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

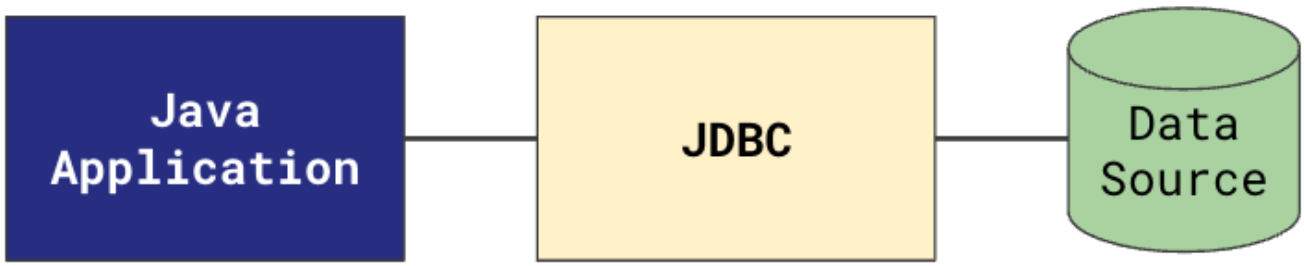
base de datos y conectarse a este desde la aplicación.

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.

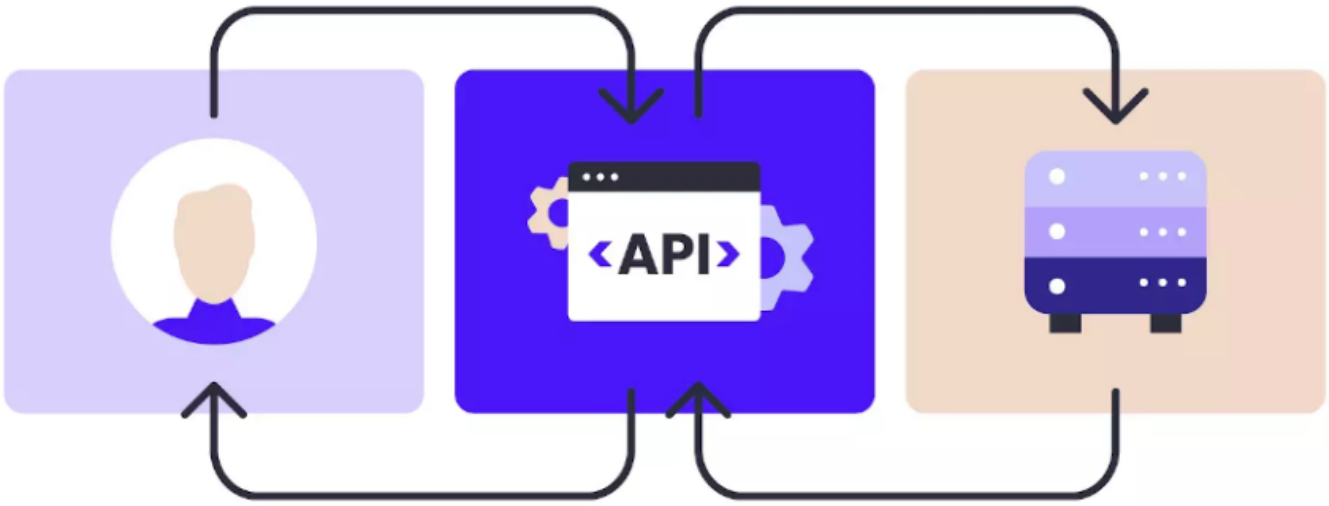
JDBC - API

JDBC utiliza drivers para conectarse a la DB.

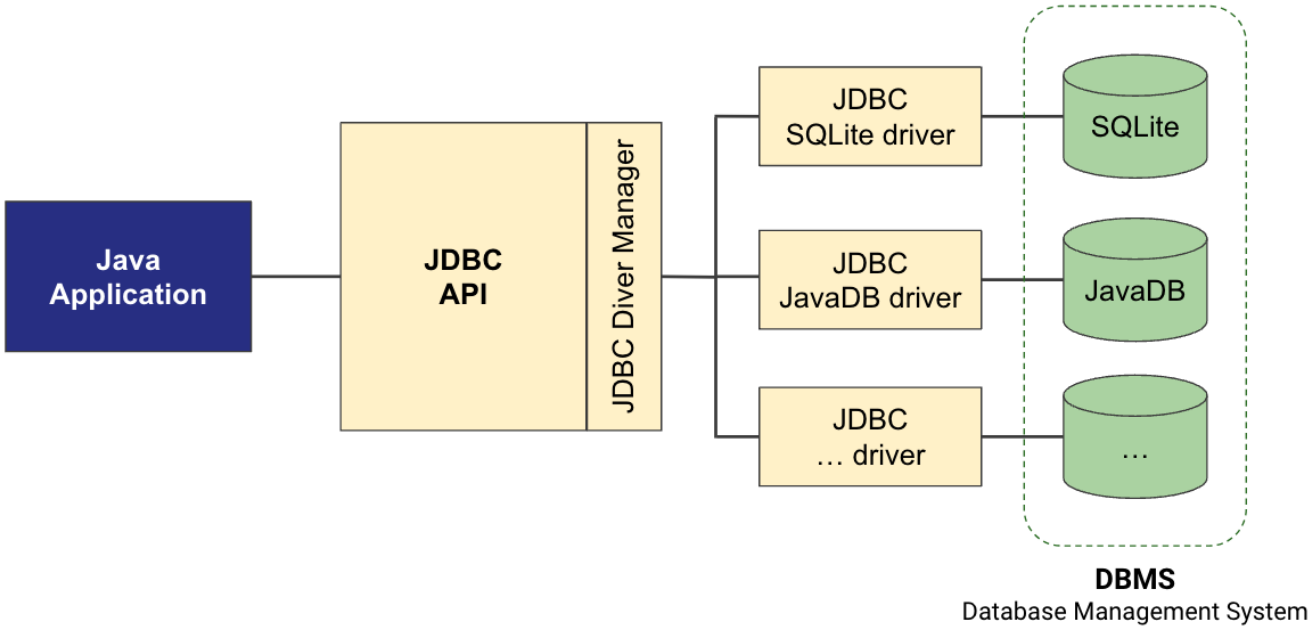


API (Application Programming Interfaces)

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

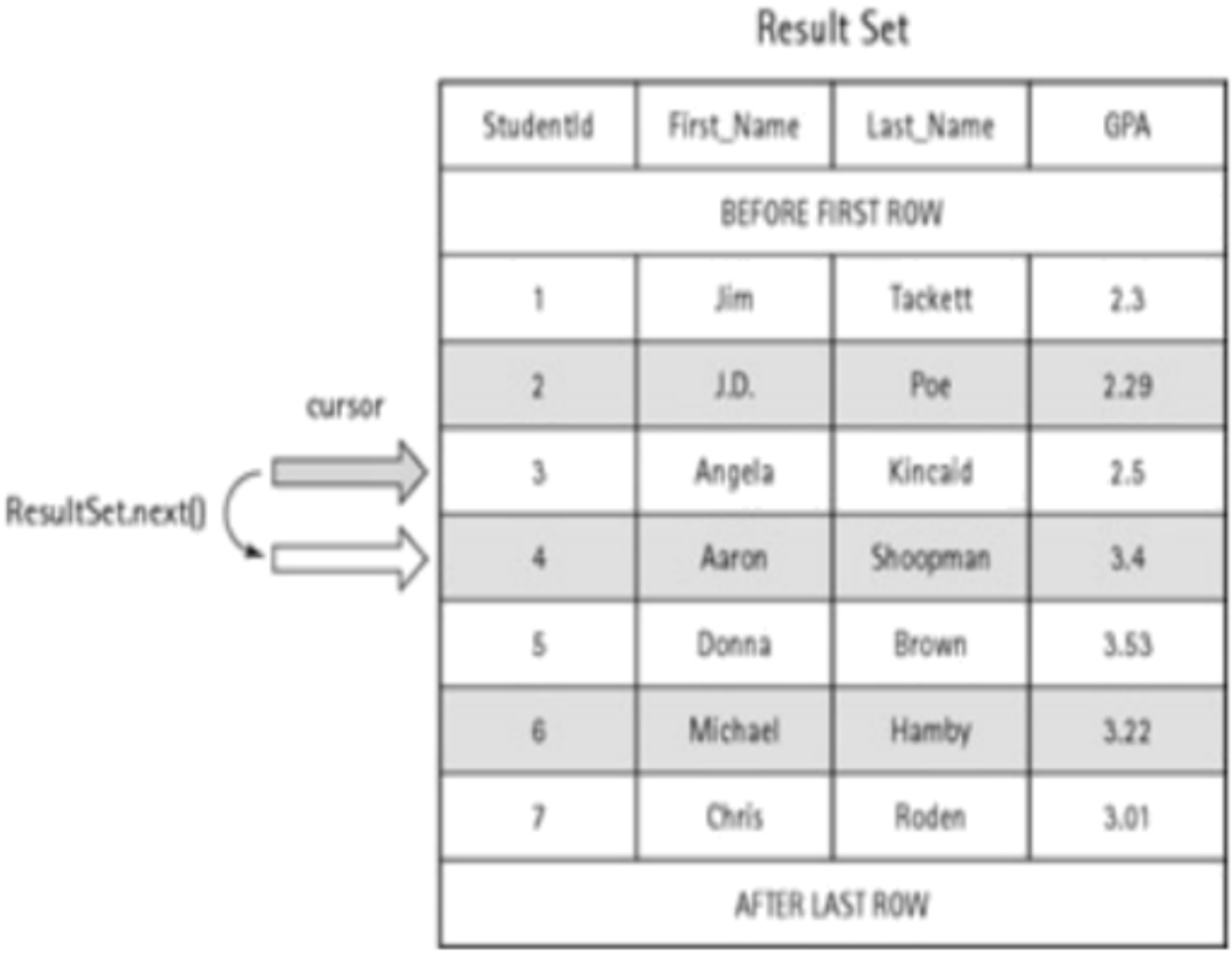
El `ResultSet` es el objeto que representa el resultado. No carga toda la información en memoria, internamente tiene un cursor que apunta a un fila concreta del resultado en la base de datos.

