UD 2. Acceso a BD remotas relacionales. Creación de una interfaz web sencilla (Vaadin)

Alicia Martínez

Contenido

[02.01. Sistemas gestores de bases de datos 3](#_Toc187393819)

[1. SQLite 3](#_Toc187393820)

[2. DAO (Data Access Object) 6](#_Toc187393821)

[3. Creación de una base de datos en H2 6](#_Toc187393822)

<https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/02db/0101databases/#3-ejemplo>

## 02.01. Sistemas gestores de bases de datos

* [1. SQLite](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/02db/0101databases/#1-sqlite)
  + [1.1. Dependencias Maven](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/02db/0101databases/#11--dependencias-maven)
  + [1.2. Bases de datos en memoria](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/02db/0101databases/#12-bases-de-datos-en-memoria)
  + [1.3. Bases de datos en archivo](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/02db/0101databases/#13-bases-de-datos-en-archivo)
  + [1.4. Ayuda y referencias](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/02db/0101databases/#14--ayuda-y-referencias)
* [2. DAO (Data Access Object)](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/02db/0101databases/#2-dao-data-access-object)
* [3. Creación de una base de datos en H2](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/02db/0101databases/#3-creaci%C3%B3n-de-una-base-de-datos-en-h2)
  + [2.1. Dependencias](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/02db/0101databases/#21-dependencias)
* [3. Ejemplo](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/02db/0101databases/#3-ejemplo)

### 1. SQLite

Para facilitar el trabajo de aplicaciones sencillas, existen muchos SGBD relacionales orientados a archivo (embebidos) opensource como H2, SQLite, HSQL, tinySQL, smallSQL o comerciales:

* **SQLite**: [Sitio Oficial](https://sqlite.org/index.html)
* **HSQLDB**: [Sitio Oficial](https://hsqldb.org/) (HyperSQL database management system)
* **H2Database**: <http://h2database.com/html/main.html>
* **MariaDB**: <https://mariadb.org/>
* **PostgreSQL**: <https://www.postgresql.org/>
* **Derby**: <https://db.apache.org/derby/>
* **tinySQL**: [Enlace](http://priede.bf.lu.lv/ftp/pub/DatuBazes/tinySQL/tinySQL.htm)
* **SmallSQL**: [Enlace](http://www.smallsql.de/)
* **Microsoft SQL Server**
* **Oracle**

Uno de los SGBD más empleados, sobre todo en dispositivos móviles, es [SQLite](https://sqlite.org/index.html).

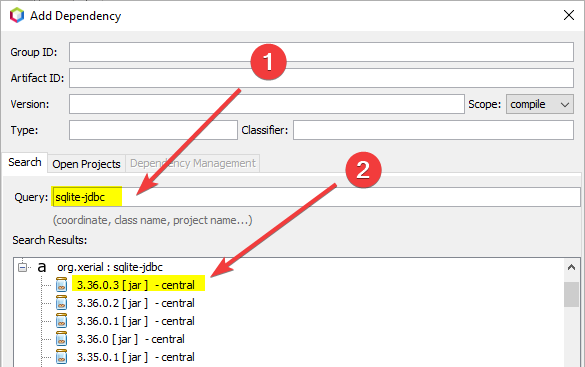
Como trabajaremos con dependencias a los Drivers JDBC, cuyo archivo jar precisamos en nuestro proyecto y en el classpath de ejecución/compilación, recomendaría realizar un proyecto Maven, aunque podría descargarse el driver JDBC de SQLite y añadirse como biblioteca al proyecto Java.

