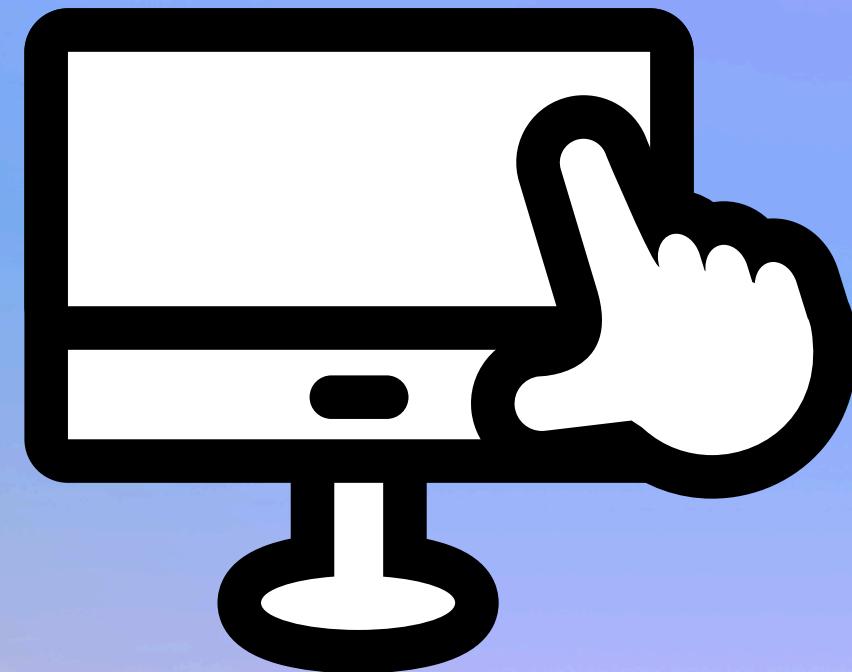


DESARROLLO DE UNA APLICACIÓN WEB PARA LA GESTIÓN DE EMPLEADOS CON CONEXIÓN A BASE DE DATOS

Desarrollo Web
Entorno Servidor

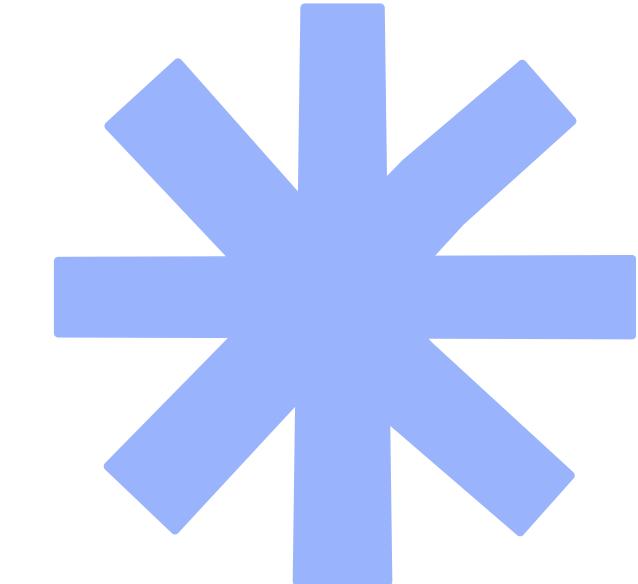
Lidia García Muñoz





Contenido

- 1 **Introducción**
- 2 **Arquitectura de la aplicación**
- 3 **Tecnologías y Frameworks utilizados**
- 4 **Conexión a la base de datos con JDBC**
- 5 **Model**
- 6 **Operaciones CRUD**
- 7 **Interfaz gráfica con Swing**
- 8 **Conclusión**



Introducción

El proyecto consiste en una aplicación de escritorio en Java que **permite administrar la información de los empleados de manera rápida y segura.**

A través de una interfaz intuitiva construida con Swing y una base de datos MySQL, esta aplicación facilita el registro, consulta, actualización y eliminación de empleados, optimizando la gestión dentro de la empresa. Además, cuenta con una búsqueda avanzada a través de la utilización de filtros.

📌 Funciones principales:

- ✓ Búsqueda avanzada y consulta de información.
- ✓ Registro de nuevos empleados.
- ✓ Actualización de datos.
- ✓ Eliminación de registros de forma segura.
- ✓ Interfaz gráfica amigable e intuitiva.

Arquitectura de la aplicación

El proyecto sigue una **arquitectura en capas** para una mejor organización, mantenimiento y escalabilidad.
Las capas principales son:

MODELO

- Representa las **entidades o los objetos que manejan los datos** (empleados).
- Define las estructuras de los datos, como atributos y relaciones.
- Ejemplo: Clases Java con anotaciones como `@Entity` para definir modelos de base de datos.

