UTN FACULTAD REGIONAL CÓRDOBA
INGENIERÍA EN SISTEMAS DE INFORMACIÓN

Backend de Aplicaciones
Apunte de Clases

Apunte 12 - JDBC (Java Database Connectivity)

Antes de entrar al "Hola Mundo JDBC", conviene detenernos en **por qué usamos bases de datos**: estos sistemas proveen servicios de persistencia, integridad de datos, concurrencia, seguridad y consultas eficientes, como ya vimos en los contenidos de la asignatura Bases de Datos, estas capacidades descargan la responsabilidad en el sistema de administración de la base de datos y liberan a nuestro código de tener que lidiar con ellas. Nuestro trabajo entonces será conocer como interactuar con estos sistemas o librerías.

Existen múltiples alternativas modernas (bases NoSQL, documentales, clave-valor, grafos, etc.), pero en este curso vamos a enfocarnos en **bases de datos relacionales**, por una continuidad con la asignatura Bases de Datos, por su relevancia y por ser la base sobre la cual se apoyan la mayoría de los frameworks y herramientas de backend.

En este apunte nos centraremos en cómo Java se conecta e interactúa con bases de datos a través de JDBC que es la capa de más bajo nivel en esta relación. Aprenderemos las alternativas de base de datos relacionales que podemos conteplar en backend con microservicios y luego, los fundamentos de JDBC, cómo configurar conexiones a diferentes bases de datos y cómo realizar operaciones básicas.

Introducción: Alternativas de uso de base de datos

Antes de entrar al "Hola Mundo JDBC", dejamos claro **qué alternativas tenemos para montar la base de datos** en nuestros aplicativos.

En el mercado existen múltiples motores de bases de datos, tanto **relacionales (SQL)** como **no relacionales (NoSQL)**.
Entre los más reconocidos en el ámbito relacional encontramos:

- **Oracle Database** – Propietario, licenciamiento comercial. Es uno de los motores más completos y usados a nivel corporativo, con costos elevados de licencias y soporte.
- **IBM Db2** – Propietario, orientado a entornos empresariales con alto volumen de transacciones.
- **Microsoft SQL Server** – Propietario, con versiones comerciales y una edición gratuita limitada (Express). Fuerte integración con ecosistema Microsoft.
- **MySQL** – Originalmente open source, hoy bajo el paraguas de Oracle. Se ofrece con licencia dual: GPL (para proyectos libres) y comercial (para uso corporativo con soporte).
- **MariaDB** – Fork comunitario de MySQL, 100% open source (GPL). Se posiciona como alternativa libre y totalmente compatible.
- **PostgreSQL** – 100% open source bajo licencia **PostgreSQL License** (similar a MIT/BSD). Reconocido por su robustez, extensibilidad, cumplimiento de estándares SQL y gran comunidad.
- **SQLite** – 100% open source (dominio público). Motor ligero, embebido en dispositivos y aplicaciones.
- **H2** – Motor relacional open source (MPL 2.0 y EPL 1.0). Muy utilizado en entornos de desarrollo y pruebas por su simplicidad y modo embebido.

Nuestra elección en la cátedra

De entre todas estas alternativas, en la materia **elegimos dos motores**:

- **PostgreSQL** → por ser el motor open source de mayor envergadura, con un ecosistema muy activo, características avanzadas (roles, schemas, extensiones, transacciones complejas) y un modelo de licenciamiento permisivo que lo hace ideal tanto para proyectos académicos como profesionales.
- **H2** → como versión de entrada, ligera y embebida, perfecta para los primeros pasos de los alumnos. Permite trabajar sin instalación compleja, directamente en memoria o en archivos locales, y es compatible con el mismo ecosistema JDBC que luego usaremos con PostgreSQL.

De este modo, los estudiantes comienzan con un motor **simple y rápido (H2)** y progresan hacia un motor **robusto y productivo (PostgreSQL)**, replicando un camino natural en el desarrollo profesional.

Cabe aclarar aquí que si bien optamos por PostgreSql por ser una extensión natural de H2, MariaDB sería una opción igualmente válida para trabajar e implementar el trabajo práctico integrado de Backend o un Trabajo final por ejemplo.

H2

Opciones de uso de H2: embebido vs servidor independiente

Habiendo aclarado el panorama general, nos proponemos ahora documentar las **opciones de uso de H2**, ya que este motor puede utilizarse de dos formas principales:

1. **Modo embebido (Embedded)**
2. **Modo servidor independiente (Server)**

¿Qué significa que un motor sea *embebido*?

Cuando hablamos de un motor embebido nos referimos a que la base de datos **vive dentro del mismo proceso de la aplicación Java**. Podríamos decir entonces que en lugar de un Sistema de Administración de Bases de Datos (DBMS por su sigla en inglés) es una librería que brinda los mismos servicios dentro de nuestra propia JVM.

En el caso de H2:

- La base corre en la misma **JVM** que la aplicación.
- Puede residir **en memoria (RAM)** o en un **archivo local**.
- Comparte el ciclo de vida con la aplicación: cuando la JVM termina, la base también.
- No requiere procesos externos ni sockets de red.

Este enfoque hace que el acceso sea extremadamente rápido, ya que la aplicación y la base se comunican directamente en memoria.

Otra alternativa del mismo principio, y quizás el principal exponente, es SQLite que ya hemos usado en DDS.

Modo embebido (Embedded)

```
+-----------------------+
|  JVM de la aplicación  |
|  +-----------------+  |
|  |  Aplicación Java |  |
|  +-----------------+  |
|  +-----------------+  |
|  |  H2 Database     |  |  -> en memoria o en archivo
|  +-----------------+  |
+-----------------------+
```

- **Ventajas**: simplicidad, velocidad, cero configuración.
- **Limitaciones**:
  - Solo una aplicación puede conectarse a la base a la vez.
  - No es multiusuario real.
  - Pérdida de datos si se usa en memoria y no se persiste en archivo.

Memoria vs Archivo único: en el modo embebido se puede optar por crear la base directamente en memoria (`jdbc:h2:mem:nombre`) lo que implica máxima velocidad pero datos volátiles, o bien en un único archivo local (`jdbc:h2:file:./ruta/nombre`) lo que permite persistir los datos entre ejecuciones aunque se pierda la simplicidad del modo totalmente en memoria.