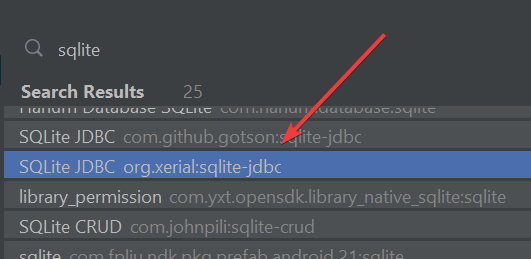
Una de las mejores páginas para consultar información sobre SGBD es:

* [DB-Engines](https://db-engines.com/)
* [Ranking de SGBD](https://db-engines.com/en/ranking): <https://db-engines.com/en/ranking>
* [Comparativa de Derby vs. H2 vs. SQLite](https://db-engines.com/en/system/Derby%3bH2%3bSQLite)

#### 1.1. Dependencias Maven

Para trabajar con SQLite se precisa tener añadida la dependencia con los Driver JDBC de SQLite, por ejemplo, en Netbeans:





Puede hacerse a mano en el propio archivo pom.xml:

<dependencies>

<dependency>

<groupId>org.xerial</groupId>

<artifactId>sqlite-jdbc</artifactId>

<version>3.43.2.2</version>

</dependency>

</dependencies>

A día de hoy la última versión es la 3.43.2.2

Los drives JDBC del SQLite pueden descargarse de:

* [Repositorio Maven](https://mvnrepository.com/artifact/org.xerial/sqlite-jdbc)
* [GitHub Releases](https://github.com/xerial/sqlite-jdbc/releases)

Puedes ver un ejemplo de uso en la [página oficial](https://github.com/xerial/sqlite-jdbc#usage).

Se puede trabajar tanto con datos en memoria (durante la ejecución del programa) como en archivo:

#### 1.2. Bases de datos en memoria

Bases de datos en memoria:

Connection conex = DriverManager.getConnection("jdbc:sqlite::memory:");

#### 1.3. Bases de datos en archivo

En archivo:

Connection conex = DriverManager.getConnection("jdbc:sqlite:rutaArchivo.sqlite3"

El hecho de realizarlo en SQLite da **portabilidad al proyecto**, pues este SGBDR es orientado a archivo y **no precisa estar instalado como servicio en ningún computado**r.

Como dice en la página del proyecto:

“SQLite es una biblioteca ’en proceso’ que implanta un motor de bases de datos SQL autónomo, sin servidor, sin configuración y transaccional…”

“SQLite es el motor de base de datos más utilizado del mundo. SQLite está integrado en todos los teléfonos móviles y la mayoría de las computadoras y viene incluido dentro de innumerables otras aplicaciones que la gente usa todos los días.”

#### 1.4. Ayuda y referencias

* [Documentación de Java tabla con Swing](https://docs.oracle.com/javase/tutorial/uiswing/components/table.html)
* [Sitio Oficial de SQLite](https://sqlite.org/index.html)
* [SQLite Browser](https://sqlitebrowser.org/dl/)

### 2. DAO (Data Access Object)

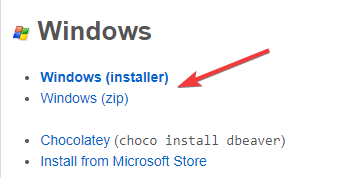
* [Acódigo Blog - DAO con JDBC](http://acodigo.blogspot.com/2016/03/data-access-object-dao-con-jdbc.html)
* [Oracle - Data Access Object](https://www.oracle.com/java/technologies/dataaccessobject.html)
* [YouTube - Tutorial DAO](https://www.youtube.com/watch?v=CEDKxPCgosY)
* [YouTube - JDBC DAO Tutorial](https://www.youtube.com/watch?v=NjY-WA-jeJ8)

### 3. Creación de una base de datos en H2

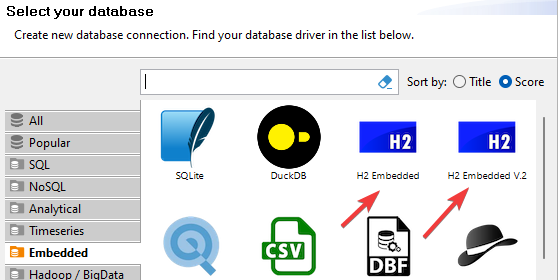
Para la creación de la BD podemos emplear los propios IDE, HeidiSQL, DBeaver (recomendación personal en este caso) o similar. En este caso tan concreto, recomendaría usar DBeaver, pues permite gestionar muchas bases de datos (prácticamente todas las que disponen de Drivers JDBC, pues este programa está escrito en Java):