REPOSITORIO

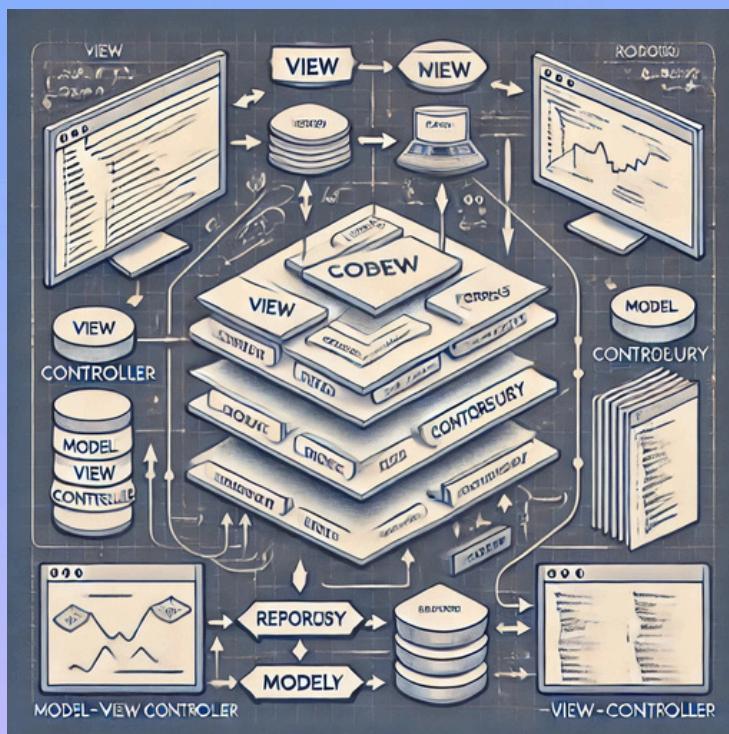
- Uso del **driver JDBC** puro para manejar la base de datos.
- Proporciona **operaciones CRUD** (crear, leer, actualizar, eliminar) de manera automática.
- Maneja las consultas y operaciones sobre los modelos.

VISTA

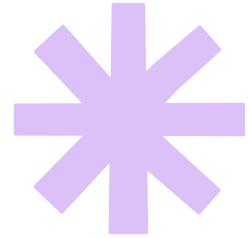
- Implementada con **Swing** para la interfaz gráfica de usuario.
- Contiene formularios, botones y otros componentes UI.
- Se encarga de la presentación de la información al usuario.

CONEXIÓN

- Gestiona la **conexión a la base de datos**.
- Maneja la configuración de credenciales y URL de la base de datos.
- Incluye un **pool de conexiones** para mejorar el rendimiento.



Tecnologías y frameworks utilizados



Backend:

Java y Spring Boot es una de las tecnologías más utilizadas para desarrollar aplicaciones web y servicios backend.

Es un **framework** basado en Spring que permite desarrollar aplicaciones en Java de forma rápida y sencilla, eliminando configuraciones complejas.



Base de Datos:

MySQL es un sistema de gestión de bases de datos relacional (RDBMS) de código abierto, ampliamente utilizado en aplicaciones web y empresariales.



JDBC (Java Database Connectivity) es una API de Java que permite la conexión y manipulación de bases de datos relacionales como MySQL.

Interfaz Gráfica:

Conjunto de **bibliotecas en Java** para crear interfaces gráficas de usuario (GUI) en aplicaciones de escritorio.

Es parte de **Javax** y proporciona componentes como ventanas, botones, tablas y más.



Gestor de Dependencias:

Herramienta de **gestión de proyectos en Java** que facilita la compilación, empaquetado y administración de dependencias. Se basa en un archivo de configuración (**pom.xml**) donde se definen las librerías y configuraciones del proyecto.



Conexión a la base de datos con JDBC

JDBC (Java Database Connectivity) es una API que **facilita la conexión entre aplicaciones Java y bases de datos**, permitiendo interactuar con ellas de forma sencilla y eficiente.

A continuación, se muestra un ejemplo de conexión a una base de datos MySQL utilizando Apache Commons DBCP para la gestión del pool de conexiones:

- Se utiliza *BasicDataSource* de Apache Commons DBCP para administrar el pool de conexiones.
- Se define la configuración de conexión, incluyendo la URL de MySQL, el usuario y la contraseña.
- Se establecen parámetros de optimización como el tamaño inicial del pool y el número de conexiones activas e inactivas.
- Se proporciona un método *getConnection()* para obtener conexiones de la base de datos de manera eficiente.

```
public class DatabaseConnection {  
  
    //-----CLASE PARA CONECTAR BASE DE DATOS  
    private static String url = "jdbc:mysql://localhost:3306/app_empleados";  
    private static String user = "root";  
    private static String password = "";  
  
    private static BasicDataSource pool;  
  
    public static BasicDataSource getInstance() throws SQLException {  
  
        if(pool == null){  
            pool = new BasicDataSource();  
            pool.setUrl(url);  
            pool.setUsername(user);  
            pool.setPassword(password);  
  
            //estas configuraciones dependen de lo que vayamos a necesitar  
            pool.setInitialSize(3); //tamaño inicial de nuestro pool de conexiones  
            pool.setMinIdle(3); //numero minimo de conexiones inactivas que se deben de mantener  
            pool.setMaxIdle(10); // establece el numero maximo de conexiones inact que se deben mantener  
            pool.setMaxTotal(10); //numero de conexiones total que el pool puede mantener activas simultaneamente  
        }  
        return pool;  
    }  
  
    public static Connection getConnection() throws SQLException {  
        return getInstance().getConnection();  
    }  
}
```

Model

¿Para qué necesitamos nuestra **clase modelo Employee**?

Es vital para **representar la estructura de los datos de un empleado** dentro de la aplicación. Se utiliza para almacenar y gestionar la información de los empleados en la base de datos.

En ella, además de los datos de los empleados, incluiremos también sus **constructores** y métodos **getter** y **setter** para obtener o modificar datos desde cualquier parte del código.