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/

UTN FACULTAD REGIONAL CÓRDOBA
INGENIERÍA EN SISTEMAS DE INFORMACIÓN

Backend de Aplicaciones
Apunte de Clases

Apunte 13 - ORM con JPA (Java Persistence API)

Introducción y fundamentos de ORM

El problema de la impedancia objeto–relacional

En la programación orientada a objetos (POO) trabajamos con **clases, objetos, herencia, polimorfismo y asociaciones**. En cambio, en las bases de datos relacionales usamos **tablas, filas, claves primarias, claves foráneas y joins**. Estos dos mundos no encajan de manera natural: a esto se lo conoce como el **problema de la impedancia objeto–relacional**.

Ejemplos de desajustes típicos:

- **Objetos vs. Tablas**: un objeto puede tener jerarquía y composición, mientras que una tabla es plana.
- **Identidad**: en Java la identidad es la referencia en memoria; en la base se define con claves primarias.
- **Relaciones**: en Java tenemos referencias y colecciones; en la base tenemos claves foráneas y tablas de unión.
- **Herencia**: no existe de forma directa en el modelo relacional.

Estrategias históricas

Antes de los ORM, las aplicaciones Java usaban directamente:

- **JDBC** con sentencias SQL escritas a mano.
- **DAOs (Data Access Objects)** que encapsulaban las consultas.

Esto funcionaba, pero resultaba repetitivo, propenso a errores y difícil de mantener en proyectos grandes.

El concepto de ORM

Un **ORM (Object Relational Mapping)** es una técnica que permite mapear clases y objetos de un lenguaje de programación a tablas y registros de una base de datos. De esta manera:

- Las entidades del dominio se representan como objetos Java.
- El ORM se encarga de traducir operaciones sobre los objetos en sentencias SQL.

Ventajas de un ORM

- **Productividad**: menos código SQL repetitivo.
- **Portabilidad**: cambio de motor de base de datos sin reescribir todas las consultas.
- **Mantenibilidad**: las entidades se describen en un único lugar y reflejan el modelo del dominio.
- **Integración**: se conecta naturalmente con frameworks de alto nivel (ej. Spring Data).

Riesgos y limitaciones

- **Sobrecarga de abstracción**: ocultar el SQL puede generar consultas ineficientes.
- **Consultas complejas**: no siempre son fáciles de expresar en JPQL/Criteria.
- **Tuning**: a veces es necesario optimizar con SQL nativo.

El estándar JPA y Hibernate

- **JPA (Jakarta Persistence API)**: especificación estándar para ORM en Java.
- **Hibernate**: la implementación más utilizada de JPA.

Con JPA definimos **interfaces y anotaciones estándar**; con Hibernate tenemos la implementación concreta que ejecuta las operaciones.

Características principales:

- **Entidades:** Las clases de tu modelo se anotan como entidades (@Entity), lo que permite a JPA saber qué clases deben ser mapeadas a tablas en la base de datos.
- **EntityManager:** El EntityManager es el punto de entrada para realizar operaciones CRUD en tus entidades.
- **JPQL:** JPA introduce un lenguaje de consulta llamado JPQL (Java Persistence Query Language) que se utiliza para realizar consultas de manera similar a SQL pero orientado a objetos.

JPA - Historia y Evolución

- **EJB 2.x / Entity Beans:** Antes de JPA, la persistencia se hacía con Entity Beans en EJB 2 .x, una solución compleja y difícil de manejar.
- **JPA 1.0 (2006):** Introducido con Java EE 5 (JSR 220), buscaba simplificar la persistencia.
- **JPA 2.0 (2009):** Parte de Java EE 6 (JSR 317), incorporó Criteria API, mejoras en el mapping.
- **JPA 2.1 (2013):** En Java EE 7 (JSR 338), agregó converters, stored procedures, entity graphs.
- **JPA 2.2 (2017):** En Java EE 8; añadió streaming, anotaciones repetibles y tipos de fecha/hora de Java 8.
- **Jakarta Persistence 3.1 (2022):** Renombrado como parte de Jakarta EE 10; incluye funciones JPQL nuevas y mejor soporte para UUID.
- **Jakarta Persistence 3.2 (2024):** Mejoras en la API y funcionalidades de JPQL.

Importancia de JPA en el Ecosistema Java

- **Independencia de proveedor:** Podemos cambiar entre Hibernate, EclipseLink, OpenJPA, etc., sin cambiar el código.
- **Desacoplamiento:** Separamos los objetos de dominio de la lógica de persistencia con anotaciones estándar.
- **Productividad:** Menos boilerplate, consultas legibles en JPQL.
- **Estándar de facto:** JPA (hoy Jakarta Persistence) es ampliamente aceptado en aplicaciones empresariales Java.

JPA Implementación y Uso

La especificación JPA se centra en 3 ejes fundamentales a la hora de formalizar el Mapeo Objeto Relacional en Java. Estos 3 ejes van a ser los 3 ejes fundamentales de estudio y a los que le vamos a tener que dedicar esfuerzo para comprender su funcionamiento y capacidades de control. Estas serán las capacidad que nos permitan modificar la Implementación subyacente para que responda de acuerdo con las decisiones de diseño que hemos tomado para el proyecto y se adecúe de la mejor manera a nuestro esquema funcional.

Estos 3 ejes fundamentales nos en órden de necesidad para la implementación:

- **Configuración** donde especificaremos la conexión a la base de datos, la implementación subyacente, el esquema de transacciones y las configuraciones específicas de la implementación.
- **Mapeo** donde realizaremos la conexión entre los atributos de las clases de entidad o Entidades JPA y las columnas de las tablas de la base de datos, además de marcar las restricciones o configuraciones específicas asociadas a dichos mapeos.
- **Administración de Entidades y Consultas** finalmente el código en el que aprovecharemos el uso del EntityManager para crear, actualizar o borrar entidades y las consultas JPQL para obtener listas de entidades a partir de los datos de la base de datos.

A continuación trabajaremos específicamente en cada uno de estos 3 ejes.