* [DBeaver](https://dbeaver.io/)
* [Descargar DBeaver](https://dbeaver.io/download/)

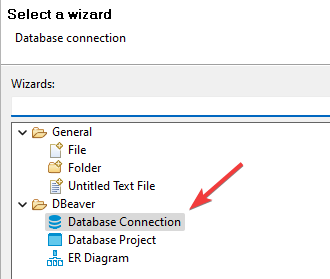
Descarga para Windows:



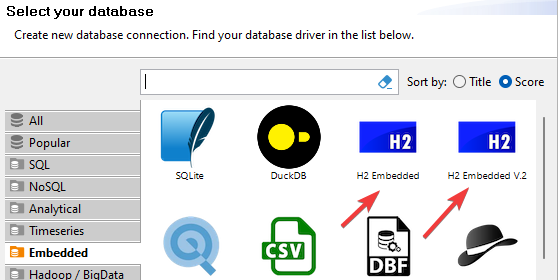
Es importante **crear la base de datos y usar la biblioteca de la aplicación con el mismo formato 1.X o 2.X (no son compatibles)**:



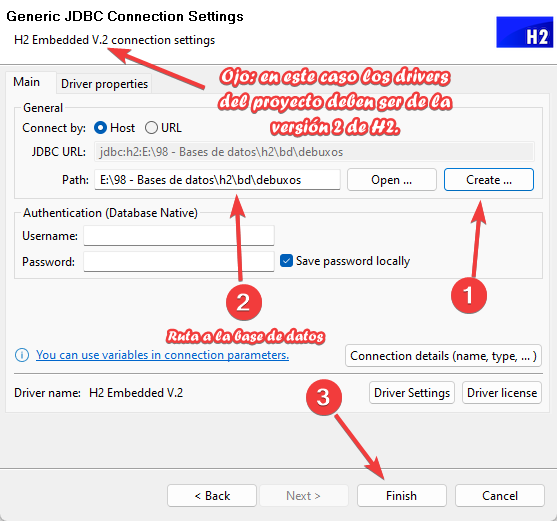
Creamos una nueva conexión:



Seleccionamos el base de datos H2 1.X o la versión 2.X, en cuyo caso debemos añadir las dependencias del proyecto con la versión correspondiente:



Configuración:



#### 3.1. Dependencias

Para la versión 2:

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

<version>2.2.224</version>

</dependency>

3. Ejemplo

Por ejemplo, puedes crear la base de datos con de acuerdo con el script SQL o por medio de la interface gráfica (no entraré en detalles):

CREATE TABLE PUBLIC.Debuxo (

idDebuxo INTEGER NOT NULL AUTO\_INCREMENT,

nome CHARACTER VARYING(64) NOT NULL,

CONSTRAINT DEBUXO\_PK PRIMARY KEY (idDebuxo)

);

CREATE INDEX DEBUXO\_NOME\_IDX ON PUBLIC.DEBUXO (nome);

COMMENT ON TABLE PUBLIC.DEBUXO IS 'Debuxo da base de datos composto por figuras.';

COMMENT ON COLUMN PUBLIC.DEBUXO. idDebuxo IS 'Clave primaria';

COMMENT ON COLUMN PUBLIC.DEBUXO.nome IS 'Nome do debuxo';

CREATE TABLE PUBLIC.Shape (

idDebuxo INTEGER NOT NULL,

shape BINARY LARGE OBJECT,

CONSTRAINT SHAPE\_FK FOREIGN KEY (idDebuxo) REFERENCES PUBLIC.Debuxo(idDebuxo) ON DELETE CASCADE ON UPDATE CASCADE

);

CREATE INDEX SHAPE\_IDDEBUXO\_IDX ON PUBLIC.SHAPE (idDebuxo);

COMMENT ON TABLE PUBLIC.shape IS 'Figuras de dibujo';

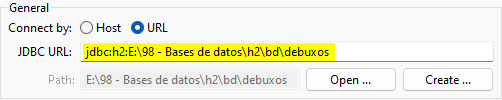
COMMENT ON COLUMN PUBLIC.Shape. idDebuxo IS 'Referencia ó debuxo';

COMMENT ON COLUMN PUBLIC.Shape.shape IS 'BLOB con el objeto de la figura';

En el ejemplo de BD se emplea un tipo dato BLOB (binario grande), para guardar un objeto binario en la base de datos.