Veamos a continuación como implementarla en Java:



```
public class Employee {  
  
    //entidad que representa los datos para nuestros empleados  
    private Integer id;  
    private String nombre;  
    private String cargo;  
    private String departamento;  
    private Float rango_salarial;  
  
    //constructores  
    public Employee() {}  
  
    public Employee(Integer id, String nombre, String cargo, String departamento, Float rango_salarial) {  
        this.id = id;  
        this.nombre = nombre;  
        this.cargo = cargo;  
        this.departamento = departamento;  
        this.rango_salarial = rango_salarial;  
    }  
    // métodos getter y setter  
  
    public Integer getId() { return id; }  
    public void setId(Integer id) { this.id = id; }  
    public String getNombre() { return nombre; }  
    public void setNombre(String nombre) { this.nombre = nombre; }  
    public String getCargo() { return cargo; }  
    public void setCargo(String cargo) { this.cargo = cargo; }  
    public String getDepartamento() { return departamento; }  
    public void setDepartamento(String departamento) { this.departamento = departamento; }  
    public Float getRango_salarial() { return rango_salarial; }  
    public void setRango_salarial(Float rango_salarial) { this.rango_salarial = rango_salarial; }  
  
    //método toString  
  
    @Override  
    public String toString() {  
        return "Empleado{" +  
            "ID=" + id +  
            ", Nombre='" + nombre + '\'' +  
            ", Cargo='" + cargo + '\'' +  
            ", Departamento='" + departamento + '\'' +  
            ", Rango Salarial='" + rango_salarial +  
            '}';  
    }  
}
```

Operaciones CRUD

Las **operaciones CRUD (Create, Read, Update y Delete)** son la base fundamental de cualquier aplicación que maneje datos. Estas operaciones permiten gestionar la información en una base de datos, proporcionando las acciones esenciales para registrar, consultar, modificar y eliminar registros.

En este caso, trabajamos con la entidad **Employee** (Empleado), cuya información se almacena en una base de datos. Para interactuar con la base de datos de manera eficiente, utilizamos *EmployeeRepository.java*, una clase que implementa las operaciones CRUD utilizando **JDBC**.

En nuestra arquitectura, **EmployeeRepository.java** actúa como una **capa intermedia entre la aplicación y la base de datos**. Se encarga de ejecutar consultas SQL específicas, encapsulando la lógica de acceso a datos para que otras partes del sistema puedan interactuar con los empleados sin preocuparse por los detalles de la base de datos. En ella, implementaremos los métodos para obtener datos, crear, editar y eliminar empleados.

Por otro lado, **Repository** representa una abstracción para la gestión de entidades. En nuestra app, es una interfaz que define los métodos comunes para acceder a datos, permitiendo que *EmployeeRepository.java* los implemente específicamente para la entidad *Employee*.

A continuación, exploraremos cada operación CRUD con su respectiva implementación en *EmployeeRepository.java*, incluyendo ejemplos de código y representaciones visuales para ilustrar su funcionamiento.

Operaciones CRUD

```
public interface Repository <T> {  
  
    //métodos que desarrollaremos en EmployeeRepository  
  
    List<T> findAll() throws SQLException;  
    T getById(Integer id) throws SQLException;  
    void save (T t) throws SQLException;  
    void delete (Integer id);  
  
    List<T> findByFilters(String nombre, String cargo, String departamento, Float salarioMin, Float salarioMax)  
        throws SQLException;  
}
```

Como vemos, tenemos la interfaz *Repository*, que es implementada por la clase *EmployeeRepository*. En esta clase, la interfaz *Repository* pasa a ser de tipo *Employee*.

En las siguientes diapositivas veremos como implementar los métodos necesarios para buscar, crear, modificar y eliminar datos, además de un método para filtrar empleados.



```
public class EmployeeRepository implements Repository<Employee>
```

Operaciones CRUD - Create (crear) y Update (actualizar)

Método save():

Permite insertar un nuevo empleado en la base de datos. Si el empleado ya tiene un ID, se actualiza en lugar de insertarse.

- Si el ID existe (es decir, el empleado ya está en la base de datos y deseas actualizar su información), se construye una sentencia SQL de actualización (**UPDATE**). Aquí, el ? es un placeholder para los valores que se asignarán más adelante.
- Si el ID no existe (lo que indica que se trata de un nuevo empleado), se construye una sentencia SQL de inserción (**INSERT INTO**). También tiene placeholders (?) que se llenarán con los datos del empleado.

A continuación, establecemos una conexión con la base de datos y con **PreparedStatement** ejecutamos la consulta SQL de actualización o inserción, asignando los valores del empleado a los placeholders. Por último, ejecutamos la consulta con **executeUpdate()**, que modifica los datos en la base de datos.

```
@Override //metodo para insertar o actualizar empleado
public void save(Employee employee) throws SQLException {
    String sql;
    if (employee.getId() != null && employee.getId() > 0) {
        sql = "UPDATE empleados SET nombre=?, cargo=?, departamento=?, rango_salarial=? WHERE id=?";
    } else {
        sql = "INSERT INTO empleados (nombre, cargo, departamento, rango_salarial) VALUES (?, ?, ?, ?)";
    }

    try (Connection myCon = getConnection();
         PreparedStatement myStat = myCon.prepareStatement(sql)) {
        myStat.setString( parameterIndex: 1, employee.getNombre());
        myStat.setString( parameterIndex: 2, employee.getCargo());
        myStat.setString( parameterIndex: 3, employee.getDepartamento());
        myStat.setFloat( parameterIndex: 4, employee.getRango_salarial());

        if (employee.getId() != null && employee.getId() > 0) {
            myStat.setInt( parameterIndex: 5, employee.getId());
        }

        myStat.executeUpdate();
    }
}
```



CREAR EMPLEADO

The image shows a user interface for managing employees. It consists of two main windows: a main window titled "Gestión de Empleados" and a modal dialog titled "Agregar Empleado".

