GENERALITAT VALENCIANA
Conselleria d'Educació, Universitats i Ocupació
Institut Educació Secundària El Caminàs
Unió Europea
Fons Social Europeu
El FSE inverteix en el teu futur

Acceso a bases de datos en Java

ÍNDICE

1. Introducción

En la mayoría de las aplicaciones modernas, el manejo eficiente y seguro de la información es crucial. A continuación, se presentan las razones fundamentales por las cuales las bases de datos son esenciales en los programas:

1. Persistencia de datos: Los programas necesitan almacenar datos de manera que no se pierdan cuando se apaga el sistema o se cierra la aplicación. Una base de datos permite que la información persista más allá de la duración de la ejecución del programa.

2. Eficiencia en el manejo de datos: Las bases de datos están diseñadas para gestionar grandes volúmenes de datos de manera eficiente. Ofrecen mecanismos optimizados para realizar operaciones de búsqueda, inserción, actualización y eliminación de datos.

3. Acceso concurrente: En aplicaciones multiusuario, es común que varias personas accedan y modifiquen los datos al mismo tiempo. Las bases de datos proporcionan mecanismos para manejar el acceso concurrente de manera segura y eficiente, asegurando la integridad de los datos.

4. Integridad de los datos: Las bases de datos permiten definir reglas de integridad (como restricciones de claves primarias) para asegurar que los datos almacenados sean válidos y consistentes.

5. Seguridad: Las bases de datos proporcionan mecanismos de seguridad para proteger la información, como autenticación, autorización y encriptación.

6. Escalabilidad: A medida que una aplicación crece, el volumen de datos también lo hace. Las bases de datos están diseñadas para escalar, permitiendo que las aplicaciones manejen cantidades crecientes de datos sin pérdida de rendimiento.

7. Facilidad de mantenimiento: Las bases de datos permiten administrar y actualizar los datos de manera centralizada. Esto facilita las tareas de mantenimiento (copias de seguridad, restauraciones y migraciones de datos).

8. Consultas Complejas: Las bases de datos permiten realizar consultas complejas para obtener información específica de manera rápida y eficiente usando lenguajes de consulta como SQL.

2. Bases de datos relacionales en Java

JDBC (Java Database Connectivity) es una API (Interfaz de Programación de Aplicaciones) proporcionada por Java que permite a las aplicaciones Java interactuar con bases de datos relacionales. Es un estándar que define cómo un cliente puede acceder a bases de datos de manera independiente del sistema gestor de bases de datos (SGBD) específico. Sus componentes principales son:

1. **Driver JDBC**: Es una implementación de la API JDBC específica para un SGBD concreto. Cada base de datos tiene su propio controlador JDBC, como MySQL, PostgreSQL, SQLite, etc. Los controladores traducen las llamadas API de JDBC en comandos específicos del DBMS.

2. **Conexión (Connection)**: Representa una conexión a una base de datos específica. A través de este objeto, se pueden enviar consultas y comandos SQL. Se obtiene utilizando `DriverManager.getConnection()`.

3. **Declaración (Statement)**: Es para ejecutar consultas SQL estáticas y recuperar los resultados. Hay tres tipos principales:
    - **Statement**: Para ejecutar sentencias SQL simples sin parámetros.
    - **PreparedStatement**: Para ejecutar sentencias SQL con parámetros, lo que mejora la seguridad (prevención de SQL Injection) y el rendimiento.
    - **CallableStatement**: Para llamar a procedimientos almacenados en la base de datos.

4. **Resultado (ResultSet)**: Contiene los datos recuperados de la base de datos después de ejecutar una consulta `SELECT`. Proporciona métodos para iterar sobre las filas de datos obtenidas.

3. SQLite

SQLite es una biblioteca de software que proporciona un sistema de gestión de bases de datos relacionales contenido en un archivo de formato ligero. Es una de las bases de datos más utilizadas en el mundo debido a su simplicidad, eficiencia y naturaleza auto-contenida. Sus características principales son:

1. **Auto-contenida**: Todo el sistema de gestión de bases de datos reside en una biblioteca de programación y no requiere un servidor separado para operar. Los datos se almacenan en un único archivo de base de datos en el sistema de archivos local.