Modo servidor independiente (Server)

En este modo, H2 corre como un **proceso aparte** de la JVM de la aplicación, al que se accede vía **sockets TCP**.

```
+-----------------+        TCP/IP        +-----------------+
| Aplicación Java |  <-----------------> |  H2 Server DB   |
| (JVM Cliente)   |                      | (JVM Servidor)  |
+-----------------+                      +-----------------+
```

- **Ventajas**:
  - Permite múltiples conexiones concurrentes.
  - Simula más de cerca un motor de BD real (como PostgreSQL).
  - Mantiene datos disponibles aunque la aplicación se cierre.
- **Limitaciones**:
  - Requiere levantar el proceso del servidor H2.
  - Más complejidad en la configuración y administración.

Resumen comparativo

Característica	H2 Embebido	H2 Servidor
Proceso de ejecución	Misma JVM que la aplicación	JVM independiente
Acceso	Directo en memoria/archivo	Conexión vía TCP/IP

Característica	H2 Embebido	H2 Servidor
Concurrencia	Una sola aplicación	Varias aplicaciones/usuarios
Persistencia	Opcional (memoria o archivo)	Persistencia en archivo
Similitud con producción	Baja	Alta (simula motores reales)

En conclusión:

- Usaremos **H2 embebido** al inicio, por su simplicidad y velocidad.
- Luego exploraremos **H2 servidor**, que se acerca más al escenario de un motor real como PostgreSQL, permitiendo multiusuario y persistencia entre ejecuciones.

H2 Embedded

Resumen: Se levanta en memoria o en archivo local. Comparte el ciclo de vida de la JVM. Modelo de memoria volátil (mem) o persistente en archivo (file). Acceso directo sin sockets ni procesos externos.

Ventajas (+): simplicidad, rapidez de arranque, cero configuración, ideal para ejemplos rápidos y pruebas unitarias. **Desventajas (–):** no apta para entornos multiusuario ni persistencia prolongada, los datos pueden perderse al terminar el proceso.

En resumen:

- Bloque de memoria embebido en la JVM del proyecto contenedor
- URL típica: `jdbc:h2:mem:testdb` o `jdbc:h2:file:./data/testdb`
- Se apaga cuando termina la VM (modo memoria)

H2 Server (Compartido)

Resumen: Se levanta como proceso independiente, con datos en memoria o archivos gestionados por el servidor. Los clientes se conectan vía sockets TCP al puerto del servidor H2.

Ventajas (+): persistencia más prolongada, soporte multiusuario, simulación realista de un motor de BD. **Desventajas (–):** requiere iniciar y mantener el proceso del servidor, agrega complejidad frente al modo embebido.

En viñetas:

- La base de datos se levanta como un servidor de base de datos en el puerto de acuerdo con el esquema de ejecución que es configurable
  - TCP Server en puerto 9092
  - Web Server (con H2 Console) en puerto 8082
- Útil cuando queremos que el estado de la base sobreviva a varios runs de la app o compartir datos entre clientes
- URL típica: `jdbc:h2:tcp://localhost:port/~/test`

Utilización

En el caso de querer utilizar H2 en su modo servidor, hemos dejado un paso a paso en el ejmplo 3 que acompaña este apunte documentando tanto la forma de iniciar el servidor, como también la de inicializar la

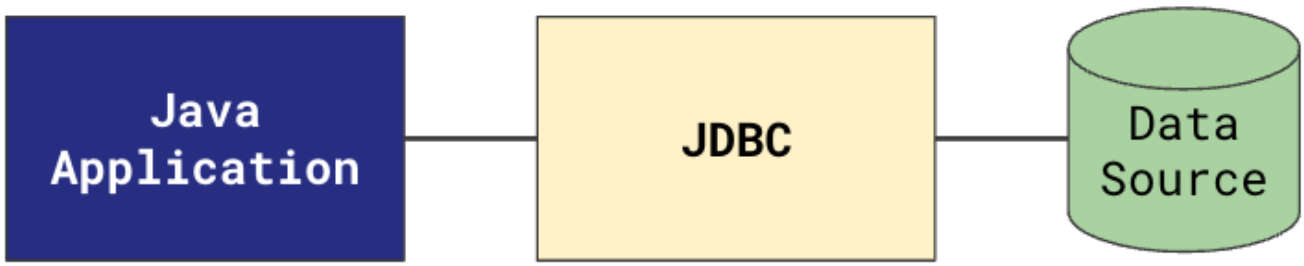
base de datos y conectarse a este desde la aplicación.

Esencialmente podemos, o bien crear un archivo `.sh` y utilizar el propio jar de la librería que descarga maven desde su repositorio local en `~/.m2` o bien crear un proyecto java y codificar el lanzamiento del servidor en el `main()` de dicho proyecto.

Una vez levantado el servidor solo restará conectarse al mismo implementando la url de conexión correspondiente.

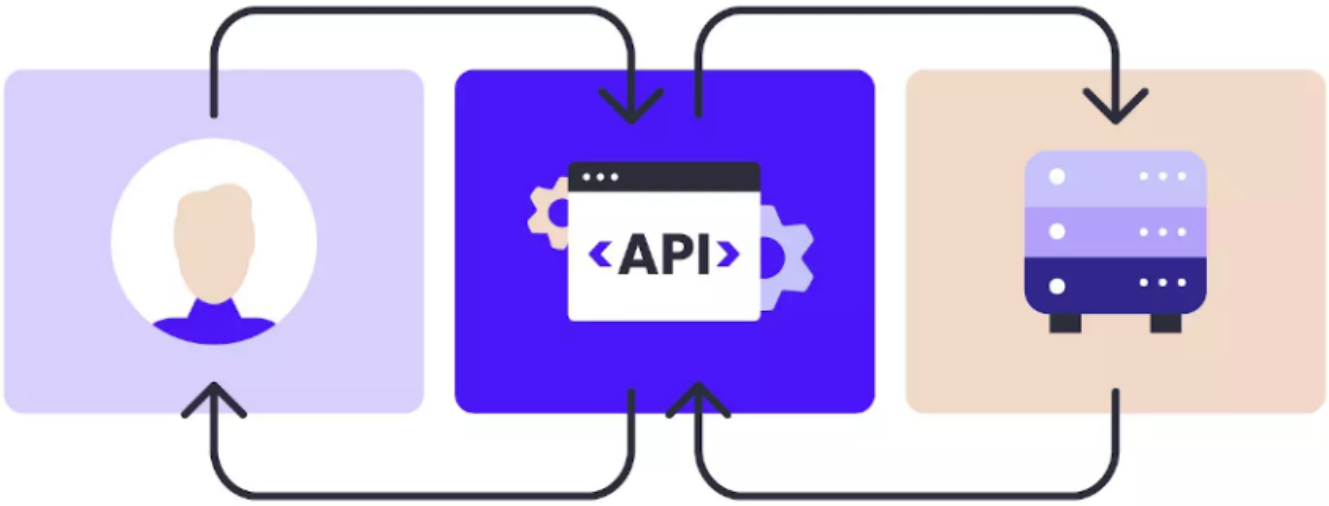
JDBC - API

JDBC utiliza drivers para conectarse a la DB.