La configuración de la URL a la base de datos es la que aparece en las propiedades de la conexión:

"jdbc:h2:RutaABaseDatos\debuxos"



Los parámetros de la conexión deben ser:

JDBC\_DRIVER = "org.h2.Driver"; // No se precisa en JDBC versión mayor a 4.0

DB\_URL = "jdbc:h2:RutaABaseDatos\nomeBD”;

Ahora podemos proceder como cualquier otro proyecto de conexión a base de datos empleando los dos parámetros (el primero no se precisa desde JDBC 4.0).

## 02. Introducción a las bases de datos relacionales

* [1. Introducción a Bases de Datos Relacionales y SQL](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/02db/0102introdb/#1-introducci%C3%B3n-a-bases-de-datos-relacionales-y-sql)
  + [1.1. Derby](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/02db/0102introdb/#11-derby)
* [2. Ejemplo de una base de datos relacional](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/02db/0102introdb/#2-ejemplo-de-una-base-de-datos-relacional)
  + [2.1 Código para configurar la base de datos](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/02db/0102introdb/#21-c%C3%B3digo-para-configurar-la-base-de-datos)
* [3. Repaso de declaraciones SQL básicas](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/02db/0102introdb/#3-repaso-de-declaraciones-sql-b%C3%A1sicas)

### 1. Introducción a Bases de Datos Relacionales y SQL

**Datos** son información. Un dato es un hecho, como tu primer nombre. Una **base de datos** es una colección organizada de datos. En el mundo real, un archivador es un tipo de base de datos. Tiene carpetas, cada una con documentos. Las carpetas se organizan de alguna manera, a menudo alfabéticamente. Cada documento es como un dato. De manera similar, las carpetas en tu computadora son como una base de datos. Las carpetas proporcionan organización y cada archivo es un dato.

Una **base de datos relacional** está organizada en tablas, que constan de filas y columnas.

Hay dos formas principales de acceder a una base de datos relacional desde Java.

1. **Java Database Connectivity Language (JDBC)**: Accede a los datos como filas y columnas. JDBC es la API cubierta en este capítulo.
2. **Java Persistence API (JPA)**: Accede a los datos a través de objetos Java mediante un concepto llamado object-relational mapping (ORM). La idea es que no tienes que escribir tanto código y obtienes tus datos en objetos Java. LO veremos en la unidad de Hibernate, un framework para trabajar con JPA.

Una base de datos relacional se accede mediante **Structured Query Language (SQL)**, un lenguaje de programación utilizado para interactuar con registros de bases de datos. JDBC funciona enviando un comando SQL a la base de datos y luego procesando la respuesta.

Además de las bases de datos relacionales, existe otro tipo de base de datos llamada base de datos **NoSQL**. Esto es para bases de datos que almacenan sus datos en un formato diferente a las tablas, como almacenes de claves/valores, almacenes de documentos y bases de datos basadas en gráficos. NoSQL lo veremos en unidades siguientes.

En los siguientes apartados, veremos una pequeña base de datos relacional que usaremos en algún ejemplo y veremos las sentencias SQL para acceder a ella.

#### 1.1. Derby

En esta introducción veremos **Derby** (<http://db.apache.org/derby>) pero en el aula, para facilitar el trabajo emplearemos SQLite o H2, también haremos ejemplos con MadiaDB.