Main Window:

- Fields: Nombre, Cargo, Departamento, Salario Min.
- Search: Buscar
- Table: Shows employee data with columns ID, Nombre, Cargo, Departamento, and Rango Salarial.
- Buttons: Agregar (highlighted with a red box), Actualizar, Eliminar.

Modal Dialog ("Agregar Empleado"):

- Fields: Nombre (David Ochoa), Cargo (Subdirector), Departamento (Compras), Rango Salarial (4500).
- Icon: Java logo.
- Buttons: Cancel, OK.

Success Message:

- Message: "Empleado agregado correctamente".
- Icon: Java logo.
- Buttons: OK.

Data Tables:

- Employee Data Table (Main Window):

ID	Nombre	Cargo	Departamento	Rango Salarial
1	Juan Pérez	Contable	Compras	5000.0
2	Ana López	Subdirector	Compras	4500.0
3	Carlos Martínez	Subdirector	Contabilidad	3000.0
4	María Rodríguez	Subdirector	Contabilidad	3500.0
5	Pedro Sánchez	Subdirector	Producción	2800.0
6	Laura Gómez	Subdirector	Producción	2500.0
7	José Fernández	Subdirector	Producción	5200.0
8	Carmen Díaz	Subdirector	Producción	4600.0
9	Miguel Torres	Subdirector	Producción	3100.0
10	Rosa Jiménez	Subdirector	Producción	3600.0
11	Lidia García	Subdirector	Producción	2500.0
12	Sergio Sanz	Subdirector	Producción	2000.0

- Employee Data Table (Modal):

ID	Nombre	Cargo	Departamento	Rango Salarial
1	Juan Pérez	Contable	Compras	5000.0
2	Ana López	Subdirector	Compras	4500.0
3	Carlos Martínez	Subdirector	Contabilidad	3000.0
4	María Rodríguez	Subdirector	Contabilidad	3500.0
5	Pedro Sánchez	Subdirector	Producción	2800.0
6	Laura Gómez	Subdirector	Producción	2500.0
7	José Fernández	Subdirector	Producción	5200.0
8	Carmen Díaz	Subdirector	Producción	4600.0
9	Miguel Torres	Subdirector	Producción	3100.0
10	Rosa Jiménez	Subdirector	Producción	3600.0
11	Lidia García	Subdirector	Producción	2500.0
12	Sergio Sanz	Subdirector	Producción	2000.0
13	David Ochoa	Subdirector	Compras	4500.0