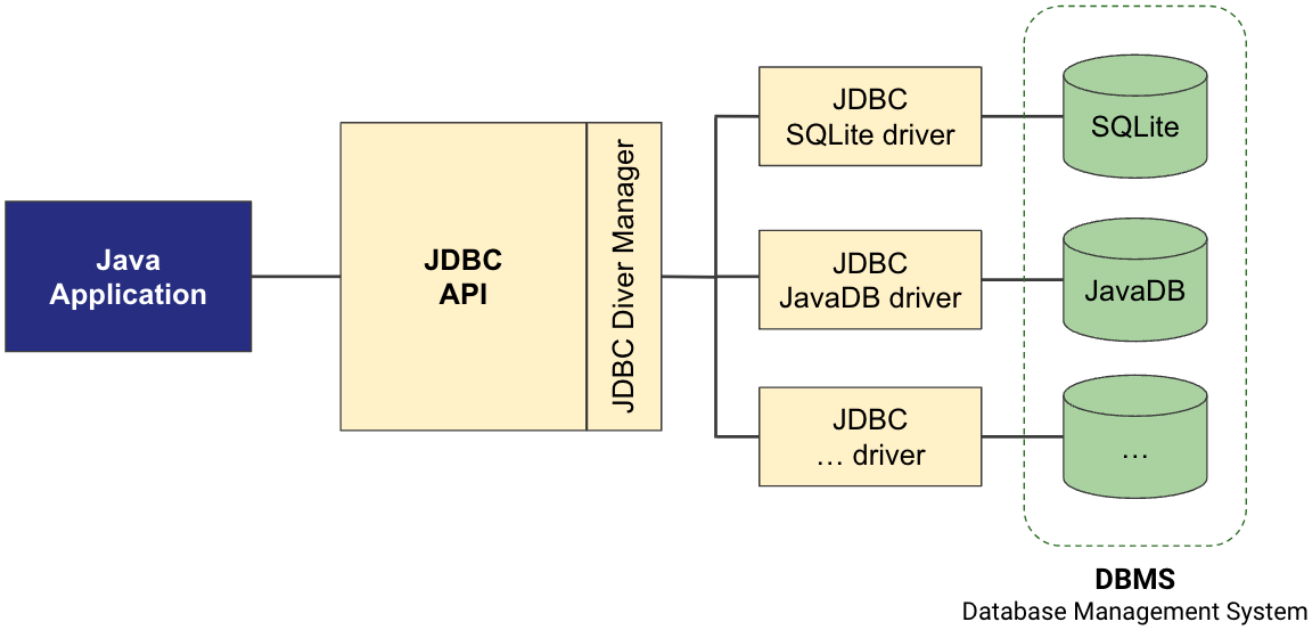


API (Application Programming Interfaces)

Es un conjunto de definiciones y protocolos que se utiliza para desarrollar e integrar el software de las aplicaciones, permitiendo la comunicación a través de un conjunto de reglas.



JDBC - Implementación



JDBC Driver es un componente de software que habilita a la aplicación java a interactuar con la DB.

Existen 4 Tipos de Drivers Jdbc

- **Tipo 1 - Bridge:** Controladores que implementan la API JDBC como una asignación a otra API de acceso a datos, como ODBC. El JDBC-ODBC Bridge es un ejemplo de un driver de Tipo 1.

  > Nota: El JDBC-ODBC Bridge debe considerarse solo si su DBMS no ofrece un controlador JDBC solo para Java y como una solución de transición.

- **Tipo 2 - Native:** Los controladores que se escriben en parte en el lenguaje de programación Java y en parte en código nativo. Estos controladores utilizan una biblioteca de cliente nativa específica para el origen de datos al que se conectan. El driver Oracle controlador del lado del cliente OCI (Oracle Call Interface) de Oracle es un ejemplo de un driver Tipo 2.
- **Tipo 3 - Network:** Controladores que utilizan un cliente Java puro y se comunican con un servidor middleware mediante un protocolo independiente de la base de datos. El servidor middleware entonces comunica las peticiones del cliente al origen de datos.
- **Tipo 4 – Thin o Pure Java:** Los controladores son Java puro e implementan el protocolo de red para una fuente de datos específica. El cliente se conecta directamente al origen de datos. El driver Oracle Thin es un ejemplo de driver Tipo 4.

Para interactuar con una Base de Datos debemos considerar 5 pasos:

1. Cargar el driver necesario para comprender el protocolo que usa la base de datos concreta (Hoy opcional en Drivers compatibles con JDBC 4 y SPI - Service Provider Interface)
2. Establecer una conexión con la base de datos, normalmente a través de red
3. Enviar consultas SQL y procesar el resultado
4. Liberar los recursos al terminar
5. Manejar los errores que se puedan producir

A continuación vamos a documentar cada uno de estos pasos en base al uso de H2 como base de datos embebida y asumiendo que la misma ya contiene el esquema de base de datos de ejemplo `Chinook` que se documenta en detalle en el primero de los ejemplos que acompaña el presente material.