**Derby es una base de datos pequeña en memoria**. De hecho, sólo se necesita un archivo JAR para ejecutarlo.  
La descarga e implantación es muy sencilla, pero os mostraré cómo hacerlo (podré un enlace al final)

También hay bases de datos “independientes”, orientadas a servicio, que podéis probar par instanalr un motor completo de base de datos. Los más populares (y recomendables) son: **MySQL** ([www.mysql.com](https://www.mysql.com/)), **MariaDB** (<https://mariadb.org/>) o **PostgreSQL** (<https://www.postgresql.org/>),de código abierto y con más de 20 años de existencia.  
Aunque las principales bases de datos tienen muchas similitudes, también tienen diferencias importantes y características avanzadas. Elegir la base de datos correcta para tu trabajo es una decisión importante que debes investigar mucho. Cualquier Sistema Gestor de Bases de Datos es bueno para practicar.

[Diferencias entre MariaDB y MySQL](https://aws.amazon.com/es/compare/the-difference-between-mariadb-vs-mysql/)

Hay muchos manuales para instalar y comenzar con cualquiera de estos Sistemas Gestores de Bases de Datos (SGBD o BDMS). Está fuera de los contenidos de esta materia configurar una base de datos, pero es algo que se precisa conocer si deseamos implantar nuestras aplicaciones de acceso a datos.

### 2. Ejemplo de una base de datos relacional

A modo de ejemplo, emplearemos una base de datos con dos tablas. Una tiene una fila por cada especie del zoológico. La otra tiene una fila por cada animal. Estas dos están relacionadas porque un animal pertenece a una especie. Estas relaciones son por las que este tipo de base de datos se llama base de datos relacional.

**Tablas en nuestra base de datos relacional:**

**Especie:**

| **idEspecie (PK)** | **nome varchar(255)** | **area (decimal)** |
| --- | --- | --- |
| 1 | Elefante Africano | 9.5 |
| 2 | Cebra | 3.1 |

**Animal:**

| **idAnimal (PK)** | **idEspecie (FK)** | **nome, varchar (255)** | **dataNacemento (timestamp)** |
| --- | --- | --- | --- |
| 1 | 1 | Pepa | 2001-05-06 02:15:00 |
| 2 | 2 | Lola | 2012-08-15 09:12:00 |
| 3 | 1 | Dumbo | 2022-09-09 10:36:00 |
| 4 | 1 | Babar | 2010-06-08 01:24:00 |
| 5 | 2 | Rayas | 2013-06-08 01:24:00 |

Hemos declarado dos tablas (después lo ajustaremos un poco), una se llama Especie y la otra Animal. Cada tabla tiene una clave primaria (PK), que nos da una forma única de referenciar cada fila. Después de todo, dos animales pueden tener el mismo nombre, pero no pueden tener la misma ID.

Nota: En el ejemplo, la clave primaria es sólo una columna. En algunas situaciones, es una combinación de columnas llamada clave compuesta. Por ejemplo, un identificador de estudiante y año podrían ser una clave compuesta. Hay dos filas y tres columnas en la tabla *Especie* y cinco filas y tres columnas en la tabla *Animal*.

#### 2.1 Código para configurar la base de datos

En el código SQL se utilizan partes de SQL llamadas **lenguaje de definición de base de datos (DDL) y lenguaje de manipulación de datos (DML)**.

Antes de ejecutar el código, debes agregar un archivo .jar al classpath o añadir la dependencia Maven. Agrega <PATH TO DERBY>/derby.jar al classpath. Asegúrate de reemplazar <PATH TO DERBY> con la ruta real en tu sistema de archivos.

Código de creación:

import java.sql.\*;

public class SetupDerbyDatabase {

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

String url = "jdbc:derby:zoo;create=true";

try (Connection conexion = DriverManager.getConnection(url)) {

// Para eliminar las tablas:

// executar(conexion,"DROP TABLE Animal");

// executar(conexion,"DROP TABLE Especie");

executar(conexion,"CREATE TABLE Especie ("

+ "idEspecie INTEGER PRIMARY KEY, "

+ "nome VARCHAR(255), "

+ "area DECIMAL(4,1))");

executar(conexion,"CREATE TABLE Animal ("