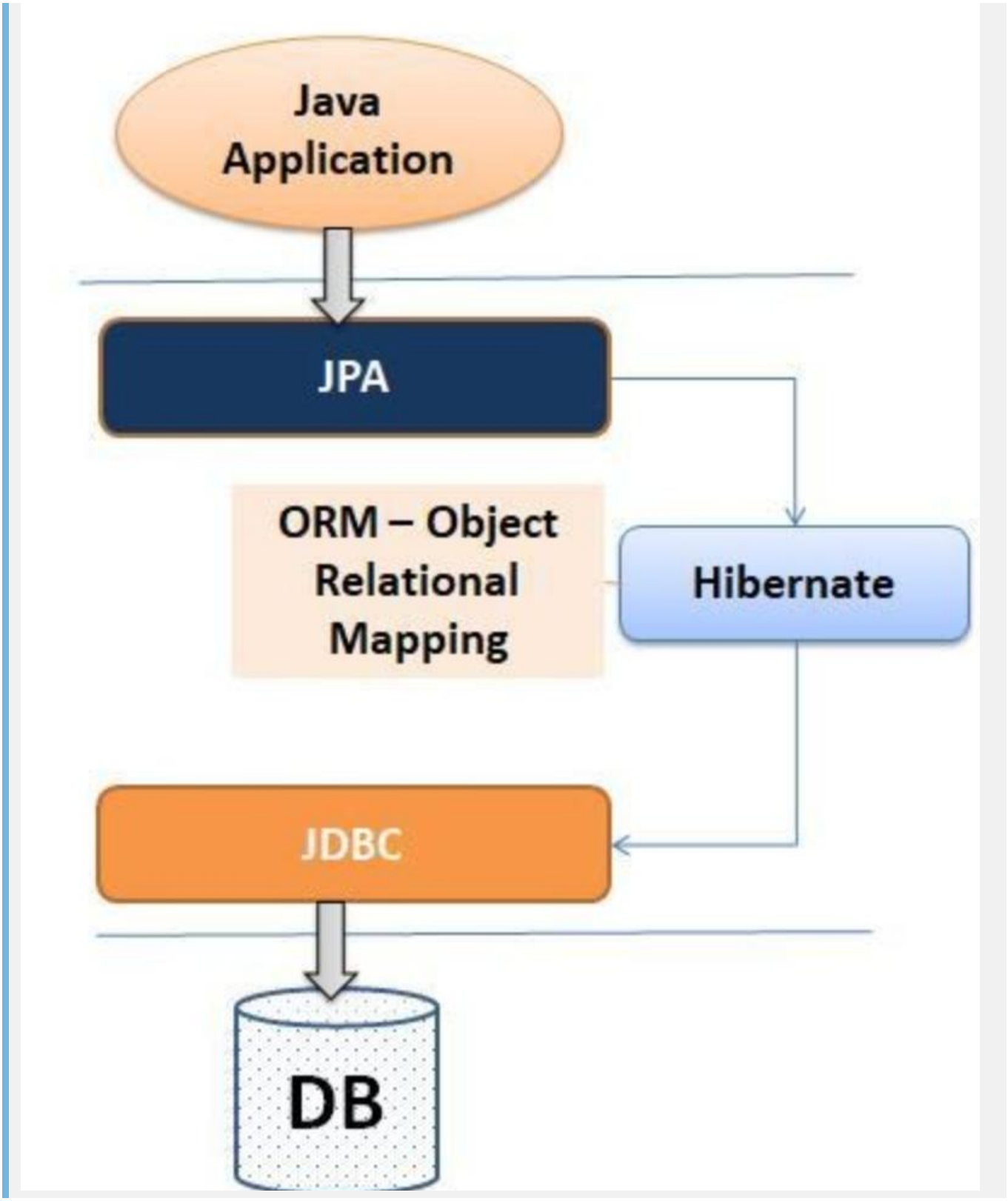
Configuración de JPA con Hibernate

¿Qué estamos configurando?

Antes de escribir código, definimos **qué piezas intervienen** en la persistencia con JPA:

- **API estándar (Jakarta Persistence)**: el contrato de anotaciones e interfaces que usamos en nuestro código.
- **Implementación (Hibernate)**: el motor ORM que ejecuta esas operaciones y traduce a SQL.
- **Driver JDBC**: el conector específico del motor (H2, PostgreSQL, etc.).
- **Pool de conexiones**: opcional en estos labs, clave en producción para eficiencia.

> Objetivo: que el alumno entienda **para qué sirve cada dependencia** y cómo encaja en la arquitectura (App → JPA → Hibernate → JDBC → Base de datos).



Dependencias Maven necesarias

- **Jakarta Persistence API** (`jakarta.persistence-api`).
- **Hibernate Core** como implementación de referencia.
- **Driver JDBC** del motor que usemos (H2, PostgreSQL, MySQL, etc.).
- Opcional: **pool de conexiones** como HikariCP.

Ejemplo de `pom.xml` mínimo:

```
<dependencies>
  <dependency>
```

```
      <groupId>com.h2database</groupId>
      <artifactId>h2</artifactId>
      <version>2.2.224</version>
      <scope>runtime</scope>
    </dependency>

    <dependency>
      <groupId>jakarta.persistence</groupId>
      <artifactId>jakarta.persistence-api</artifactId>
      <version>3.1.0</version>
    </dependency>

    <dependency>
      <groupId>org.hibernate.orm</groupId>
      <artifactId>hibernate-core</artifactId>
      <version>6.4.4.Final</version>
    </dependency>
  </dependencies>
```

Archivo `persistence.xml`

El archivo `META-INF/persistence.xml` define la **unidad de persistencia** (PU) y sus propiedades. Es el punto central de configuración cuando **no** usamos Spring. Sus bloques clave son:

- **Raíz y versión**: define el esquema XML de Jakarta Persistence.
- `<persistence-unit name=...>`: agrupa la configuración bajo un nombre lógico; nuestra app abrirá esa PU.
- `<provider>`: clase del proveedor JPA (Hibernate). Si se omite, Hibernate suele autodetectarse si está en el classpath.
- **Propiedades JDBC**: driver, URL, usuario y contraseña.
- `hibernate.dialect`: guía a Hibernate para emitir SQL compatible con el motor elegido.
- **Estrategia de DDL (`hibernate.hbm2ddl.auto`)**: controla cómo se crea/valida el esquema.
- **Opciones de log**: `show_sql`, `format_sql`, etc.
- **Transacciones**: `transaction-type` puede ser `RESOURCE_LOCAL` (lo usaremos en los labs) o `JTA` (aplicaciones EE/Spring con gestor transaccional).
- **Exploración de entidades**: por defecto, se escanean las clases anotadas en el classpath de la PU; se puede ajustar con `exclude-unlisted-classes` o `<class>` explícitas.

Ejemplo para H2 en memoria (recomendado para los primeros labs)

```
<persistence xmlns="https://jakarta.ee/xml/ns/persistence" version="3.0">
  <persistence-unit name="MiUnidad" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>

    <properties>
      <!-- JDBC (H2 en memoria) -->
      <property name="jakarta.persistence.jdbc.driver" value="org.h2.Driver"/>
      <property name="jakarta.persistence.jdbc.url"
value="jdbc:h2:mem:demo;DB_CLOSE_DELAY=-1"/>
      <property name="jakarta.persistence.jdbc.user" value="sa"/>
      <property name="jakarta.persistence.jdbc.password" value=""/>

      <!-- Dialecto específico -->
      <property name="hibernate.dialect" value="org.hibernate.dialect.H2Dialect"/>

      <!-- DDL: para práctica inicial -->
      <property name="hibernate.hbm2ddl.auto" value="create-drop"/>

      <!-- Log -->
      <property name="hibernate.show_sql" value="true"/>
      <property name="hibernate.format_sql" value="true"/>
    </properties>
```

```
  </persistence-unit>
</persistence>
```

Ejemplo alternativo para PostgreSQL (cuando pasemos a motor real)

```
<persistence xmlns="https://jakarta.ee/xml/ns/persistence" version="3.0">
  <persistence-unit name="MiUnidad" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>

    <properties>
      <property name="jakarta.persistence.jdbc.driver" value="org.postgresql.Driver"/>
      <property name="jakarta.persistence.jdbc.url"
value="jdbc:postgresql://localhost:5432/sakila"/>
      <property name="jakarta.persistence.jdbc.user" value="appuser"/>
      <property name="jakarta.persistence.jdbc.password" value="apppass"/>

      <property name="hibernate.dialect" value="org.hibernate.dialect.PostgreSQLDialect"/>

      <!-- En proyectos reales suele usarse 'validate' o 'update'; evitamos 'create' en
producción -->
      <property name="hibernate.hbm2ddl.auto" value="update"/>

      <property name="hibernate.show_sql" value="true"/>
      <property name="hibernate.format_sql" value="true"/>
    </properties>
  </persistence-unit>
</persistence>
```

Fundamentos de cada propiedad clave

- **`jakarta.persistence.jdbc.*`**: datos de conexión. En H2 memoria, `DB_CLOSE_DELAY=-1` mantiene la DB viva mientras exista la JVM.
- **`hibernate.dialect`**: *hints* de SQL para cada motor.
- **`hibernate.hbm2ddl.auto`**:
  - `none`/omitir: no toca el esquema.
  - `validate`: valida que el esquema exista y coincida con las entidades.
  - `update`: aplica cambios incrementales (útil en desarrollo, cuidado con drift).
  - `create`/`create-drop`: crea (y opcionalmente borra) el esquema al iniciar/finalizar.
- **`transaction-type`**:
  - `RESOURCE_LOCAL`: nosotros controlamos transacciones con `EntityTransaction`.
  - `JTA`: un contenedor/gestor externo coordina transacciones.
- **Logging**: `show_sql`/`format_sql` ayudan en el aprendizaje; en producción se recomienda usar un logger SQL y parametrización.

> Regla para `hibernate.hbm2ddl.auto`: **H2 en memoria + `create-drop`** para los primeros labs; luego migramos a **PostgreSQL + `validate`/`update`** para simular entornos reales.

Consideraciones de conexión

- Para proyectos educativos, podemos usar el pool por defecto de Hibernate.
- En producción, conviene un pool dedicado (HikariCP, C3P0, etc.).
- Es importante cerrar el `EntityManagerFactory` al finalizar la aplicación.