Antes de continuar demos un vistazo al método `main(...)` del *Hola Mundo JDBC*:

```
    public static void main(String[] args) {

        // Paso 1 (Opcional hoy en día): Registrar driver
        try {
            // La clase del driver debe estar en el classpath de ejecución
            // Normalmete se puede encontrar en la documentación del motor
de base de datos
            Class.forName("org.h2.Driver");
        }
        // Si no se encuentra el driver, no tiene sentido continuar
        // Y en ese caso arroja ClassNotFoundException que es una
excepción verificada y por lo tanto
        // debe ser capturada o declarada.
        catch (ClassNotFoundException e) {
            e.printStackTrace();
            System.exit(1);
        }

        // Paso 2: Establecer conexión (o primero en caso de JDBC 4+ con
Drivers SPI)

        // La URL de conexión debe estar en la documentación del motor de
base de datos
        // En el caso de H2 embebido, la base de datos se crea al
conectarse si no existe
        // En este caso se crea en memoria (volátil) y va a existir
mientras dure la conexión
        // Por ultimo notar que el try-with-resources cierra
automáticamente la conexión al salir del bloque
        try (Connection conn =
DriverManager.getConnection("jdbc:h2:mem:chinook", "sa", "")) {

            // Inicializar base de datos (con tablas y datos de ejemplo
ver ejemplo 1)
            initDatabase(conn);

            // Paso 3 y 4: Enviar consulta y procesar resultados

            // Consulta de PlayLists en la base de datos
            String sql = "select PLAY_LIST_ID, NAME from PLAY_LIST";

            // Tanto el statement como el ResultSet son AutoCloseable y se
liberan automáticamente al salir del
            // bloque
            // Notar que solo se nos permitirá recorrer el ResultSet una
sola vez y solo se puede avanzar hacia
            // adelante
            // mientras el ResultSet esté abierto
            try (Statement st = conn.createStatement(); ResultSet rs =
st.executeQuery(sql)) {
```

```
                // Los statements diponen de varios métodos para ejecutar
consultas
                // executeQuery() para consultas que devuelven filas
(SELECT)
                // executeUpdate() para consultas que modifican filas
(INSERT, UPDATE, DELETE)
                // execute() para consultas genéricas (devuelven booleano
indicando si devolvieron filas o no)

                System.out.println("Listado de PlayLists:");

                // Recorrer el ResultSet
                // El cursor comienza antes de la primera fila
                // Por lo tanto se debe llamar a next() para avanzar a la
primera fila
                // next() devuelve false si no hay más filas
                while (rs.next()) {
                    // Se puede acceder por índice o por nombre
                    // En el caso de los índices, comienza en 1 a
contramano del mundo Java
                    // Personalmente me gusta pensar que esto es por que
las columnas son un conjunto y
                    // por lo tanto no existen el elemento 0
                    int id = rs.getInt(1);
                    // La otra alternativa es por nombre de columna
                    // Notar también que el método getXXX() es polimórfico
y se puede pedir el mismo dato
                    // en distintos tipos
                    // Por ejemplo si la columna es un INT se puede pedir
con getInt(), getString(), etc
                    String name = rs.getString("NAME");

                    System.out.println(id + " - " + name);
                }
            }

        }
        // Paso 5: Manejar errores
        // Todas las operaciones JDBC arrojan SQLException en caso de
error
        // Las operaciones de E/S arrojan IOException en caso de error
        // En este caso las capturamos juntas y IOException es una
excepción
        // que puede ser provocada por el método initDatabase()
        catch (SQLException | IOException e) {
            e.printStackTrace();
            System.exit(1);
        }
    }
```

Paso 1. Registrar driver (Hoy Opcional)

Antes de poder conectarse a la base de datos es necesario cargar el driver JDBC. Sólo hay que hacerlo una única vez al comienzo de la aplicación, esta acción hoy es opcional, en rigor de verdad desde JDBC 4+ y Java 6.

Para registrar el Driver pedimos al `Class Loader` de la `JVM` que intente cargar una clase a partir del nombre:

```
Class.forName("com.mysql.jdbc.Driver");
```

> esta clase contiene un bloque estático cuya función es, esencialmente, ejecutar `DriverManager.registerDriver(...)` con una instancia del Driver. Al cargarse la clase se registra el driver en la colección de drivers disponibles del DriverManager. En este caso sería:
>
> ```
> DriverManager.registerDriver(new org.h2.Driver());
> ```
>
> Lo cual también obliga a tener disponible el driver en momento de desarrollo y compilación. Sin embargo, en este caso evitamos la excepción `ClassNotFoundException` porque de no estar la clase no hubiera compilado.

El nombre de la clase del driver lo encontramos normalmente en la documentación de la base de datos.

Se puede obtener la excepción ClassNotFoundException si hay un error en el nombre del driver, pero fundamentalmente se obtiene si el archivo `.jar` no está correctamente en el CLASSPATH o en el proyecto, **lo que indica que no tenemos correctamente configurado el driver en el proyecto**

Desde **JDBC 4.0** (incluido en Java 6, 2006), se incorporó la carga automática de drivers mediante el mecanismo de Service Provider Interface (SPI).

Los drivers modernos (PostgreSQL, MySQL/MariaDB, H2, etc.) incluyen en su .jar un archivo en `META-INF/services/java.sql.Driver` que indica qué clase implementar.

El DriverManager usa ServiceLoader para descubrirlos y registrarlos automáticamente cuando están en el `classpath/modulepath`.

2. Crear el objeto de conexión

Las bases de datos actúan como servidores y las aplicaciones como clientes que se comunican a través de la red. Un objeto `Connection` representa una conexión física entre el cliente y el servidor. Para crear una conexión se usa la clase DriverManager donde se especifica la URL, el nombre y la contraseña:

```
Connection conn = DriverManager.getConnection("jdbc:h2:mem:chinook", "sa",
"pass");
```

El formato de la URL normalmente lo encontramos en la documentación del driver. En gejeral las url de conexión JDBC contemplan más o menos estos parámetros:

```
    # En servidores embebidos encontramos
     api db  conexión => en nuestro ejemplo en memoria con nombre chinook
    jdbc:h2:mem:chinook

     api db  conexión => si usamos h2 embebido con archivo
    jdbc:h2:file:./data/chinook.mv.db

    # Y si ya nos enfocamos en casos generales de conexiónes de red
encontramos
    # Para el caso de H2
     api db chnl ip-servidor   puerto    base de datos en archivo
    jdbc:h2:tcp://<servidor>[:<puerto>]/[<ruta>/]<nombreBaseDatos>

    # y ya para PostgreSQL
     api    dbms     ip-servidor  puerto   base de datos
    jdbc:postgresql://<servidor>:<puerto>/<base_de_datos>
```

> Además de lo revisado, en la url de conexión se pueden agregar configuraciones, por ejemplo en nuestra conexión a H2 en memoria podríamos agregar `DB_CLOSE_DELAY=#` que controla cuánto tiempo permanece abierta una base en memoria después de que la última conexión se cierra.
> Valores típicos:
> -1 → mantiene la base en memoria mientras viva la JVM, aunque se cierren todas las conexiones.
> 0 (valor por defecto) → la base en memoria se destruye inmediatamente al cerrarse la última conexión.
> 0 → tiempo en segundos que H2 mantiene la base antes de destruirla, esperando que se abra una nueva conexión.

Cada objeto `Connection` representa una conexión física con la base de datos.