+ "idAnimal INTEGER PRIMARY KEY, "

+ "idEspecie integer REFERENCES Especie (idEspecie), "

+ "nome VARCHAR(255),"

+ "dataNacemento TIMESTAMP)");

executar(conexion,"INSERT INTO Especie VALUES (1, 'Elefante africano', 9.5)");

executar(conexion,"INSERT INTO Especie VALUES (2, 'Cebra', 3.1)");

executar(conexion,"INSERT INTO Animal VALUES (1, 1, 'Pepa', '1972−05−06 02:15')");

executar(conexion,"INSERT INTO Animal VALUES (2, 2, 'Lola', '2009−08−15 09:12')");

executar(conexion,"INSERT INTO Animal VALUES (3, 1, 'Dumbo', '2012−09−09 10:36')");

executar(conexion,"INSERT INTO Animal VALUES (4, 1, 'Babar', '2010−06−08 01:24')");

executar(conexion,"INSERT INTO Animal VALUES (5, 2, 'Rayas', '2015−11−12 03:44')");

printCount(conexion,"SELECT count(\*) FROM Animal");

}

}

private static void executar(Connection conexion, String sql) throws SQLException {

try (PreparedStatement ps = conexion.prepareStatement(sql)) {

ps.executeUpdate();

}

}

private static void printCount(Connection conexion, String sql)

throws SQLException {

try (PreparedStatement ps = conexion.prepareStatement(sql)) {

ResultSet rs = ps.executeQuery();

rs.next();

System.out.println(rs.getInt(1));

}

}

}

Se ejecuta desde línea de órdenes con la siguiente orden:

java -cp "<path\_to\_derby>/derby.jar" SetupDerbyDatabase.java

Lo más correcto es **introducir la biblioteca de Derby en el directorio db/lib** de Java y tener configurado de manera adecuada la variable JAVA\_HOME: <JAVA\_HOME>/db/lib/derby.jar

java −cp "/mi/home/jdk/db/lib/derby.jar:." SetupDerbyDatabase

En Windows:

java −cp "c:\program files\jdk\db\lib\derby.jar;." SetupDerbyDatabase

El programa se conecta a la base de datos y crea dos tablas. Luego carga datos en esas tablas.

De momento sólo es un ejemplo sencillo, veremos cómo una Connection y un PreparedStatement de varios modos.

### 3. Repaso de declaraciones SQL básicas

Hay cuatro tipos de operaciones para trabajar con los datos en las bases de datos. Se conocen como **CRUD (Crear, Leer, Actualizar, Eliminar)**. Las palabras clave de SQL no coinciden con el acrónimo:

**Tabla 21.1 Operaciones CRUD:**

| **Operación** | **Palabra Clave SQL** | **Descripción** |
| --- | --- | --- |
| Crear | INSERT | Agrega una nueva fila a la tabla |
| Leer | SELECT | Recupera datos de la tabla |
| Actualizar | UPDATE | Cambia cero o más filas en la tabla |
| Eliminar | DELETE | Elimina cero o más filas de la tabla |

Si ya conoces SQL, puedes saltar el resto de este apartado. Se muestra lo básico para aquellos que no saben o por si quieres repasar algún concepto básico.

A diferencia de Java, las palabras clave de **SQL no distinguen entre mayúsculas y minúsculas**. Esto significa que select, SELECT y Select son equivalentes. Muchas personas usan mayúsculas para las palabras clave de la base de datos para que destaquen. También es una práctica común usar snake case (guión bajo para separar “palabras”) en los nombres de las columnas. Tened en cuenta que **en algunas bases de datos, los nombres de tabla y columna pueden ser sensibles a mayúsculas y minúsculas**.

Al igual que los tipos primitivos de Java, SQL tiene varios tipos de datos. La mayoría son autoexplicativos, como INTEGER. También hay DECIMAL, que funciona de manera similar a un double en Java. El más extraño es VARCHAR, que significa “variable character” y es similar a un String en Java. La parte variable significa que la base de datos debe usar sólo el espacio necesario para almacenar el valor.