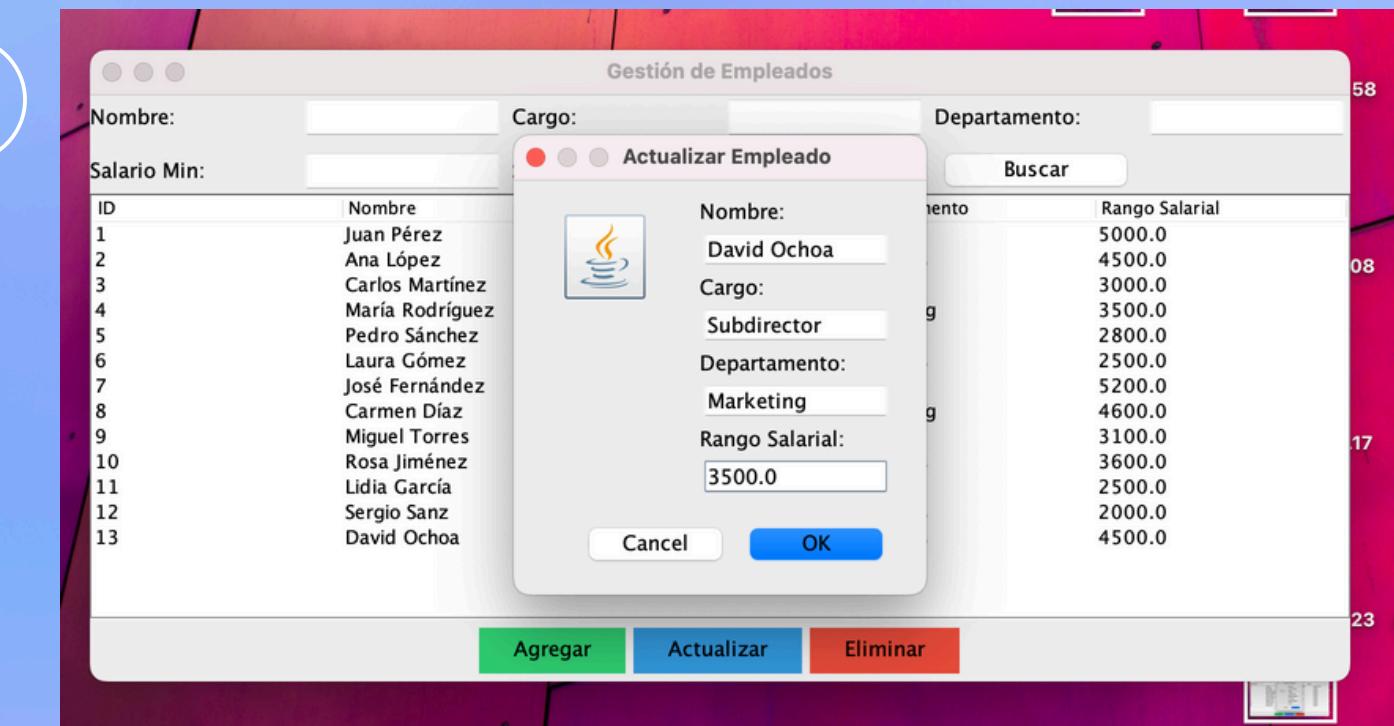


ACTUALIZAR EMPLEADO

1



2



3



Operaciones CRUD - Read (leer)

```
@Override //método para leer uno
public Employee getById(Integer id) throws SQLException {
    Employee employee = null;
    String sql = "SELECT * FROM empleados WHERE id = ?";

    try (Connection myCon = getConnection();
        PreparedStatement myStat = myCon.prepareStatement(sql)) {
        myStat.setInt(parameterIndex: 1, id);
        try (ResultSet myRes = myStat.executeQuery()) {
            if (myRes.next()) {
                employee = createEmployee(myRes);
            }
        }
    }
    return employee;
}
```

Método **getById()**:

El método se conecta a la base de datos, prepara y ejecuta la consulta "**SELECT * FROM empleados WHERE id = ?**" usando un **PreparedStatement** (para evitar inyección SQL) y asigna el ID al *placeholder*. Si la consulta devuelve un registro, crea y retorna un objeto *Employee* con los datos.

```
@Override //método para leer todos
public List<Employee> findAll() throws SQLException {
    List<Employee> employees = new ArrayList<>();
    String sql = "SELECT * FROM empleados";

    try (Connection myCon = getConnection();
        Statement myStat = myCon.createStatement();
        ResultSet myRes = myStat.executeQuery(sql)) {
        while (myRes.next()) {
            employees.add(createEmployee(myRes));
        }
    }
    return employees;
}
```

Método **findAll()**:

Se encarga de leer todos los empleados de la base de datos. El método se conecta a la base de datos y ejecuta la consulta "**SELECT * FROM empleados**" mediante un **Statement**. Luego, recorre el **ResultSet**, crea un objeto *Employee* para cada registro y retorna una lista con todos los empleados.

Operaciones CRUD - Delete (eliminar)

Método `delete()`:

Este método elimina un registro de la base de datos según el ID proporcionado.

- La consulta **SQL DELETE FROM empleados WHERE id=?** prepara una instrucción para eliminar un empleado basado en su ID. El signo de interrogación (?) actúa como un marcador de posición para el valor del ID.
- Usamos **PreparedStatement** para prevenir ataques de inyección SQL, asegurando que el valor del ID se inserte correctamente en la consulta.
- Por último, ejecutamos la consulta SQL con **executeUpdate()** y eliminamos así el registro de la base de datos.
- Si ocurre un error durante la operación, se captura la excepción gracias a **catch** y se lanza una nueva *RuntimeException*.

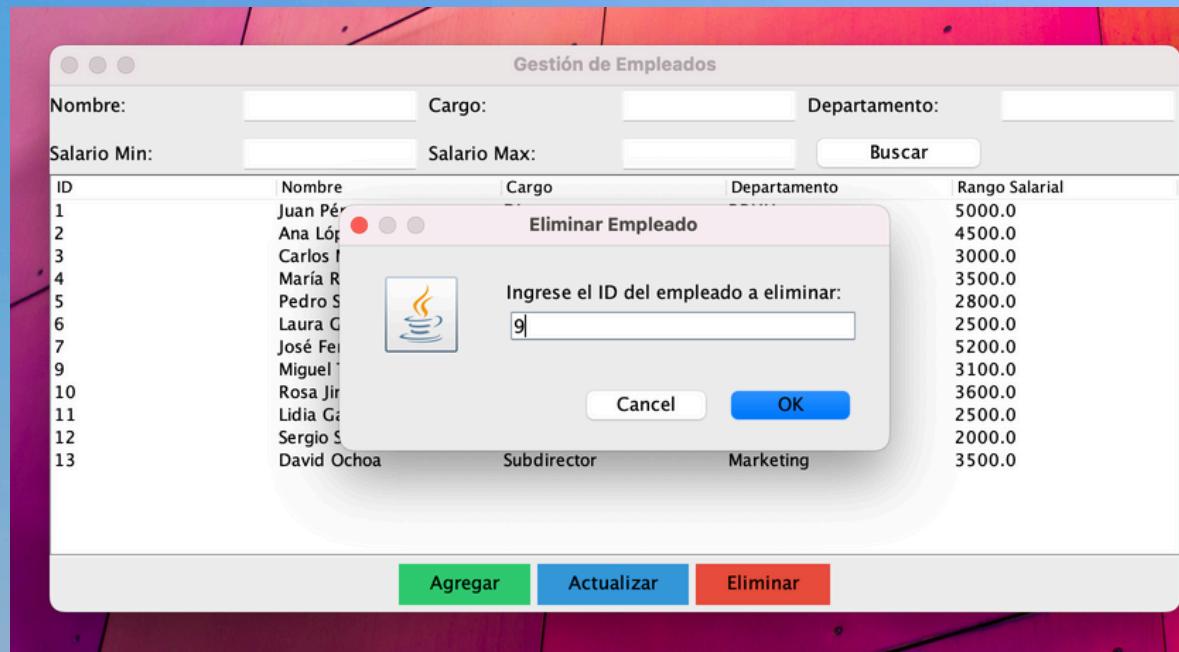
```
@Override //método para borrar empleado
public void delete(Integer id) {
    String sql = "DELETE FROM empleados WHERE id=?";

    try (Connection myCon = getConnection();
        PreparedStatement myStat = myCon.prepareStatement(sql)) {
        myStat.setInt( parameterIndex: 1, id);
        myStat.executeUpdate();
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
}
```