Se pueden especificar más propiedades además del usuario y la password al crear una conexión. Estas propiedades se pueden especificar usando métodos `getConnection(…)` sobrecargados de la clase `DriverManager`.

Alternativas para crear Conexiones a la base de datos ahora con un ejemplo de MySql

```
String url = "jdbc:mysql://localhost:3306/sample";
String name = "root";
String password = "pass" ;
Connection c = DriverManager.getConnection(url, user, password);
```

```
String url =
   "jdbc:mysql://localhost:3306/sample?user=root&password=pass";
Connection c = DriverManager.getConnection(url);
```

```
String url = "jdbc:mysql://localhost:3306/sample";
Properties prop = new Properties();
prop.setProperty("user", "root");
prop.setProperty("password", "pass");
Connection c = DriverManager.getConnection(url, prop);
```

Finalmente tamién existe la opción de usar un DataSource como proveedor de la conexión o implementar un pool de conexiones mediante alguna librería como podría ser HikariCP pero eso lo veremos más adelante.

3. Crear la sentencia

Esta sentencia es la responsable de ejecutar las consultas a la DB. Una vez que tienes una conexión puedes ejecutar sentencias SQL:

- Primero se crea el objeto `Statement` desde la conexión
- Posteriormente se ejecuta la consulta y su resultado se devuelve como un `ResultSet`.

```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT ... FROM ... ");
```

Uso de Statement

Tiene diferentes métodos para ejecutar una sentencia

- `executeQuery(...)`: Se usa para sentencias SELECT. Devuelve un ResultSet.
- `executeUpdate(…)`: Se usa para sentencias INSERT, UPDATE, DELETE o sentencias DDL. Devuelve el número de filas afectadas por la sentencia.
- `execute(…)`: Método genérico de ejecución de consultas. Puede devolver uno o más ResulSet y uno o más contadores de filas afectadas.

Acceso al conjunto de resultados

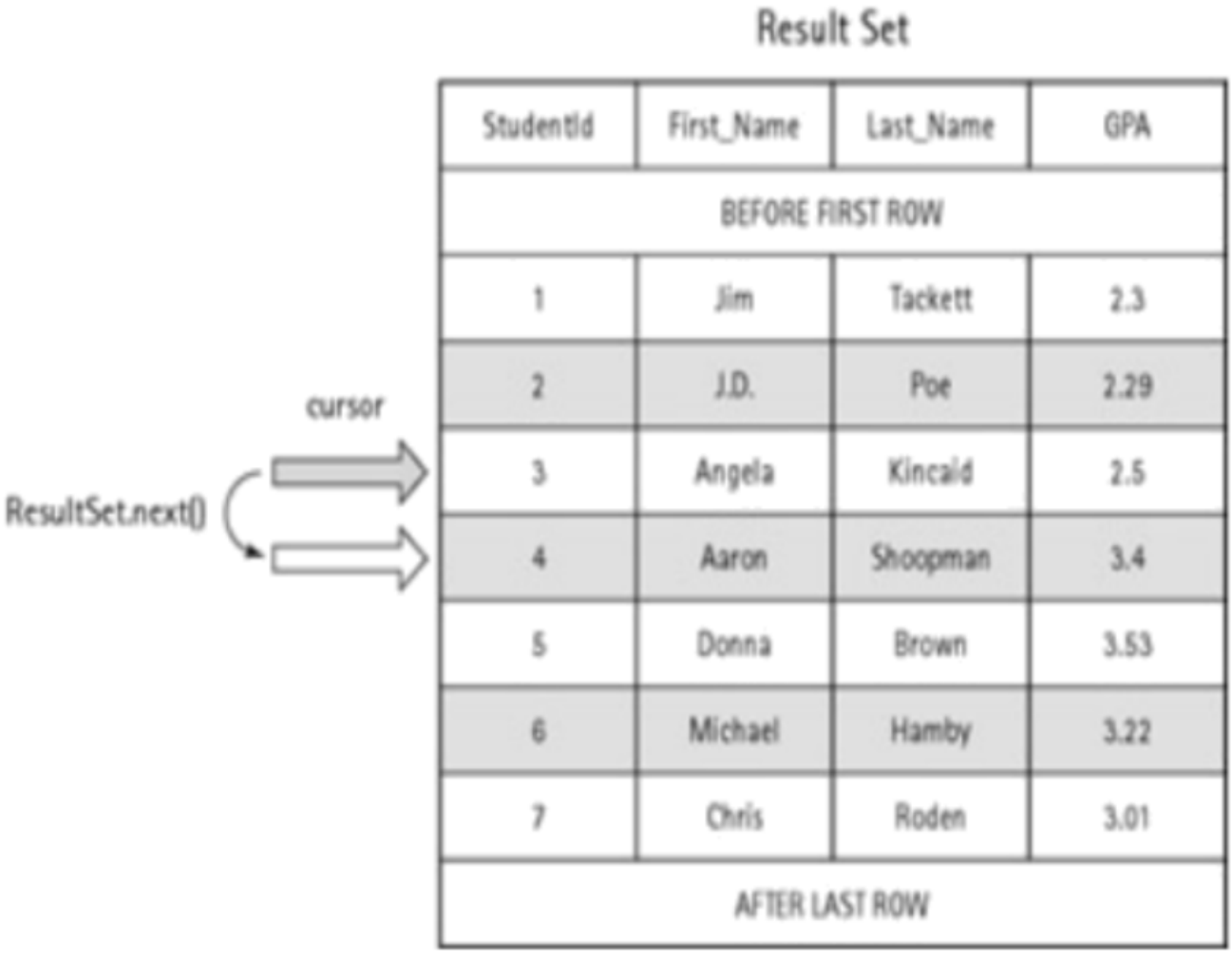
El `ResultSet` es el objeto que representa el resultado. No carga toda la información en memoria, internamente tiene un cursor que apunta a un fila concreta del resultado en la base de datos.

Hay que posicionar el cursor en cada fila y obtener la información de la misma.

Posicionamiento del cursor

- El cursor puede estar en una fila concreta.
- También puede estar en dos filas especiales.
    - Antes de la primera fila (Before the First Row, BFR)
    - Después de la última fila (After the Last Row, ALR)
- Inicialmente el `ResultSet` está en BFR.
- `next()` mueve el cursor hacia delante,

- devuelve `true` si se encuentra en una fila concreta
- y `false` si alcanza el ALR.