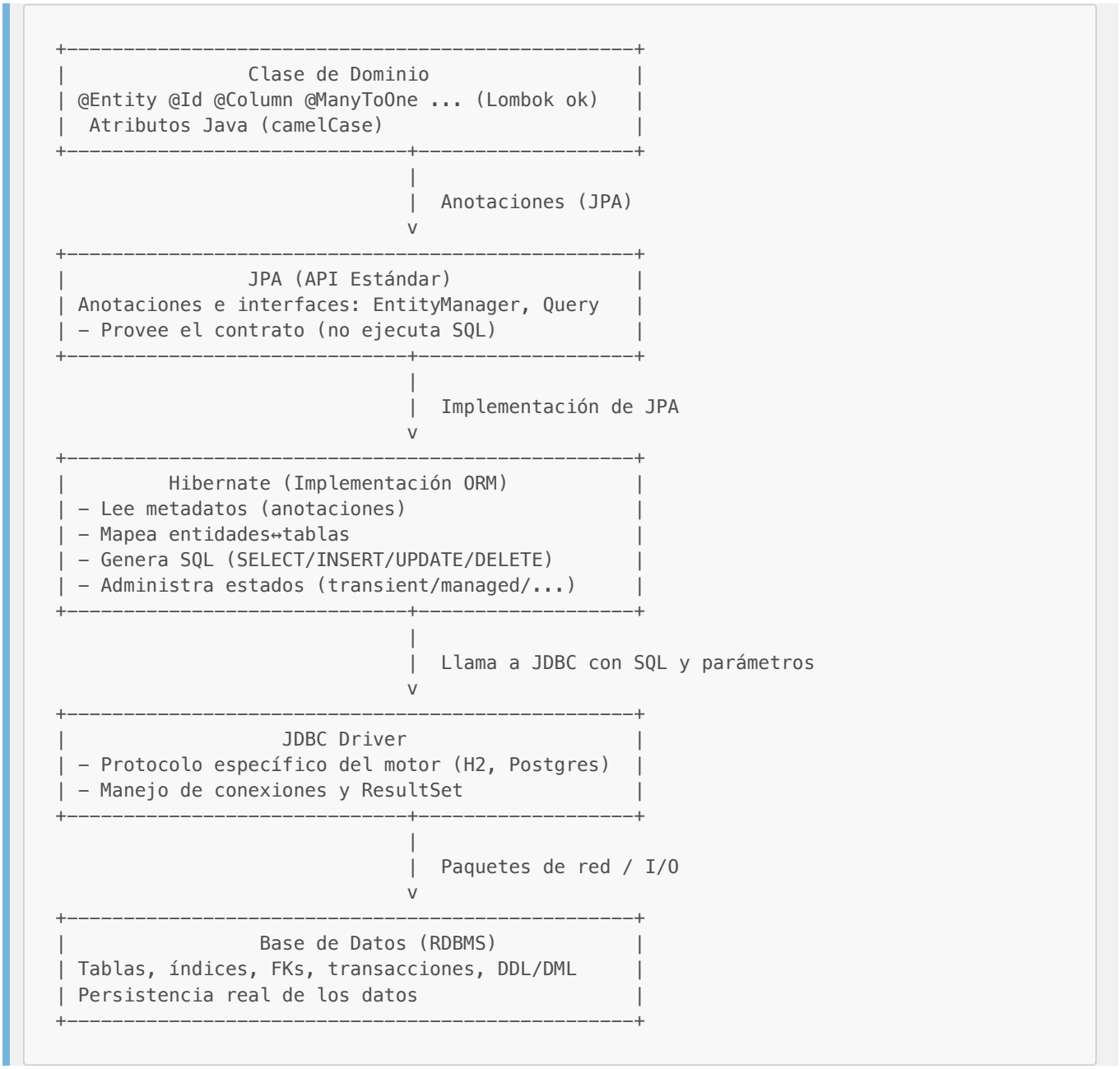
Mapeo de entidades

Proceso de Mapeo: de la tabla a la entidad

El mapeo en JPA consiste en **trazar la correspondencia** entre las estructuras de una base de datos relacional y las clases Java que representan el modelo de dominio.

Tabla ↔ Entidad

- Cada **tabla** de la base de datos se representa como una **clase anotada con `@Entity`**.
- Cada **columna** se convierte en un **atributo de la clase**, usando `@Column` cuando el nombre o propiedades difieren.
- La **clave primaria** se marca con `@Id`, y puede incluir una estrategia de generación (`@GeneratedValue`).
- Las **claves foráneas** se representan con relaciones (`@ManyToOne`, `@OneToMany`, `@ManyToMany`) según corresponda.



Restricciones generales (mínimas para que el mapeo funcione)

- **Entidad válida**
  - Debe estar anotada con `@Entity`.
  - Debe tener **constructor por defecto** (público o protegido).
  - Debe declarar **al menos una clave primaria** con `@Id` (o clave compuesta con `@EmbeddedId` / `@IdClass`).
  - La clase debe ser pública y no final; los campos persistentes no deben ser `static` ni `transient` (Java).
- **Tabla objetivo existente**
  - Si **no** se especifica `@Table`, JPA asumirá el nombre por convención (normalmente el nombre de la clase).
  - Si el nombre **no coincide** con la tabla real, usar `@Table(name = "NOMBRE_TABLA")`.
  - Si el esquema **no** es el por defecto, usar `@Table(schema = "ESQUEMA")`.

    - La tabla debe **existir** (o generarse vía `hibernate.hbm2ddl.auto`) y tener **PK** definida en DB si trabajamos con `validate`/`update`.

- **Columnas y atributos**
    - Para **diferencias de nombre** entre atributo y columna, usar `@Column(name = "COLUMNA_REAL")`.
    - **Not Null** en DB → `@Column(nullable = false)` en la entidad (coherencia de restricciones).
    - **Unique** por columna → `@Column(unique = true)`; para **unicidad compuesta**, usar `@Table(uniqueConstraints = @UniqueConstraint(columnNames = { ... }))`.
    - **Longitud** de `VARCHAR` → `@Column(length = N)`.
    - **Monetarios/precisión** → `BigDecimal` con `@Column(precision = P, scale = S)`.
    - **Enums** → `@Enumerated(EnumType.STRING)` (evita problemas si cambia el ordinal).
    - **Fechas/tiempos** → `java.time.*`; mapear con `@Temporal` solo si usás `java.util.Date/Calendar`.
- **Claves primarias y estrategias**
    - `@GeneratedValue(strategy = GenerationType.IDENTITY)` para autoincrement (H2/PostgreSQL).
    - `SEQUENCE` con `@SequenceGenerator` cuando uses secuencias nativas (muy común en PostgreSQL).
    - Para **claves compuestas**:
        - `@EmbeddedId` con clase `@Embeddable` (preferido por claridad), **o**
        - `@IdClass` (requiere duplicar campos de PK en la entidad y en la clase Id).
    - La PK debe ser **inmutable** a nivel semántico: evitá modificarla en runtime.
- **Nomenclatura: puente DB ↔ Java**
    - En DB solemos usar **UPPER_SNAKE_CASE** (p. ej., `FIRST_NAME`).
    - En Java usamos **camelCase** (`firstName`).
    - Resolver con `@Table(name = "TABLA_EN_DB")` y `@Column(name = "COLUMNA_EN_DB")`.
    - Si el motor es **case-sensitive** con identificadores entrecomillados, mantené la coherencia: mejor **evitar comillas** en DDL y mapear con nombres simples (H2 y Postgres sin comillas → nombres en minúscula normalizados).
- **Estados y DDL**
    - Si `hibernate.hbm2ddl.auto = validate`, el esquema **debe existir** y coincidir con el mapeo (tabla, columnas, tipos, PK/UK/NN).
    - Si `update`, Hibernate intentará ajustar el esquema (útil en desarrollo, con cautela).
    - Para labs con **H2 en memoria**, usar `create-drop` facilita la iteración rápida.
- **Relaciones (visión rápida)**
    - **Muchos→uno**: `@ManyToOne(fetch = LAZY)` con `@JoinColumn(name = "FK")` y tipos compatibles con la PK de la entidad objetivo.
    - **Uno→muchos**: usar con **cautela** (riesgo de cargas grandes). Preferir la navegación por consultas JPQL específicas.
    - **Muchos↔muchos**: preferible modelar **entidad intermedia** si necesitás atributos en la relación.

Checklist rápido de una entidad mínima

- `@Entity` presente.
- **PK** con `@Id` (y `@GeneratedValue` si corresponde).
- `@Table(name = "...", schema = "...")` si el nombre/esquema no coinciden.
- Atributos mapeados con `@Column(name = "...")` cuando el nombre difiere (snake_case ↔ camelCase).
- Restricciones alineadas: `nullable`, `unique`, `length`, `precision/scale`.
- Constructor por defecto y clase pública.
- Para relaciones: `@ManyToOne` + `@JoinColumn` con `fetch = LAZY`.

> Regla práctica en la cátedra: comenzamos con **entidades planas** y **N→1**; recién después, y solo si es necesario, agregamos **1→N** y consideramos **M↔M**. De esta forma mantenemos la consistencia con el esquema relacional y a la vez seguimos las convenciones de codificación de Java.

En este bloque mapeamos entidades de **Chinook** (versión simplificada, ya que luego lo haremos con Spring Data) usando **Lombok** y **JPA (Hibernate)** sobre **H2 en memoria**.