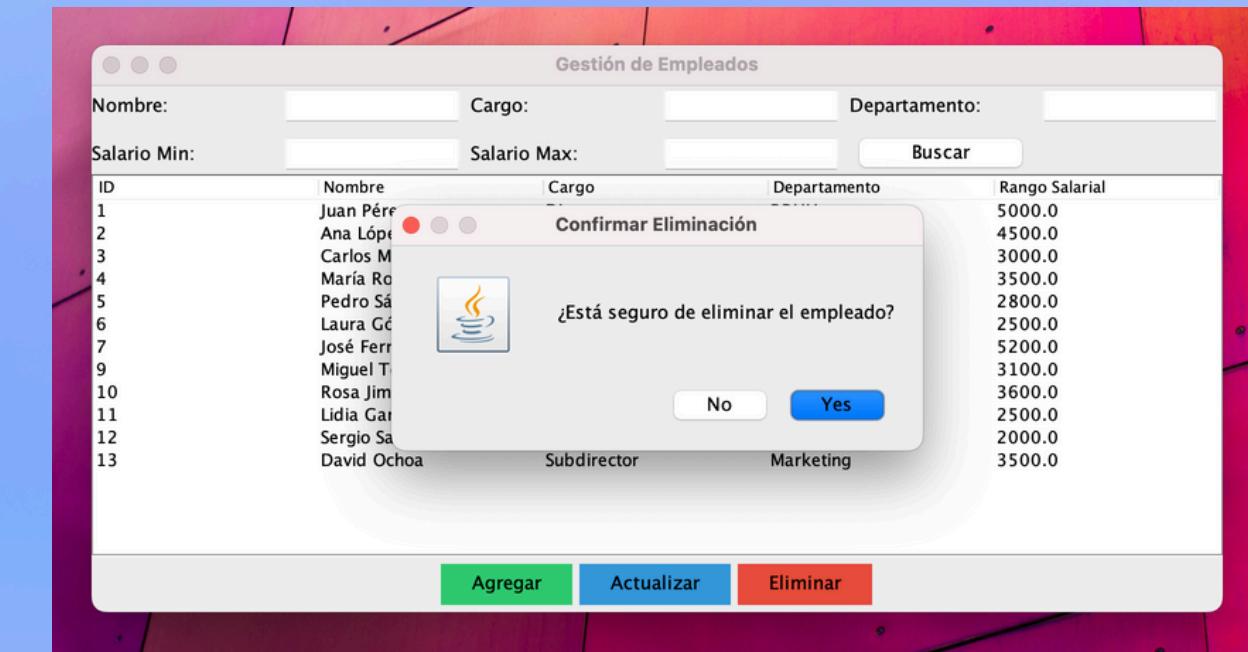


ELIMINAR EMPLEADO

1



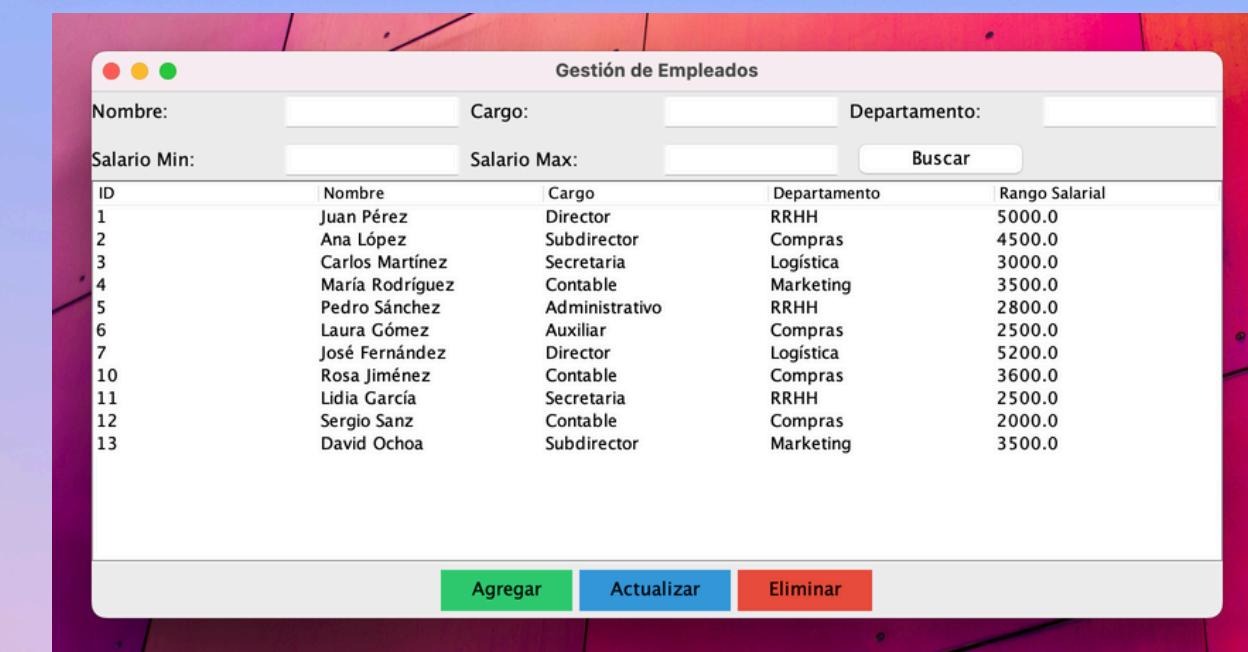
2



3



4



Operaciones CRUD - *findByFilters*

```
//Funcion para busqueda por filtros
private final Connection connection;

public EmployeeRepository() throws SQLException {
    this.connection = DatabaseConnection.getConnection();
}

public List<Employee> findByFilters(String nombre, String cargo, String departamento, Float salarioMin, Float salarioMax)
    throws SQLException {
    String sql = "SELECT * FROM empleados WHERE 1=1";
    List<Object> params = new ArrayList<>();

    if (nombre != null) {
        sql += " AND nombre LIKE ?";
        params.add("%" + nombre + "%");
    }
    if (cargo != null) {
        sql += " AND cargo LIKE ?";
        params.add("%" + cargo + "%");
    }
    if (departamento != null) {
        sql += " AND departamento LIKE ?";
        params.add("%" + departamento + "%");
    }
    if (salarioMin != null) {
        sql += " AND rango_salarial >= ?";
        params.add(salarioMin);
    }
    if (salarioMax != null) {
        sql += " AND rango_salarial <= ?";
        params.add(salarioMax);
    }
}
```

```
try (PreparedStatement statement = connection.prepareStatement(sql)) {
    for (int i = 0; i < params.size(); i++) {
        statement.setObject(parameterIndex: i + 1, params.get(i));
    }

    ResultSet resultSet = statement.executeQuery();
    List<Employee> employees = new ArrayList<>();
    while (resultSet.next()) {
        Employee employee = new Employee();
        employee.setId(resultSet.getInt(columnLabel: "id"));
        employee.setNombre(resultSet.getString(columnLabel: "nombre"));
        employee.setCargo(resultSet.getString(columnLabel: "cargo"));
        employee.setDepartamento(resultSet.getString(columnLabel: "departamento"));
        employee.setRango_salarial(resultSet.getFloat(columnLabel: "rango_salarial"));
        employees.add(employee);
    }
    return employees;
}
```

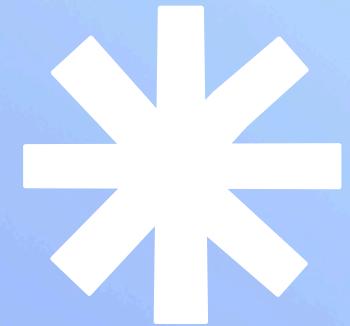