Obtención de los datos de la fila

Cuando el `ResultSet` se encuentra en una fila concreta se pueden usar los métodos de acceso a las columnas:

- String getString(String columnLabel)
- String getString(int columnIndex)
- int getInt(String columnLabel)
- int getInt(int columnIndex)
- … (existen dos métodos por cada tipo)

4. Liberar recursos

Cuando se termina de usar una `Connection`, un `Statement` o un `ResultSet` es necesario liberar los recursos que necesitan.

Puesto que la información de un ResultSet no se carga en memoria, existen conexiones de red abiertas.

Métodos close():

- `ResultSet.close()` – Libera los recursos del ResultSet. Se cierra automáticamente al cerrar el Statement que lo creó o al reejecutar el Statement.
- `Statement.close()` – Libera los recursos del Statement.
- `Connection.close()` – Finaliza la conexión con la base de datos

5. Manejar los errores

Hay que gestionar los errores apropiadamente:

- Se pueden producir excepciones `ClassNotFoundException` si no se encuentra el driver.
- Se pueden producir excepciones `SQLException` al interactuar con la base de datos
  - SQL mal formado
  - Conexión de red rota
  - Problemas de integridad al insertar datos (claves duplicadas)

Statements SQL en JDBC

Con JDBC disponemos de **tres interfaces principales** para ejecutar SQL. A continuación se muestra su **jerarquía**, **propósito**, y un resumen de **ventajas/desventajas** con ejemplos.

Jerarquía de tipos

```
java.sql.Statement
        └── java.sql.PreparedStatement
                    └── java.sql.CallableStatement
```

- `Statement` es la interfaz base.
- `PreparedStatement` "especializa" a `Statement` y añade **parámetros** (`?`) y preparación de la sentencia.
- `CallableStatement` añade soporte para **procedimientos almacenados** y **parámetros de salida**.

Cuándo usar cada uno (propósito)

- **Statement** → SQL **estático y puntual** (DDL o consultas simples) donde **no** hay parámetros ni se requiere reutilización del plan.
- **PreparedStatement** → Consultas/operaciones **parametrizadas y repetitivas**. Previene **SQL Injection** y mejora el rendimiento por **plan reutilizable**.
- **CallableStatement** → Cuando la lógica vive en la BD: **procedimientos/funciones** con **parámetros IN/OUT** y **códigos de retorno**.

Ventajas y desventajas

Tipo	Ventajas	Desventajas / Riesgos
Statement	Simplicidad; útil para DDL o pruebas rápidas.	**SQL Injection** si se concatenan valores; re-parsing del SQL en cada ejecución.

Tipo	Ventajas	Desventajas / Riesgos
PreparedStatement	**Parámetros** (?), evita inyección; posible **reutilización** del plan; **batch** eficiente.	Requiere enlazar parámetros; no hay parámetros con nombre (solo índices).
CallableStatement	**IN/OUT** params, **códigos retorno**, encapsula lógica en la BD.	**Portabilidad** limitada (dialectos/procedimientos difieren); acopla a la BD.

> Nota: La "preparación" puede ser **server-side o client-side** según el driver y motor; aun así, `PreparedStatement` mantiene sus beneficios de seguridad y API.

Ejemplos de uso

Statement (SQL simple, sin parámetros)

Como ya vimos en el caso de los statements simple directamente enviamos la consulta en el momento de la ejecución.

```
try (Statement st = conn.createStatement();
     ResultSet rs = st.executeQuery("SELECT GENRE_ID, NAME FROM GENRE
ORDER BY NAME")) {
    while (rs.next()) {
        System.out.printf("%d - %s%n", rs.getInt(1), rs.getString(2));
    }
}
```

PreparedStatement (parametrizado)

En el caso de los prepared statements, generamos un template con la consulta y podemos incrustar parámetros

```
String sql = "SELECT TRACK_ID, NAME FROM TRACK WHERE ALBUM_ID = ? AND
MILLISECONDS > ?";
try (PreparedStatement ps = conn.prepareStatement(sql)) {
    ps.setInt(1, 5);          // ALBUM_ID
    ps.setInt(2, 180_000);    // > 3 minutos
    try (ResultSet rs = ps.executeQuery()) {
        while (rs.next()) {
            System.out.printf("%d - %s%n", rs.getInt(1), rs.getString(2));
        }
    }
}
```

Batch con PreparedStatement (inserciones múltiples):

```
String ins = "INSERT INTO PLAY_LIST_TRACK (PLAY_LIST_ID, TRACK_ID) VALUES
(?, ?)";
try (PreparedStatement ps = conn.prepareStatement(ins)) {
    for (var trackId : trackIds) {
        ps.setInt(1, playlistId);
        ps.setInt(2, trackId);
        ps.addBatch();
    }
    int[] counts = ps.executeBatch();
}
```

Generated Keys:

```
String ins = "INSERT INTO ARTIST (NAME) VALUES (?)";
try (PreparedStatement ps = conn.prepareStatement(ins,
Statement.RETURN_GENERATED_KEYS)) {
    ps.setString(1, "New Artist");
    ps.executeUpdate();
    try (ResultSet keys = ps.getGeneratedKeys()) {
        if (keys.next()) {
            int id = keys.getInt(1);
        }
    }
}
```

CallableStatement (procedimientos/funciones)

> Disponibilidad y sintaxis **dependen del motor**. PostgreSQL típicamente expone **funciones**; H2 tiene soporte para **ALIAS**/funciones. El siguiente es un ejemplo genérico JDBC:

```
try (CallableStatement cs = conn.prepareCall("{ ? = call
GET_TOTAL_SALES_BY_CUSTOMER(?) }")) {
    cs.registerOutParameter(1, java.sql.Types.DECIMAL);
    cs.setInt(2, 42); // CUSTOMER_ID
    cs.execute();
    var total = cs.getBigDecimal(1);
}
```

Buenas prácticas cruzadas

- **Siempre** usar `PreparedStatement` para entradas de usuario (evita inyección, tipa parámetros).
- Cerrar recursos con **try-with-resources**.
- Usar `setAutoCommit(false)` y **transacciones** para operaciones múltiples coherentes; finalizar con `commit()`/`rollback()`.
- **Batch** para inserciones/updates masivos.
- Manejar **NULL** explícitamente (e.g. `ps.setNull(idx, Types.INTEGER)`).

- Evitar concatenar SQL dinámico; si lo necesitás (por ejemplo listas variables en `IN`), generá "placeholders" (`?, ?, ?`) acorde al tamaño.