Empezamos con entidades planas, luego agregamos **relaciones muchos-a-uno** (con menos riesgo), mencionamos **uno-a-muchos** (con cautela), y cerramos con **muchos-a-muchos** a modo informativo.

Caso 1 - Entidades planas (sin relaciones)

Anotaciones fundamentales

- `@Entity`: marca la clase como entidad JPA (debe tener constructor por defecto).
- `@Table(name = "...")`: nombre de la tabla (si difiere del nombre de clase).
- `@Id`: identifica la PK.
- `@GeneratedValue(...)`: estrategia de generación de PK.
  - `IDENTITY`: autoincrement en DB.
  - `SEQUENCE`: usa secuencias (PostgreSQL, Oracle).
  - `AUTO`: deja al proveedor decidir.
- `@Column(name = "...", nullable = ..., length = ..., unique = ...)`: mapea columna y restricciones.

Lombok: recomendaciones

- `@Data`: getters/setters, `equals/hashCode`, `toString`.
- `@NoArgsConstructor` y `@AllArgsConstructor`: constructores.
- `@Builder`: patrón builder (útil en tests y labs).
- **Cuidado**: en entidades con relaciones bidireccionales, evitar ciclos en `toString`/`equals`. Podés usar `@ToString(exclude = ...)` y `@EqualsAndHashCode(of = "id")`.
  Estos son algunos de los problemas que hay que tener en cuenta al manejar el mapeo de relaciones.

Ejemplo: `Genre` (plano)

```
package utnfc.back.isi.jpa.domain;

import jakarta.persistence.*;
import lombok.*;

@Entity
@Table(name = "GENRE")
@Data @NoArgsConstructor @AllArgsConstructor @Builder
@EqualsAndHashCode(of = "id")
public class Genre {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  @Column(name = "GENRE_ID")
  private Integer id;

  @Column(name = "NAME", length = 120, nullable = false)
  private String name;
}
```

Ejemplo: `MediaType` (plano)

```
@Entity
@Table(name = "MEDIA_TYPE")
@Data @NoArgsConstructor @AllArgsConstructor @Builder
@EqualsAndHashCode(of = "id")
public class MediaType {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  @Column(name = "MEDIA_TYPE_ID")
  private Integer id;

  @Column(name = "NAME", length = 120, nullable = false)
```

```
  private String name;
}
```

Mini-ejemplo (persistencia básica en H2)

```
// Dentro de un método de prueba/lab
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();
Genre rock = Genre.builder().name("Rock").build();
em.persist(rock);
em.getTransaction().commit();
em.close();
```

Caso 2 - Relaciones muchos-a-uno (preferidas)

En este tipo de relaciones cada objeto se relaciona con otro único objeto haciendo referencia a su clave primaria. Esto mapeado en nuestra entidad provoca un nuevo atributo que contendrá una instancia del objeto de la tabla relacionada.

¿Por qué preferimos relaciones Muchos→Uno (N→1)?

En la práctica docente y en proyectos reales solemos comenzar modelando **desde el lado muchos hacia el lado uno**. La razón principal es que este tipo de relación:

- Nos garantiza que **cada fila de la tabla N solo añade la referencia a un objeto adicional** (el del lado 1).
- La carga de datos queda **acotada a un único objeto extra**, sin riesgo de traer colecciones completas por accidente.
- Evitamos los problemas de **carga masiva involuntaria** que suelen aparecer en las relaciones 1→N (colecciones potencialmente enormes).
- Es más **natural de implementar**: la tabla con la clave foránea es la que conoce a quién pertenece, por lo que el mapeo `@ManyToOne` + `@JoinColumn` refleja fielmente la estructura del esquema.
- Permite **consultas más predecibles y eficientes**, ya que Hibernate resuelve la clave ajena mediante un `JOIN` o una carga diferida de un solo objeto.

> En síntesis: las relaciones N→1 nos dan **simplicidad, previsibilidad y menor riesgo**. Por eso proponemos como primera opción al introducir el tema de asociaciones en JPA.

Anotaciones clave

- `@ManyToOne(fetch = FetchType.LAZY, optional = false)`: relación N→1.
- `@JoinColumn(name = "...", nullable = ...)`: columna FK en la tabla de la entidad propietaria.
- **Fundamento**: modelar **desde el lado N** reduce el riesgo de cargas masivas involuntarias y simplifica el grafo.

Ejemplo: `Album` → `Artist` (N→1)

```
@Entity
@Table(name = "ALBUM")
@Data @NoArgsConstructor @AllArgsConstructor @Builder
@EqualsAndHashCode(of = "id")
public class Album {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  @Column(name = "ALBUM_ID")
  private Integer id;

  @Column(name = "TITLE", length = 200, nullable = false)
  private String title;

  @ManyToOne(fetch = FetchType.LAZY, optional = false)
  @JoinColumn(name = "ARTIST_ID", nullable = false)
```

```
  @ToString.Exclude
  private Artist artist;
}
```

```
@Entity
@Table(name = "ARTIST")
@Data @NoArgsConstructor @AllArgsConstructor @Builder
@EqualsAndHashCode(of = "id")
public class Artist {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  @Column(name = "ARTIST_ID")
  private Integer id;

  @Column(name = "NAME", length = 160, nullable = false)
  private String name;
}
```

Ejemplo: `Track` → `Album` / `Genre` / `MediaType` (N→1)

```
@Entity
@Table(name = "TRACK")
@Data @NoArgsConstructor @AllArgsConstructor @Builder
@EqualsAndHashCode(of = "id")
public class Track {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  @Column(name = "TRACK_ID")
  private Integer id;

  @Column(name = "NAME", length = 200, nullable = false)
  private String name;

  @ManyToOne(fetch = FetchType.LAZY, optional = false)
  @JoinColumn(name = "ALBUM_ID", nullable = false)
  @ToString.Exclude
  private Album album;

  @ManyToOne(fetch = FetchType.LAZY)
  @JoinColumn(name = "GENRE_ID")
  @ToString.Exclude
  private Genre genre;

  @ManyToOne(fetch = FetchType.LAZY, optional = false)
  @JoinColumn(name = "MEDIA_TYPE_ID", nullable = false)
  @ToString.Exclude
  private MediaType mediaType;

  @Column(name = "MILLISECONDS", nullable = false)
  private Integer milliseconds;

  @Column(name = "UNIT_PRICE", nullable = false)
  private java.math.BigDecimal unitPrice;
}
```

Mini-ejemplo (persistir respetando N→1)

```
em.getTransaction().begin();
Artist artist = Artist.builder().name("AC/DC").build();
em.persist(artist);
Album album = Album.builder().title("Back in Black").artist(artist).build();
em.persist(album);
MediaType mt = MediaType.builder().name("MPEG audio").build();
em.persist(mt);
Genre g = Genre.builder().name("Hard Rock").build();
em.persist(g);
Track t = Track.builder().name("Hells Bells").album(album).genre(g).mediaType(mt)
              .milliseconds(312000).unitPrice(new java.math.BigDecimal("0.99")).build();
em.persist(t);
em.getTransaction().commit();
```

Caso 3 - Uno-a-muchos (con cautela)

¿Qué pasa con las relaciones Uno→Muchos (1→N)?

El mapeo de relaciones 1→N en JPA refleja el lado inverso de una relación N→1: un objeto puede estar asociado a una **colección de muchos otros objetos**.
Si bien es una construcción válida, requiere precauciones:

- **Riesgo de carga masiva**: un `Artist` puede tener cientos de `Album`. Si accedemos a la colección sin cuidado, Hibernate intentará cargar todos los elementos, con el consiguiente impacto de memoria y consultas múltiples (problema conocido como *N+1 queries*).
- **Colecciones grandes**: no siempre tiene sentido traer todos los elementos asociados; muchas veces conviene una consulta específica con JPQL o Criteria para obtener solo lo necesario.
- **Complejidad en equals/hashCode**: incluir colecciones en estos métodos puede generar ciclos infinitos o comparaciones muy costosas.
- **Gestión de cascadas**: al tener colecciones, hay que ser cuidadoso con el uso de `cascade = ALL`, ya que operaciones de borrado o persistencia pueden propagarse sin control.

Buenas prácticas

- Mantener siempre **`fetch = LAZY`** en las colecciones.
- Usar `@ToString.Exclude` y `@EqualsAndHashCode(of = "id")` con Lombok para evitar ciclos.
- Evitar navegar directamente por colecciones grandes: en su lugar, escribir **consultas específicas** (`SELECT a FROM Album a WHERE a.artist.id = :id`).
- Modelar primero el lado N→1 y **agregar el 1→N solo si la navegación es realmente necesaria**.

> En síntesis: las relaciones 1→N son útiles para expresar la navegación desde un objeto padre hacia sus hijos, pero conllevan **más riesgos de rendimiento y complejidad**. Por eso en la cátedra recomendamos usarlas **con cautela** y siempre priorizar las N→1 como base.