Operaciones CRUD - *findByFilters*

Método *findByFilters()*:

Este método permite buscar empleados basándose en filtros como nombre, cargo, departamento y rango salarial. Si un filtro no se proporciona, se omite de la consulta.

- La consulta comienza con "**SELECT * FROM empleados WHERE 1=1**", lo cual es una forma de garantizar que la cláusula WHERE esté siempre presente. A partir de allí, se van agregando condiciones si los parámetros de filtro no son nulos.
- Para cada parámetro (nombre, cargo, departamento, salario mínimo y máximo), si el valor no es nulo, se añade una condición a la consulta SQL. La **instrucción LIKE** se usa para hacer búsquedas parciales (por ejemplo, buscando empleados cuyo nombre contenga un valor parcial).
- Se prepara la consulta con un **PreparedStatement**, y los valores de los filtros se asignan a los parámetros mediante `statement.setObject()`.
- Después de ejecutar la consulta, se recorren los resultados (**ResultSet**) y se crean objetos *Employee* para cada fila obtenida, que luego se añaden a la lista *employees*.
- Al final, la lista de empleados filtrados se devuelve.



FILTRO NOMBRE

Gestión de Empleados

ID	Nombre	Cargo	Departamento	Rango Salarial
11	Lidia García	Secretaria	RRHH	2500.0

Nombre: Lidia Cargo: Departamento:

Salario Min: Salario Max: Buscar

Agregar Actualizar Eliminar

FILTRO CARGO

Gestión de Empleados

ID	Nombre	Cargo	Departamento	Rango Salarial
3	Carlos Martínez	Secretaria	Logística	3000.0
9	Miguel Torres	Secretaria	RRHH	3100.0
11	Lidia García	Secretaria	RRHH	2500.0

Nombre: Cargo: Secretaria Departamento:

Salario Min: Salario Max: Buscar

Agregar Actualizar Eliminar

FILTRO SALARIO

Gestión de Empleados

ID	Nombre	Cargo	Departamento	Rango Salarial
3	Carlos Martínez	Secretaria	Logística	3000.0
4	María Rodríguez	Contable	Marketing	3500.0
5	Pedro Sánchez	Administrativo	RRHH	2800.0
6	Laura Gómez	Auxiliar	Compras	2500.0
9	Miguel Torres	Secretaria	RRHH	3100.0
11	Lidia García	Secretaria	RRHH	2500.0
12	Sergio Sanz	Contable	Compras	2000.0

Nombre: Cargo: Departamento:

Salario Min: 2000 Salario Max: 3500 Buscar

Agregar Actualizar Eliminar

FILTRO COMBINADO

Gestión de Empleados

ID	Nombre	Cargo	Departamento	Rango Salarial
9	Miguel Torres	Secretaria	RRHH	3100.0
11	Lidia García	Secretaria	RRHH	2500.0

Nombre: Cargo: Secretaria Departamento: RRHH

Salario Min: Salario Max: Buscar

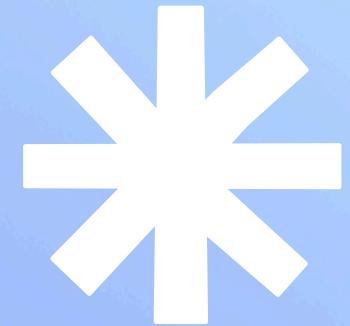
Agregar Actualizar Eliminar

Interfaz gráfica con Swing

Swing es una de las **bibliotecas gráficas más utilizadas en Java para crear interfaces de usuario (GUIs)**. Es parte de la Java Foundation Classes (JFC) y ofrece una amplia variedad de componentes visuales para construir aplicaciones de escritorio interactivas.

En esta aplicación de gestión de empleados, hemos utilizado **Swing** para construir una interfaz intuitiva y funcional. Algunos de los beneficios y características clave de Swing en esta aplicación incluyen:

- Usamos un **JTable** para mostrar los empleados y **JTextField** para ingresar datos de búsqueda y nuevos empleados.
- Gracias a los *listeners* (como **ActionListener**), podemos asociar acciones del usuario, como presionar un botón o escribir en un campo de texto, a funcionalidades específicas (buscar, agregar, actualizar o eliminar empleados).
- Organizamos la interfaz de manera flexible utilizando *layout managers* como **BorderLayout** o **GridLayout**. Esto facilita la creación de una interfaz ordenada y adaptable, donde los componentes se colocan automáticamente según el diseño deseado.
- Hemos personalizado los botones con colores y estilos específicos para mejorar la experiencia visual del usuario.
- Swing es completamente multiplataforma, lo que significa que nuestra aplicación puede ejecutarse en diferentes sistemas operativos (Windows, Mac, Linux) sin necesidad de modificaciones adicionales, lo que garantiza una amplia accesibilidad para los usuarios.



GESTIÓN DE EMPLEADOS APP

Gestión de Empleados

ID	Nombre	Cargo	Departamento	Rango Salarial
1	Juan Pérez	Director	RRHH	5000.0
2	Ana López	Subdirector	Compras	4500.0
3	Carlos Martínez	Secretaria	Logística	3000.0
4	María Rodríguez	Contable	Marketing	3500.0
5	Pedro Sánchez	Administrativo	RRHH	2800.0
6	Laura Gómez	Auxiliar	Compras	2500.0
7	José Fernández	Director	Logística	5200.0
10	Rosa Jiménez	Subdirector	Compras	7000.0
11	Lidia García	Secretaria	RRHH	2500.0
12	Sergio Sanz	Contable	Compras	2000.0
13	David Ochoa	Subdirector	Marketing	3500.0
14	Roberto	Subdirector	Compras	3000.0
15	David Pérez	Contable	Marketing	2400.0
17	Chini	Contable	RRHH	6000.0

Nombre: Cargo: Departamento:
Salario Min: Salario Max: Buscar

En conclusión, la implementación de una aplicación de gestión de empleados con **Spring Boot** y **MySQL** mediante **JDBC** proporciona una solución eficiente, escalable y segura para el manejo de datos. La conexión con JDBC permite una interacción directa y optimizada con la base de datos, asegurando un acceso rápido y confiable a la información de los empleados.

Además, Spring Boot facilita el **desarrollo ágil** con su integración simplificada de dependencias y herramientas, reduciendo la complejidad en la gestión de la base de datos.

Esta arquitectura garantiza un **sistema robusto, fácil de mantener y con un alto rendimiento**, ideal para entornos empresariales en constante crecimiento.



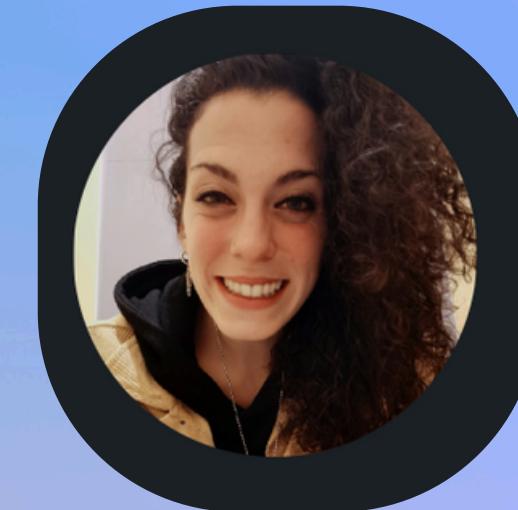
Conclusión



Fuentes externas

- ✓ Temario Cesur, módulo Desarrollo Web Entorno Servidor, ud6 - “*Utilización de técnicas de acceso a datos*”.
- ✓ Spring Boot Official Documentation - [*Focus on maximum service to retain existing customers*](#)
- ✓ JDBC (Java Database Connectivity) Documentation - [*Provide solutions to customers*](#)
- ✓ Tech Viajero, Youtube - [*“Cómo crear un CRUD #API con Spring Boot y MySQL en 40 minutos”*](#).

¡MUCHAS GRACIAS!



Lidia García Muñoz