2. **Cero Configuración**: No requiere instalación ni configuración compleja. Puedes empezar a usarla inmediatamente después de incluir la biblioteca SQLite en tu proyecto.

3. **Ligera**: Ocupa poco espacio en disco, y su código fuente es pequeño, lo que la hace ideal para aplicaciones donde el espacio es una preocupación, como en dispositivos móviles o sistemas embebidos.

4. **Transaccional**: SQLite soporta transacciones completas que cumplen con las propiedades ACID (Atomicidad, Consistencia, Aislamiento, Durabilidad), asegurando que las operaciones se completen correctamente o se reviertan en caso de error.

5. **Compatibilidad con SQL**: Admite la mayor parte del lenguaje SQL estándar, lo que permite realizar consultas complejas, aunque con algunas limitaciones en características avanzadas como procedimientos almacenados.

6. **Multiplataforma**: Funciona en casi todos los sistemas operativos, incluyendo Windows, macOS, Linux, Android, e iOS, lo que la convierte en una opción versátil para el desarrollo de software.

Ventajas de Usar SQLite:

- **Simplicidad**: Ideal para aplicaciones que no requieren un servidor de base de datos complejo.
- **Portabilidad**: La base de datos se almacena en un solo archivo, facilitando su traslado y copia de seguridad.

- **Rendimiento**: En aplicaciones pequeñas o de uso moderado, SQLite es extremadamente rápida debido a su naturaleza ligera.
- **Uso en Desarrollos Pequeños**: Es perfecta para prototipos, aplicaciones móviles, herramientas de escritorio, y sistemas embebidos.

Limitaciones de SQLite:

- **No Escalable para Grandes Aplicaciones**: No es adecuada para aplicaciones con alta concurrencia o grandes volúmenes de datos donde un RDBMS de servidor como MySQL o PostgreSQL sería más apropiado.
- **Soporte Limitado de Concurrencia**: Aunque maneja concurrencia de lectura bien, tiene limitaciones en escritura concurrente.

Uso Común de SQLite:

- **Aplicaciones Móviles**: Muchas aplicaciones en Android e iOS usan SQLite para almacenamiento local.
- **Desarrollo de Software Embebido**: Dispositivos como routers o televisores inteligentes la utilizan debido a su baja huella de memoria.
- **Prototipos y Pruebas**: Los desarrolladores la emplean para pruebas rápidas y desarrollo de prototipos.

Para poder acceder a una base de datos como SQLite desde nuestro código Java hay que realizar los siguientes pasos en nuestro proyecto:

1. **Registrar el controlador JDBC**: Ya hemos visto que cada SGBD tiene su controlador, por tanto, deberemos cargar el controlador JDBC correspondiente a SQLIte en nuestro proyecto.

2. **Establecer conexión con la base de datos**: Para ello deberemos utilizar el método `DriverManager.getConnection()` con la URL de la base de datos y, en caso necesario su usuario y contraseña.

4. Sentencias SELECT

Para extraer información de la base de datos, después de registrar el controlador JDBC y establecer conexión con la base de datos, hay que realizar lo siguiente:

1. **Crear el objeto de manejo de la base de datos:** Usando los objetos `Statement` o `PreparedStatement`.

2. **Ejecutar una consulta SQL**: A través del método `executeQuery` y guardar el resultado en un `ResultSet`.

3. **Procesar los Resultados**: Iterar sobre el `ResultSet` para obtener los datos.

4. **Cerrar Conexiones**: Es importante cerrar el `ResultSet`, `Statement`, y `Connection` para liberar recursos.

El siguiente ejemplo muestra cómo conectar con la base de datos llamada *sample.db* (situada en la carpeta del proyecto), consultar la información de su tabla *employees*, recorrer el resultado de la consulta y mostrar la información por consola.