Anotaciones clave para 1→N

- `@OneToMany(mappedBy = "...", fetch = FetchType.LAZY, cascade = ...)` define la **vista inversa** del N→1.
- **Riesgo a mitigar**: cargar colecciones grandes por accidente (N+1, explosión de selects).
- Recomendación de cátedra: **comenzar modelando solo el N→1** y agregar 1→N **solo si es necesario** para navegación.

Ejemplo básico: `Artist` con sus `Album`

```
@OneToMany(mappedBy = "artist", fetch = FetchType.LAZY)
@ToString.Exclude
private java.util.List<Album> albums = new java.util.ArrayList<>();
```

> Buenas prácticas: mantener `LAZY`, no usar colecciones en `equals/hashCode`, y preferir consultas JPQL específicas para traer listas.

Ejemplo completo con `Album` y `Track`

```
// Ejemplo completo: Album ↔ Track (1→N y N→1) con manejo de referencia circular

package utnfc.back.isi.jpa.domain;

import jakarta.persistence.*;
import lombok.*;
import java.math.BigDecimal;
import java.util.ArrayList;
import java.util.List;

// ---------------- Entidad Album ----------------
@Entity
@Table(name = "ALBUM")
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
@EqualsAndHashCode(of = "id")
public class Album {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  @Column(name = "ALBUM_ID")
  private Integer id;

  @Column(name = "TITLE", length = 200, nullable = false)
  private String title;

  @OneToMany(mappedBy = "album", fetch = FetchType.LAZY,
             cascade = CascadeType.PERSIST, orphanRemoval = false)
  @ToString.Exclude
  private List<Track> tracks = new ArrayList<>();

  public void addTrack(Track t) {
    tracks.add(t);
    t.setAlbum(this);
  }

  public void removeTrack(Track t) {
    tracks.remove(t);
    t.setAlbum(null);
  }
}

// ---------------- Entidad Track ----------------
@Entity
@Table(name = "TRACK")
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
@EqualsAndHashCode(of = "id")
public class Track {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  @Column(name = "TRACK_ID")
  private Integer id;
```

```
  @Column(name = "NAME", length = 200, nullable = false)
  private String name;

  @ManyToOne(fetch = FetchType.LAZY, optional = false)
  @JoinColumn(name = "ALBUM_ID", nullable = false)
  @ToString.Exclude
  private Album album;

  @Column(name = "MILLISECONDS", nullable = false)
  private Integer milliseconds;

  @Column(name = "UNIT_PRICE", nullable = false, precision = 10, scale = 2)
  private BigDecimal unitPrice;
}

// ---------------- Persistencia mínima (H2) ----------------

EntityManager em = emf.createEntityManager();
em.getTransaction().begin();

Album album = Album.builder().title("Back in Black").build();

Track t1 = Track.builder()
    .name("Hells Bells")
    .milliseconds(312_000)
    .unitPrice(new java.math.BigDecimal("0.99"))
    .build();

Track t2 = Track.builder()
    .name("Shoot to Thrill")
    .milliseconds(315_000)
    .unitPrice(new java.math.BigDecimal("0.99"))
    .build();

album.addTrack(t1);
album.addTrack(t2);

em.persist(album);
em.getTransaction().commit();
em.close();

// ---------------- Consulta básica con find ----------------
Album found = em.find(Album.class, album.getId());
System.out.println("Album: " + found.getTitle());
System.out.println("Tracks: " + found.getTracks().size()); // LAZY: puede disparar un select
```

Caso 4 - Muchos-a-muchos (informativo)

- En Chinook, `Playlist` ↔ `Track` se modela con tabla intermedia `PLAYLIST_TRACK`.
- En JPA, se puede mapear con `@ManyToMany` + `@JoinTable`, pero en la práctica **preferimos modelar la tabla intermedia como entidad** (p. ej., `PlaylistTrack`) para poder agregar atributos (orden, fecha, etc.).

Esquema informativo con `@ManyToMany`

```
@ManyToMany
@JoinTable(name = "PLAYLIST_TRACK",
  joinColumns = @JoinColumn(name = "PLAYLIST_ID"),
  inverseJoinColumns = @JoinColumn(name = "TRACK_ID")
)
@ToString.Exclude
private java.util.Set<Track> tracks = new java.util.HashSet<>();
```

> Recomendación: para los labs, **evitar ManyToMany directo** y usar entidad intermedia si se necesita manipular la relación.

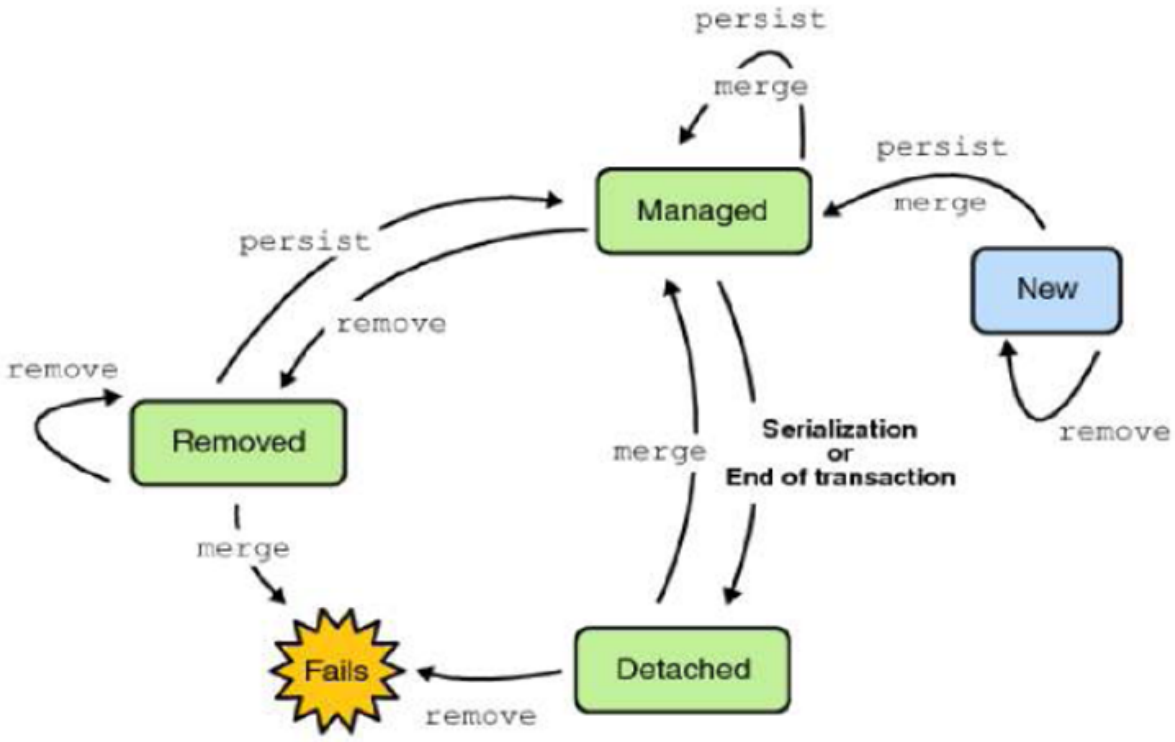
Notas de mapeo y validación

- Conviene alinear **nombres y tipos** con el esquema (todo **UPPER_SNAKE_CASE** si el script lo define así).
- Para valores monetarios, usar `BigDecimal` con precisión/escala adecuadas (`@Column(precision=, scale=)`).
- Usar `@NotNull`, `@Size`, etc. (Bean Validation) en los DTOs/entrada; en entidades solo lo necesario.
- Mantener `fetch = LAZY` por defecto en relaciones, y controlar la carga con JPQL/joins.

4. EntityManager y ciclo de vida de las entidades

El **EntityManager** es el corazón de JPA: gestiona las entidades y controla su estado dentro del *contexto de persistencia*. Una entidad puede atravesar distintos estados durante su vida útil, y conocerlos nos ayuda a evitar errores y entender los efectos de cada operación.

El EntityManager actúa como contenedor y proveedor de entidades gestionadas. Cuando una entidad entra en su persistence context, el EntityManager pasa a controlar su estado, sincronizar cambios y coordinar su persistencia en la base de datos. Esto significa que las entidades bajo su control siguen un ciclo de vida administrado.



Estados del ciclo de vida

- **New (Transient)**
  - Objeto recién creado en Java.
  - No está asociado a ningún contexto de persistencia.
  - No existe en la base de datos.
  - Operaciones válidas:
    - `persist()` → pasa a *Managed*.
    - `remove()` → no tiene efecto.
    - `merge()` → crea un nuevo objeto *Managed* con los mismos datos.
- **Managed (Persistente)**

    - Entidad dentro del *persistence context*.
    - Cualquier cambio en sus atributos será detectado y sincronizado con la BD al hacer `commit` o `flush()`.
    - Operaciones válidas:
        - `remove()` → pasa a *Removed*.
        - `detach()` / fin de la transacción → pasa a *Detached*.
        - `merge()` → mantiene estado *Managed*.
        - `persist()` → no tiene efecto (ya está gestionada).