Anti-patterns frecuentes

- Construir SQL con `String` + valores (`"... WHERE NAME = '" + userInput + "'"`) → **Inyección**.
- Reutilizar un `Statement` para lógica repetitiva y parametrizada → usá `PreparedStatement`.
- Abrir/cerrar conexiones por cada fila → considerá **pooling** y **batch**.

Resumen rápido (regla práctica)

- **¿Hay parámetros?** → `PreparedStatement`.
- **¿Es una llamada a procedimiento/función?** → `CallableStatement`.
- **¿Es un DDL o consulta ad-hoc temporal sin parámetros?** → `Statement`.

Anexo 1 - PostgreSQL (sobre Docker)

Resumen: Motor de BD de producción.
Mientras que un servidor H2 ofrece una experiencia ligera y limitada pensada para desarrollo y pruebas, PostgreSQL es un motor completo con administración de roles, schemas, transacciones avanzadas y extensiones, pensado para entornos productivos de alta concurrencia y con mayores requerimientos de recursos y configuración.
Datos en archivos gestionados por el motor, con buffers y cachés internos. Clientes se conectan vía TCP/IP al puerto 5432.

Ventajas (+): robustez, soporte multiusuario real, transacciones avanzadas, roles, schemas, extensiones, gran comunidad. **Desventajas (–):** mayor complejidad de instalación y administración, requiere más recursos y entorno adicional como Docker.

En viñetas:

- Persistencia en disco, multiusuario, roles y schemas
- URL típica: `jdbc:postgresql://localhost:5432/miBD`
- Ideal para mini-labs y para cuando pasemos a JPA

Instalación, Configuración y Uso

En Instructivo de Instalación de Postgres con Docker hemos dejado documentado de forma simple el proceso por el cual podemos montar un DBMS Postgres en nuestra estación de trabajo y dejarlo disponible para los ejemplos.

Y en el Ejemplo 3: JDBC con Postgresql hemos dejado un ejemplo más donde documentamos la conexión al servidor postgres y algunos ejemplos de statements sobre la base de datos Sakila.

Además hemos intentado un proceso funcional (en código plano y sin diseño, cabe la aclaración), para demostrar la protencción ante SQL Injection que brinda prepared statement y el uso de una transacción de Base de Datos para garantizar la integridad ante dos inserts del mismo proceso.

Anexo 2 - Pool de Conexiones JDBC

> **Propósito del anexo**: entender **qué es** y **por qué** usar un *pool de conexiones* en lugar de una sola conexión o de abrir/cerrar conexiones por operación. La meta es interpretar con solvencia los **parámetros** que luego veremos al configurar el pool (HikariCP) en Spring Boot / Spring Data.

1. El problema: costo y concurrencia de las conexiones

- **Abrir una conexión** a la BD es **caro**: handshake de red, autenticación, asignación de memoria en el servidor, configuración de sesión, etc.
- **Una única conexión** para toda la app ⇒ **cuello de botella** (un hilo por vez), riesgo de bloqueo si algo queda abierto.
- **Una conexión por operación** ⇒ alto overhead (tiempo y recursos), picos de latencia y presión excesiva sobre el servidor.

Necesidad: reutilizar conexiones abiertas y administrar cuántas están disponibles en paralelo de forma **eficiente y segura**.

2. ¿Qué es un pool de conexiones?

Un **pool** mantiene un conjunto de conexiones **pre-inicializadas** a la BD. La app **toma prestada** una conexión del pool cuando la necesita y **la devuelve** al terminar.

- La app pide conexiones a un `DataSource` (no al `DriverManager` directamente).
- Lo que "cerramos" en el código es en realidad un **proxy**: al cerrar, **no** se destruye la conexión física, **vuelve al pool** para su reutilización.
- El pool **crece y se achica** entre mínimos y máximos, valida conexiones, retira las sospechosas y **rota** las de mucha edad.

Beneficios: latencias más estables, mejor throughput, control explícito de **concurrencia** hacia la BD.

> En Spring Boot 3.x el pool por defecto es **HikariCP** (muy rápido y estable). Otros: Apache DBCP2, c3p0.

3. Ciclo de vida típico con pool

1. **Arranque**: el pool crea `minIdle` conexiones "calientes".
2. **En carga**: si faltan, crea nuevas hasta `maximumPoolSize`.
3. **Préstamo**: el código recibe un **wrapper** de `Connection`.
4. **Uso**: ejecutar SQL/tx; al **cerrar** (try-with-resources) la conexión vuelve al pool.
5. **Mantenimiento**: validación/keep-alive, expiración por `maxLifetime`, cierre de inactivas por `idleTimeout`.

4. Parámetros clave (conceptos, independientemente de la librería)

- `maximumPoolSize`: máximo de conexiones simultáneas. Define la **concurrencia** real hacia la BD.
  - Puntos de partida razonables: entre `#CPU × 2` y `#CPU × 4` **del backend**, ajustando por la latencia promedio de consultas y límites del servidor BD.
  - Para cargas muy I/O-bound (consultas lentas), puede requerir más; medir y vigilar saturación de la BD.
- `minimumIdle`: conexiones mínimas inactivas que el pool mantiene "calientes".
  - Útil para evitar "spikes" de latencia cuando llegan picos.

- `connectionTimeout`: cuánto espera un hilo para **obtener** una conexión libre antes de fallar.
  - Si se dispara, el pool está **exhausto** (o el máximo es bajo, o hay fugas, o las consultas demoran demasiado).
- `idleTimeout`: tiempo de **inactividad** tras el cual una conexión excedente se cierra.
  - No aplica si `minimumIdle == maximumPoolSize`.
- `maxLifetime`: edad máxima de una conexión antes de **rotarla** proactivamente.
  - Ponerlo **ligeramente menor** al timeout de conexiones del servidor o del balanceador para evitar cortes "a destiempo".
- `validationTimeout` / `keepaliveTime`: chequeos de salud para no entregar conexiones rotas.
- `leakDetectionThreshold`: (diagnóstico) loguea si una conexión quedó demasiado tiempo sin devolverse. Úsalo en **dev/test**.

> **Regla mental con Little's Law**: *concurrencia ≈ throughput × tiempo_medio_en_BD*. Si cada request pasa ~50 ms en BD y esperás 200 req/s **con** consulta a BD, necesitarías ≈ `0.05 × 200 = 10` conexiones concurrentes **solo** para esa carga.