- **Método `consultar`**: Recibe un objeto `Connection` como parámetro, realiza la consulta a la base de datos y procesa los resultados.
- **Llamada desde `main`**: después de establecer la conexión, se llama al método `consultar` pasando la conexión `conn` como argumento.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;


public class Main {
   public static void main(String[] args) {
        String url = "jdbc:sqlite:sample.db"; // Ruta de la base de datos
        Connection conn = null;
        try {
            conn = DriverManager.getConnection(url); // Establecer conexión
            System.out.println("Conexión establecida");

            consultar(conn); // consulta SELECT

        } catch (Exception e) {
```

```
            System.out.println(e.getMessage());
        } finally {
            try {
                if (conn != null) {
                    conn.close();
                    System.out.println("Conexión cerrada.");
                }
            } catch (Exception ex) {
                System.out.println(ex.getMessage());
            }
        }
    }

    public static void consultar(Connection conn) {
        Statement stmt = null;
        ResultSet rs = null;
        try {
            // Crear una declaración
            stmt = conn.createStatement();
            // Ejecutar consulta SQL
            rs = stmt.executeQuery("SELECT * FROM employees");

            // Procesar los resultados
            while (rs.next()) {
                System.out.println("Employee ID: " + rs.getInt("id"));
                System.out.println("Employee Name: " + rs.getString("name"));
            }

        } catch (Exception e) {
            System.out.println(e.getMessage());
        } finally {
            try {
                if (rs != null) rs.close();
                if (stmt != null) stmt.close();
            } catch (Exception ex) {
                System.out.println(ex.getMessage());
            }
        }
    }
}
```

La consulta anterior devuelve todos los registros de la tabla *employees* pero, imagina que queremos extraer solamente la información del empleado con un ID concreto que el usuario debe introducir por teclado, de forma que la consulta podría ser: `SELECT * FROM employees WHERE ID=1;` o `SELECT * FROM employees WHERE ID=2;` dependiendo del ID que el usuario haya indicado. Para poder realizar esta consulta lo primero que se nos ocurre es crear la sentencia SQL concatenando el ID que el usuario ha introducido (fíjate en las líneas sombreadas):

```
public static void consultar2(Connection conn) {
    Scanner scanner = new Scanner(System.in);
    System.out.print("Ingrese el ID del empleado a consultar: ");
    int id = scanner.nextInt();

    Statement stmt = null;
    ResultSet rs = null;
    try {
        // Crear una declaración
        stmt = conn.createStatement();

        String sql="SELECT * FROM employees WHERE ID="+id;
        rs = stmt.executeQuery(sql);

        // Procesar los resultados
        while (rs.next()) {
            System.out.println("Employee ID: " + rs.getInt("id"));
            System.out.println("Employee Name: " + rs.getString("name"));
        }

    } catch (Exception e) {
        System.out.println(e.getMessage());
    } finally {
        try {
            if (rs != null) rs.close();
            if (stmt != null) stmt.close();
        } catch (Exception ex) {
            System.out.println(ex.getMessage());
        }
    }
}
```

Pero este método es peligroso ya que no protege contra inyecciones SQL.

5. Inyecciones SQL

Una **inyección SQL** es un tipo de vulnerabilidad de seguridad que ocurre en aplicaciones que interactúan con bases de datos mediante comandos SQL. Esta vulnerabilidad permite que un atacante inserte o manipule consultas SQL maliciosas en las entradas de la aplicación para ejecutar comandos no previstos por el desarrollador. Esto puede conducir a la manipulación de los datos, la fuga de información confidencial, o incluso la destrucción de la base de datos.

La inyección SQL generalmente ocurre cuando las aplicaciones construyen consultas SQL directamente concatenando las entradas del usuario sin la debida validación o sanitización. Esto permite que un atacante inyecte fragmentos de código SQL malicioso en el comando.

Un ejemplo podría ser el de una aplicación que permite a los usuarios indicar su nombre de usuario y contraseña para iniciar sesión. Y cuyo código es:

```
String username = request.getParameter("username");
String password = request.getParameter("password");
String sql = "SELECT * FROM users WHERE username = '" + username + "' AND
password = '" + password + "'";
Statement stmt = connection.createStatement();
ResultSet rs = stmt.executeQuery(sql);
```

En este ejemplo, si una persona con malas intenciones indica en el campo de nombre de usuario el valor: `' OR '1'='1` y deja el campo de contraseña vacío, la consulta resultante será: `SELECT * FROM users WHERE username = '' OR '1'='1' AND password = ''` y la consulta siempre evaluará a verdadero, lo que permite al atacante iniciar sesión sin conocer la contraseña.

Consecuencias de la Inyección SQL

- **Acceso no autorizado**: Un atacante puede iniciar sesión como cualquier usuario, incluyendo administradores.
- **Pérdida de datos**: Un atacante puede borrar o modificar los datos de la base de datos.
- **Fuga de información**: Un atacante puede acceder a información confidencial almacenada en la base de datos.
- **Toma de control del servidor**: En algunos casos, un atacante puede ejecutar comandos en el servidor de la base de datos.

Prevenir la Inyección SQL

1. **Uso de Sentencias Preparadas (Prepared Statements)**: Utiliza sentencias preparadas en lugar de concatenar valores directamente en las consultas SQL. Esto asegura que las entradas del usuario se traten como datos, no como código ejecutable. Un ejemplo seguro es el siguiente:

```
String sql = "SELECT * FROM users WHERE username = ? AND password = ?";
PreparedStatement pstmt = connection.prepareStatement(sql);
pstmt.setString(1, username);
pstmt.setString(2, password);
ResultSet rs = pstmt.executeQuery();
```

2. **Validación y Sanitización de Entradas**: Valida y sanitiza todas las entradas del usuario antes de usarlas en consultas SQL. Esto puede incluir la eliminación de caracteres especiales o el uso de listas blancas.

3. **Uso de ORM (Object-Relational Mapping)**: Herramientas ORM como Hibernate o JPA manejan las consultas SQL de manera segura, lo que reduce significativamente el riesgo de inyección SQL.

4. **Control de Acceso**: Minimiza los privilegios de las cuentas de base de datos utilizadas por la aplicación para limitar el alcance de un ataque exitoso.

5. **Monitoreo y Auditoría**: Implementa mecanismos de monitoreo y auditoría para detectar actividades sospechosas en la base de datos.

Como ves la inyección SQL es una vulnerabilidad peligrosa, pero es completamente prevenible mediante buenas prácticas de codificación, como el uso de sentencias preparadas y la validación de entradas del usuario. Implementar estas medidas de seguridad es fundamental para proteger las aplicaciones y sus datos.

6. Sentencias con parámetros

Para evitar las inyecciones SQL existe otra manera de crear la sentencia SQL con la información que ha pasado el usuario y es utilizando parámetros. La sentencia se define así: `SELECT * FROM employees WHERE id = ?` y se utiliza un `PreparedStatement` que reemplaza el parámetro `?` con el ID proporcionado por el usuario. Este segundo método protege contra las inyecciones SQL.

```
public static void consultar3(Connection conn) {
    Scanner scanner = new Scanner(System.in);
    System.out.print("Ingrese el ID del empleado a consultar: ");
    int id = scanner.nextInt();

    PreparedStatement pstmt = null;
    ResultSet rs = null;
    try {
        // Preparar la consulta SQL para buscar el empleado por ID
        String sql = "SELECT * FROM employees WHERE id = ?";
        pstmt = conn.prepareStatement(sql);
        pstmt.setInt(1, id);

        // Ejecutar la consulta
        rs = pstmt.executeQuery();

        // Procesar los resultados
        if (rs.next()) {
            System.out.println("Employee ID: " + rs.getInt("id"));
            System.out.println("Employee Name: " + rs.getString("name"));
        } else {
            System.out.println("No se encontró un empleado con el ID
proporcionado.");
        }

    } catch (Exception e) {
        System.out.println("Error al consultar empleado: " + e.getMessage());
    } finally {
        try {
            if (rs != null) rs.close();
            if (pstmt != null) pstmt.close();
        } catch (Exception ex) {
            System.out.println(ex.getMessage());
        }
    }
}
```

7. Sentencias INSERT

Para insertar información se ejecutan las sentencias INSERT, a continuación tienes un ejemplo que también utiliza *prepareStatement* con parámetros:

```
public static void insertar(Connection conn) {
    Scanner scanner = new Scanner(System.in);
    System.out.print("Ingrese el ID del empleado: ");
    int id = scanner.nextInt();
    scanner.nextLine();  // Limpiar el buffer
    System.out.print("Ingrese el nombre del empleado: ");
    String nombre = scanner.nextLine();

    PreparedStatement pstmt = null;
    try {
        // Preparar la sentencia SQL para insertar un nuevo empleado
        String sql = "INSERT INTO employees (id, name) VALUES (?, ?)";
        pstmt = conn.prepareStatement(sql);
        pstmt.setInt(1, id);
        pstmt.setString(2, nombre);

        // Ejecutar el INSERT
        int rowsInserted = pstmt.executeUpdate();
        if (rowsInserted > 0) {
            System.out.println("Empleado insertado exitosamente.");
        }

    } catch (Exception e) {
        System.out.println("Error al insertar empleado: " + e.getMessage());
    } finally {
        try {
            if (pstmt != null) pstmt.close();
        } catch (Exception ex) {
            System.out.println(ex.getMessage());
        }
    }
}
```

8. Sentencias UPDATE

Siguiendo con el ejemplo anterior, vamos a ejecutar una sentencia UPDATE para actualizar el nombre de un empleado a partir de su ID. Pediremos ambos datos por teclado al usuario. Para ello utilizamos `PreparedStatement` para construir y ejecutar la sentencia SQL y de esta forma nos aseguramos que solo se actualiza el nombre del empleado correspondiente al ID proporcionado. El método queda así:

```
public static void actualizar(Connection conn) {
    Scanner scanner = new Scanner(System.in);
    System.out.print("Ingrese el ID del empleado que desea actualizar: ");
    int id = scanner.nextInt();
    scanner.nextLine(); // Limpiar el buffer
    System.out.print("Ingrese el nuevo nombre del empleado: ");
    String nuevoNombre = scanner.nextLine();

    PreparedStatement pstmt = null;
    try {
        // Preparar la sentencia SQL para actualizar el nombre del empleado
        String sql = "UPDATE employees SET name = ? WHERE id = ?";
        pstmt = conn.prepareStatement(sql);
        pstmt.setString(1, nuevoNombre);
        pstmt.setInt(2, id);

        // Ejecutar el UPDATE
        int rowsUpdated = pstmt.executeUpdate();
        if (rowsUpdated > 0) {
            System.out.println("Empleado actualizado exitosamente.");
        } else {
                    System.out.println("No se encontró un empleado con el ID
proporcionado.");
        }

    } catch (Exception e) {
        System.out.println("Error al actualizar empleado: " + e.getMessage());
    } finally {
        try {
            if (pstmt != null) pstmt.close();
        } catch (Exception ex) {
            System.out.println(ex.getMessage());
        }
    }
}
```

9. Sentencias DELETE

Por último, vamos a eliminar un empleado a partir de su ID. También utilizamos un `PreparedStatement` para prevenir inyecciones SQL y asegurarnos que la operación se realiza de forma segura. En el caso de las sentencias DELETE, se comprueba el número de filas eliminadas para informar si la operación se ha realizado correctamente o, si por el contrario, no se encontró ningún empleado con el ID proporcionado.

```
public static void eliminar(Connection conn) {
    Scanner scanner = new Scanner(System.in);
    System.out.print("Ingrese el ID del empleado que desea eliminar: ");
    int id = scanner.nextInt();

    PreparedStatement pstmt = null;
    try {
        // Preparar la sentencia SQL para eliminar el empleado por ID
        String sql = "DELETE FROM employees WHERE id = ?";
        pstmt = conn.prepareStatement(sql);
        pstmt.setInt(1, id);

        // Ejecutar el DELETE
        int rowsDeleted = pstmt.executeUpdate();
        if (rowsDeleted > 0) {
            System.out.println("Empleado eliminado exitosamente.");
        } else {
                    System.out.println("No se encontró un empleado con el ID
proporcionado.");
        }

    } catch (Exception e) {
        System.out.println("Error al eliminar empleado: " + e.getMessage());
    } finally {
        try {
            if (pstmt != null) pstmt.close();
        } catch (Exception ex) {
            System.out.println(ex.getMessage());
        }
    }
}
```