- **Detached**
    - Entidad que fue *Managed* pero ya no pertenece al *persistence context* (porque cerramos el `EntityManager`, terminó la transacción, o se llamó a `detach()`).
    - Sus cambios **no se sincronizan automáticamente** con la BD.
    - Operaciones válidas:
        - `merge()` → crea/actualiza una copia *Managed* con los cambios.
        - `remove()` → provoca error (`IllegalArgumentException`) si no se vuelve a adjuntar.
        - `persist()` → vuelve a insertar (puede dar error de PK duplicada).

- **Removed**
    - Entidad marcada para eliminación en el *persistence context*.
    - Al hacer `commit`, se ejecuta `DELETE` en la base.
    - Operaciones válidas:
        - `persist()` → puede revertir la eliminación, vuelve a *Managed*.
        - `merge()` → vuelve a *Managed*.
        - `remove()` → no tiene efecto adicional.

Diagrama simplificado de transiciones

```
    persist
New ----------> Managed
|               |   |
|               |   | detach / commit end
v       merge   |   v
Managed <-------- Detached
  |                 |
  v      remove     |
Removed <-------------
```

Métodos principales del EntityManager

Método	Uso principal	Estado de entrada esperado	Resultado	Riesgos/Observaciones
`persist()`	Insertar una nueva entidad	NEW (Transient)	Pasa a Managed, se inserta al hacer commit/flush	Sobre *Managed* no hace nada, sobre *Detached* puede intentar reinserción (PK duplicada).
`merge()`	Reincorporar entidad detached o actualizar estado	DETACHED o NEW	Devuelve copia *Managed* con cambios aplicados	No actualiza el objeto original, sino una nueva instancia. Seguro en la mayoría de escenarios.

Método	Uso principal	Estado de entrada esperado	Resultado	Riesgos/Observaciones
`remove()`	Marcar para eliminación	MANAGED	Pasa a Removed, se ejecuta DELETE en commit	Falla si la entidad está DETACHED o NEW.
`detach()`	Desasociar del contexto	MANAGED	Entidad pasa a Detached	Cambios posteriores no se sincronizan.
`refresh()`	Sincronizar con BD descartando cambios	MANAGED	Sobrescribe el estado con datos actuales de BD	Cambios no confirmados se pierden.
`flush()`	Forzar sincronización inmediata	MANAGED	Ejecuta SQL pendiente sin cerrar transacción	Puede exponer errores antes del commit.

> [!Note] **Pensar el ciclo de vida como un grafo de estados** ayuda a comprender los efectos de cada método del `EntityManager`.

Contexto de persistencia y obtención del EntityManager

El contexto de persistencia es el conjunto de entidades gestionadas que mantiene el EntityManager en memoria mientras dura la transacción. El primer paso para trabajar con él es obtener una instancia de EntityManager desde el EntityManagerFactory (patrón Factory). En entornos de servidores Java EE/Jakarta EE o con Spring, normalmente el propio contenedor es quien administra y provee estas instancias de forma transparente. El persistence.xml establece la configuración de la unidad de persistencia, y es el puente que conecta la definición de la base de datos con el EntityManager.

Obtención del EntityManagerFactory (EMF)

Antes de obtener un EntityManager, debemos crear una instancia de EntityManagerFactory. Este se construye a partir del nombre de la unidad de persistencia definida en persistence.xml y actúa como el puente entre la configuración y la aplicación:

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("MiUnidad");
EntityManager em = emf.createEntityManager();
```

En aplicaciones de servidor (Java EE/Jakarta EE o Spring), normalmente el contenedor administra el EntityManagerFactory y provee directamente el EntityManager a través de inyección de dependencias (@PersistenceContext).

Ejemplo básico

```
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();

Artist artist = new Artist();          // Estado: NEW
artist.setName("AC/DC");

em.persist(artist);                    // Estado: MANAGED
artist.setName("ACDC");                // Cambios se detectan

em.detach(artist);                     // Estado: DETACHED
artist.setName("AC-DC");               // Cambios NO se sincronizan

Artist merged = em.merge(artist);      // Crea copia MANAGED
em.remove(merged);                     // Estado: REMOVED

em.getTransaction().commit();
em.close();
```

Ejemplo alternativo: `find` → actualizar → `detach`

Este flujo muestra cómo **modificar una entidad recuperada**, **persistir sus cambios** y luego **desasociarla** del contexto y **devolverla al llamador** con los cambios.

```
// Supongamos que existe un Artist con id conocido
Integer id = 1;

EntityManager em = emf.createEntityManager();
em.getTransaction().begin();

// 1) Recuperamos (MANAGED)
Artist a = em.find(Artist.class, id);
String nombreOriginal = a.getName();

// 2) Actualizamos mientras está MANAGED (JPA *podría* sincronizar en commit)
a.setName("Nombre Propuesto");

// 5) Commit → al estar en estado Managed, el cambio se sincroniza en la BD
//    Detecta todas las entidades actualizadas y realiza las modificaciones en la BD
em.getTransaction().commit();
// 6) Las entidades pasan a estado detached
em.close();

// 7) Retornamos el objeto fuera del contexto de persistencia
return a;

// ------------------------- Prueba -----------------------------------------

// Verificación: abrimos un nuevo EM y consultamos nuevamente
EntityManager em2 = emf.createEntityManager();
Artist verificado = em2.find(Artist.class, id);
System.out.println("DB ahora con: " + verificado.getName()); // imprime Nombre Propuesto
em2.close();
```

> Esto es comun en contextos donde necesitamos recibir los cambios desde otra capa de la aplicación realizar la persistencia de estos cambios y finalmente devolver a la otra capa la entidad modificada pero ya fuera del contexto de persistencia.

Comparativa de acciones tras un `find()` → modificar → commit

Acción aplicada a la entidad `a`	¿Se persiste el cambio?	SQL generado	Estado de `a` durante TX	Estado de `a` después del commit (cerrando EM)	Notas clave
Nada especial (solo commit)	✅ Sí	`UPDATE`	Managed	Detached	Caso normal: al ser *Managed*, sincroniza solo.
`em.flush()` antes de `commit`	✅ Sí	`UPDATE`	Managed	Detached	Fuerza sincronizar antes; detecta errores temprano.
`em.detach(a)` antes de `commit`	❌ No	—	Detached	Detached	Al sacarla del contexto, no se sincroniza.
`em.persist(a)`	❌ No (sin efecto)	—	Managed	Detached	`persist()` solo aplica a *NEW*. Sobre *Managed* no hace nada.
`em.merge(a)`	⚠ Sí, pero en copia	`UPDATE`	Retorno de `merge` = Managed	Retorno de `merge` = Detached	La instancia devuelta es la *Managed*, la original queda *Detached*.
`em.remove(a)`	❌ Se borra	`DELETE`	Removed	Detached (ya sin fila en BD)	Marca para borrar, no para actualizar.

Acción aplicada a la entidad `a`	¿Se persiste el cambio?	SQL generado	Estado de `a` durante TX	Estado de `a` después del commit (cerrando EM)	Notas clave
`em.clear()` (sobre EM completo)	❌ No	—	Todo Detached	Detached	Vacía el contexto, no se sincroniza salvo flush previo.
`em.refresh(a)`	❌ No (descarta cambios)	`SELECT`	Managed	Detached	Pisa los cambios en memoria con lo que está en la BD.

> [!TIP] **al cerrar el `EntityManager`, todas las entidades Managed pasan a Detached**.
> Si queremos seguir trabajando con ellas en otra transacción, debemos **recuperarlas otra vez** o hacer un `merge`.

Comparativa rápida de métodos del `EntityManager`

Método	¿Crea registro si no existe?	¿Actualiza si existe?	Estado requerido de la entidad pasada	Devuelve	Cuándo usar	Riesgos / notas
`persist(e)`	Sí (INSERT)	No	`NEW` (transient)	`void` (la entidad queda *Managed*)	Alta de entidades nuevas	Falla si la PK ya existe; sobre *Managed* no hace nada; sobre *Detached* intenta insertar (duplicado).
`merge(e)`	Sí (INSERT)	Sí (UPDATE)	`NEW` o `DETACHED`	**Nueva instancia *Managed***	Guardar cambios de *detached* o *upsert* manual	¡La instancia retornada es la *Managed*! La original sigue *Detached*. Puede crear inserciones si no existe.
`remove(e)`	No	Sí (DELETE)	`MANAGED`	`void`	Eliminar entidad gestionada	Lanza `IllegalArgumentException` si *Detached*; asegurar que esté *Managed* o hacer `em.remove(em.merge(e))`.
`detach(e)`	No	No	`MANAGED`	`void`	Sacar del contexto (evitar sincronización automática)	Cambios posteriores **no** se persisten salvo `merge`. Útil para ediciones offline/DTO.
`flush()`	N/A	N/A	N/A	`void`	Forzar sincronización pendiente a la BD dentro de la transacción	No cierra la transacción; puede revelar errores de constraints antes del `commit`.
`refresh(e)`	No	No (sobrescribe)	`MANAGED`	`void`	Descartar cambios en memoria y recargar desde BD	Pérdida de cambios no sincronizados; hace SELECT y pisa el estado actual.
`find(C,id)`	No	No	N/A	Instancia *Managed* o `null`	Recuperar entidad por PK	Respeta caché de 1er nivel; puede disparar proxy/lazy en relaciones.

Método	¿Crea registro si no existe?	¿Actualiza si existe?	Estado requerido de la entidad pasada	Devuelve	Cuándo usar	Riesgos / notas
`getReference(C,id)`	No	No	N/A	Proxy *Managed* (lazy)	Referencia perezosa cuando solo se necesita la PK	Acceso a campos no-PK puede disparar carga; puede lanzar `EntityNotFoundException` si no existe.

Operaciones CRUD con JPA (Create, Read, Update, Delete)

Las operaciones CRUD son las operaciones básicas de manipulación de datos en cualquier sistema de gestión de bases de datos. En JPA, estas operaciones se pueden realizar utilizando el `EntityManager`.
En base a los ejemplos anteriores podemos generar un ejemplo de cada una de las operaciones CRUD aprovechando las capacidades del EntityManager.
A continuación se describen estas operaciones en detalle:

Create (Crear)

Para crear una nueva entidad y guardarla en la base de datos, utilizamos el método `persist`.

