



modificamos nuestra case main para que nos cargue nuestro usuarios

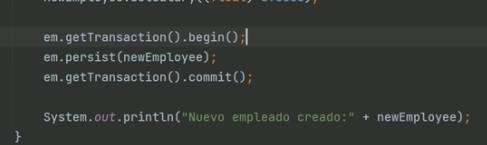


creamos la instancia para crear un empleado



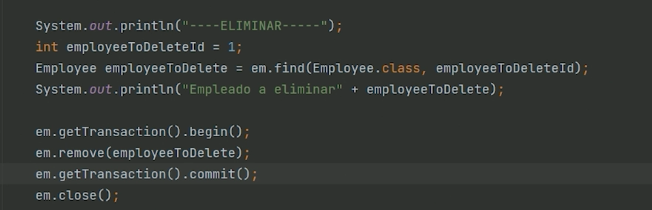


debemos inicializar la transacción

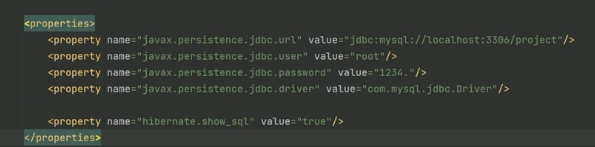




método eliminar



para poder ver las consultas que hace hibernate lo asignamos en el archivo de propiedades



# **Próximos pasos**

# **Reto: crea un clon de Netflix con Java**