5. Buenas prácticas con pool

- **Siempre** `try-with-resources` para `ResultSet` → `Statement` → `Connection`. Cerrar en ese orden.
- **Nunca** compartas `Connection`/`Statement` entre hilos ni lo guardes en `static`/singleton.
- Mantener **transacciones cortas** (solo lo necesario); confirmar (`commit`) o revertir (`rollback`) rápido.
- Evitar operaciones de **larga duración** (cálculos, I/O externo) con la conexión prestada.
- Cuidar el **estado de sesión**: cambios de `SET ...` deben resetearse o confiar en la limpieza del pool.
- Monitorear **métricas**: tasa de préstamos/espera, conexiones activas/ociosas, timeouts.

6. Mini-ejemplo conceptual (sin implementación de pool)

> Objetivo: visualizar el **patrón de uso** con `DataSource`. El cierre devuelve al pool.

```
// 1) Configuración básica del pool
HikariConfig config = new HikariConfig();
config.setJdbcUrl("jdbc:postgresql://localhost:5432/sakila");
config.setUsername("appuser");
config.setPassword("apppass");
config.setMaximumPoolSize(10);       // máximo de conexiones concurrentes
config.setMinimumIdle(2);            // conexiones en reposo "calientes"
config.setIdleTimeout(30000);        // 30 segundos antes de cerrar
inactivas
config.setMaxLifetime(1800000);      // 30 min antes de reciclar conexión
config.setConnectionTimeout(10000);  // timeout al pedir una conexión

// 2) Crear el DataSource (el pool en sí)
DataSource dataSource = new HikariDataSource(config);

// 3) Usar el DataSource como siempre con JDBC
try (Connection conn = ds.getConnection();
     PreparedStatement ps = conn.prepareStatement(
         "SELECT film_id, title FROM film WHERE rating = ? LIMIT 10");) {
```

```
        ps.setString(1, "PG");
        try (ResultSet rs = ps.executeQuery()) {
            while (rs.next()) {
                int id = rs.getInt("film_id");
                String title = rs.getString("title");
                // ... usar datos
            }
        }
    } // <- aquí no se cierra físicamente: vuelve al pool

    // 4) Cerrar el pool explícitamente al terminar el programa
    ((HikariDataSource) dataSource).close();
```

> Nota: Este ejemplo es solo un puente entre lo que vimos de la URL de conexión y credenciales y la obtención de la conexión a través de un pool. El bloque de código mostrado no tendría sentido en un programa real si quedara así lineal, ya que el valor del pool surge cuando se centraliza en un repositorio común (por ejemplo, un singleton proveedor de conexiones) y múltiples procesos/hilos solicitan conexiones de allí de manera concurrente.

7. Mapeo conceptual → propiedades (HikariCP / Spring Boot)

Cuando pasemos a Spring Boot, los conceptos anteriores se traducen en propiedades (prefijo `spring.datasource.hikari.`):

Concepto	HikariCP	Spring Boot (application.yml/properties)
Tamaño máximo	`maximumPoolSize`	`spring.datasource.hikari.maximum-pool-size`
Mínimo inactivas	`minimumIdle`	`spring.datasource.hikari.minimum-idle`
Timeout préstamo	`connectionTimeout`	`spring.datasource.hikari.connection-timeout` (ms)
Timeout inactividad	`idleTimeout`	`spring.datasource.hikari.idle-timeout` (ms)
Vida máxima	`maxLifetime`	`spring.datasource.hikari.max-lifetime` (ms)
Keep-alive (opcional)	`keepaliveTime`	`spring.datasource.hikari.keepalive-time` (ms)
Detección de fugas	`leakDetectionThreshold`	`spring.datasource.hikari.leak-detection-threshold` (ms)
Nombre del pool	`poolName`	`spring.datasource.hikari.pool-name`

> Además: la **fuente de datos** se define con `spring.datasource.url`, `username`, `password`, `driver-class-name`.

8. Checklist para dimensionar el pool (inicio)

1. **Medí** latencia p50/p95 de las consultas típicas.
2. **Estimá** el throughput objetivo (req/s) y si **cada** request golpea la BD o solo una fracción.
3. Aplicá *Little's Law* para obtener una concurrencia razonable inicial.
4. Ajustá `maximumPoolSize` considerando límites del servidor BD (workers, RAM, etc.).
5. Dejá `minimumIdle` en 20–40% del máximo para absorber picos sin crear de golpe.
6. Elegí `maxLifetime` **menor** al timeout del servidor/balanceador.
7. Activá métricas y **observá**: si hay `connectionTimeout`, el pool está corto o hay fugas/consultas lentas.

Resumen final

Un pool de conexiones es **crítico** para aplicaciones concurrentes: reduce latencias, estabiliza el throughput y protege al servidor de base de datos. Entender **qué representa** cada parámetro te permitirá **configurarlo con criterio** cuando pasemos a Spring Data/HikariCP.

Epílogo – ¿Por qué trabajamos con JDBC?

A lo largo de este bloque vimos **cómo conectar una aplicación Java a una base de datos usando JDBC** de forma directa, sin frameworks que nos abstraigan. Puede parecer un esfuerzo grande para una tecnología que rara vez usaremos "a mano" en proyectos reales, pero este recorrido nos deja aprendizajes fundamentales:

- **Conexión y drivers**: entendimos qué significa realmente *conectar* una aplicación a una base, qué papel juega el driver y cómo se construye una URL de conexión.
- **Sentencias y resultados**: aprendimos a distinguir `Statement`, `PreparedStatement` y `CallableStatement`, con sus ventajas, limitaciones y casos de uso.
- **Transacciones y autocommit**: vimos cómo controlar el ciclo de vida de una operación en la base, por qué no siempre conviene confiar en el `autocommit` y cómo hacer `commit` y `rollback`.
- **Errores reales**: nos enfrentamos a problemas típicos (encoding, timezones, restricciones, concurrencia) que son los mismos que después veremos reflejados, de forma más disfrazada, en frameworks de más alto nivel.
- **Fundamento para lo que viene**: con esta base, comprenderemos mejor el funcionamiento de JPA, Hibernate o Spring Data, y podremos interpretar los errores que nos devuelvan, sabiendo que debajo siempre hay JDBC.

En definitiva, **trabajamos con JDBC no porque vayamos a programar así todos los días**, sino porque nos da los **fundamentos** necesarios para usar con criterio las abstracciones modernas y resolver con seguridad los problemas que encontremos más adelante.

Enlaces relacionados

- https://www.xataka.com/basics/api-que-sirve
- https://www.javatpoint.com/java-jdbc
- https://dev.mysql.com/downloads/connector/j/
- https://codigoxules.org/conectar-mysql-utilizando-driver-jdbc-java-mysql-jdbc/