```
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();
Usuario nuevoUsuario = new Usuario("nombre", "email@example.com");
em.persist(nuevoUsuario);
em.getTransaction().commit();
```

Retrieve (Obtener)

Para leer una entidad de la base de datos, utilizamos el método `find`.

```
EntityManager em = emf.createEntityManager();
Usuario usuarioExistente = em.find(Usuario.class, id);
```

Update (Actualizar)

Para actualizar una entidad ya existente, utilizamos el método `merge`.

```
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();
usuarioExistente.setEmail("nuevo-email@example.com");
em.merge(usuarioExistente);
em.getTransaction().commit();
```

Delete (Eliminar)

Para eliminar una entidad de la base de datos, utilizamos el método `remove`.

```
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();
Usuario usuarioAEliminar = em.find(Usuario.class, id);
```

```
em.remove(usuarioAEliminar);
em.getTransaction().commit();
```

Estas son las operaciones CRUD básicas en JPA. Al comprender estos fundamentos, estarás bien equipado para manejar la persistencia de datos en tus aplicaciones Java.

JPQL (Jakarta Persistence Query Language)

JPQL es el lenguaje de consultas orientado a entidades de JPA. Consulta **clases y sus relaciones** (modelo de objetos), no tablas. El proveedor (Hibernate) traduce JPQL a SQL del motor elegido.

Conceptos clave

- **Dominio de consulta**: clases anotadas con `@Entity` y sus relaciones.
- **Identificadores**: usamos **alias** para entidades (`from Album a`).
- **Expresiones de ruta**: navegación por relaciones (`a.artist.name`, `t.album.title`).
- **Proyecciones**: seleccionamos entidades completas, atributos o DTOs.
- **Parámetros**: `:named` o `?1` (recomendado `:named`).
- **Tipo-safe**: opcional con Criteria API (lo vemos al final como alternativa).

Sintaxis básica

```
select a from Album a
  where a.title like :title
  order by a.id
```

> - `select` puede omitirse si proyectamos la entidad completa: `from Album a`.
> - `where` usa operadores de Java/SQL (`=`, `<>`, `<`, `>`, `between`, `like`, `in`, `is null`).
> - `order by` admite rutas y dirección (`asc`/`desc`).

Joins

- **Inner join**: `select a from Album a join a.artist ar`.
- **Left join**: `select t from Track t left join t.genre g`.
- **Join con alias y filtro**: `select t from Track t join t.album a where a.id = :id`.
- **Fetch join** (carga asociada evitando N+1):

```
select distinct a
  from Album a
  left join fetch a.tracks
  where a.id = :id
```

> `fetch` trae la colección asociada en la misma consulta. Usar con cuidado en colecciones grandes.

Parámetros y paginado

```
List<Album> page = em.createQuery(
    "select a from Album a where a.title like :q order by a.id", Album.class)
  .setParameter("q", "%black%")
  .setFirstResult(0)      // offset
  .setMaxResults(10)      // page size
  .getResultList();
```

- `getSingleResult()` lanza `NoResultException` o `NonUniqueResultException`; preferir `getResultList()` y chequear vacío.

Proyecciones

- **Entidad completa**: `select a from Album a`.
- **Campos escalares**: `select a.id, a.title from Album a` → devuelve `List<Object[]>`.
- **DTO con constructor expression**:

```
select new utnfc.back.isi.jpa.dto.AlbumSummary(a.id, a.title, ar.name)
  from Album a join a.artist ar
  where a.title like :q
```

DTO ejemplo:

```
public record AlbumSummary(Integer id, String title, String artistName) { }
```

Agregaciones y agrupamientos

```
select ar.name, count(a)
  from Album a join a.artist ar
  group by ar.name
  having count(a) > 3
  order by count(a) desc
```

- Funciones comunes: `count`, `sum`, `avg`, `min`, `max`.
- `having` filtra sobre agregaciones.

Subconsultas

```
select t
  from Track t
  where t.unitPrice > (
    select avg(t2.unitPrice) from Track t2 where t2.genre = t.genre
  )
```

- Las subconsultas solo aparecen en `where`/`having` (no en `from`).

Funciones y operaciones útiles

- **Strings**: `lower`, `upper`, `concat`, `length`, `substring`, `trim`.
- **Numéricas**: `abs`, `mod`, `sqrt`.
- **Temporales**: `current_date`, `current_time`, `current_timestamp`.
- **Null-safe**: `coalesce(x, y)`, `nullif(x, y)`.
- **Case**: `case when ... then ... else ... end`.

Consultas de actualización/borrado en bloque

- **Update bulk**:

```
int n = em.createQuery(
  "update Track t set t.unitPrice = t.unitPrice * 1.1 where t.genre.name = :g")
 .setParameter("g", "Rock")
 .executeUpdate();
```

- **Delete bulk**:

```
int m = em.createQuery(
  "delete from Track t where t.album.id = :id")
 .setParameter("id", albumId)
 .executeUpdate();
```

> Son operaciones **bulk**: se ejecutan directamente en la base y **saltan el contexto de persistencia**. Tras usarlas, conviene `em.clear()` para evitar estados inconsistentes.

Named queries

- Declaración en la entidad:

```
@Entity
@NamedQuery(
  name = "Album.findByTitle",
  query = "select a from Album a where lower(a.title) like lower(:q)"
)
public class Album { ... }
```

- Uso:

```
List<Album> res = em.createNamedQuery("Album.findByTitle", Album.class)
  .setParameter("q", "%black%")
  .getResultList();
```

Ejemplos con Chinook (H2 en memoria)

- **Top 10 tracks por precio**:

```
select t
  from Track t
  order by t.unitPrice desc, t.id asc
```

- **Tracks de un álbum**:

```
  select t from Track t where t.album.id = :albumId order by t.id
```

- **Álbum con sus tracks (fetch)**:

```
  select distinct a from Album a left join fetch a.tracks where a.id = :id
```

- **Cantidad de tracks por género**:

```
select g.name, count(t)
  from Track t join t.genre g
  group by g.name
  order by count(t) desc
```

JPQL vs SQL nativo

- **JPQL** consulta **entidades** y entiende **relaciones**; es portable.
- **SQL nativo** consulta **tablas**; usar cuando necesitamos funciones específicas del motor o tuning fino.

```
List<Object[]> rows = em.createNativeQuery(
  "select a.album_id, a.title, count(t.track_id) as tracks " +
  "from album a left join track t on t.album_id = a.album_id " +
  "group by a.album_id, a.title order by tracks desc")
.getResultList();
```

Criteria API (mención)

- Alternativa **type-safe** para construir consultas en Java sin strings.
- Útil cuando generamos consultas dinámicas complejas.

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Album> cq = cb.createQuery(Album.class);
Root<Album> a = cq.from(Album.class);
cq.select(a).where(cb.like(a.get("title"), cb.literal("%black%")));
List<Album> out = em.createQuery(cq).getResultList();
```

Buenas prácticas JPQL

- Mantener `LAZY` por defecto y usar `fetch join` solo en casos puntuales.
- Para colecciones grandes, paginar siempre (`setFirstResult` + `setMaxResults`).
- En DTOs, preferir **constructor expressions**.
- Evitar `select *` equivocado: en JPQL es `select e` (entidad completa).
- Tras **bulk update/delete**, hacer `em.clear()`.
- Validar `getSingleResult()` con cuidado; mejor `getResultList()` y controlar tamaño.

Anexo - Implementación de capa de acceso a datos con patrón Repositorio

Queda pendiente por ahora

Enlaces relacionados

- Introducción a JPA
- Java Persistence API (JPA)