La declaración INSERT se utiliza generalmente para crear una nueva fila en una tabla; aquí tienes un ejemplo:

INSERT INTO Especie

VALUES (3, 'Elefante Africano', 10.8);

La declaración INSERT enumera los valores que queremos insertar. **De forma predeterminada, utiliza el mismo orden en el que se definieron las columnas**. Los datos de cadena están encerrados entre comillas simples.

La declaración SELECT lee datos de la tabla.

SELECT \*

FROM Especie

WHERE idEspecie = 3;

La cláusula WHERE es opcional. Si la omites, se devuelven los contenidos de toda la tabla. El \* indica que se devuelvan todas las columnas en el orden en que se definieron. Alternativamente, puedes enumerar las columnas que deseas que se devuelvan.

SELECT nome, area

FROM Especie

WHERE idEspecie = 3;

Es **preferible enumerar los nombres de las columnas para mayor claridad**. También ayuda en caso de que la tabla cambie en la base de datos.

También puedes obtener información sobre todo el resultado sin devolver filas individuales mediante funciones SQL especiales.

SELECT COUNT(\*), SUM(area)

FROM Especie;

Esta consulta nos dice cuántas especies tenemos y cuánto espacio necesitamos para ellas. Devuelve sólo una fila ya que está combinando información (funciones de agregación), **Si no hay filas en la tabla, la consulta devuelve una fila que contiene cero como respuesta**.

La declaración UPDATE cambia los valores cero o más filas en la base de datos.

UPDATE Especie

SET area = area + .5

WHERE nome = 'Elefante Asiático';

Nuevamente, la cláusula WHERE es opcional. Si se omite, se actualizarán todas las filas de la tabla. La declaración UPDATE siempre especifica la tabla a actualizar y la columna a actualizar.

La declaración DELETE elimina una o más filas de la base de datos.

DELETE FROM Especie

WHERE nome = 'Elefante Asiático';

Y una vez más, la cláusula WHERE es opcional. Si se omite, se vaciará toda la tabla. ¡Así que ten cuidado! ;-)

Todo el SQL mostrado en esta sección es común en casi todos los SGBDR, mas **SQL más avanzado, hay variación entre bases de datos, con funciones o tipos de datos diferentes**.

03. PostgreSQL

* [Instalación de PostgreSQL sin privilegios de administrador](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/02db/0103postgresql/#instalaci%C3%B3n-de-postgresql-sin-privilegios-de-administrador)
  + [1. Instalación en Windows](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/02db/0103postgresql/#1-instalaci%C3%B3n-en-windows)
    - [1.1. Verificación de la instalación](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/02db/0103postgresql/#11-verificaci%C3%B3n-de-la-instalaci%C3%B3n)
    - [1.2. Creación de una base de datos y asociar un usuario en PostgreSQL](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/02db/0103postgresql/#12-creaci%C3%B3n-de-una-base-de-datos-y-asociar-un-usuario-en-postgresql)
    - [1.3. Iniciar el sistema gestor de base de datos](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/02db/0103postgresql/#13-iniciar-el-sistema-gestor-de-base-de-datos)
    - [1.4. pgAdmin4](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/02db/0103postgresql/#14-pgadmin4)
  + [2. Instalación en Ubuntu sin sudo](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/02db/0103postgresql/#2-instalaci%C3%B3n-en-ubuntu-sin-sudo)
    - [2.1. Compilación e Instalación de PostgreSQL](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/02db/0103postgresql/#21-compilaci%C3%B3n-e-instalaci%C3%B3n-de-postgresql)
    - [2.2. Configuración de PostgreSQL](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/02db/0103postgresql/#22-configuraci%C3%B3n-de-postgresql)
  + [3. Instalación en Ubuntu con Docker (sin sudo)](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/02db/0103postgresql/#3-instalaci%C3%B3n-en-ubuntu-con-docker-sin-sudo)
  + [4. Instalación en Windows con Docker](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/02db/0103postgresql/#4-instalaci%C3%B3n-en-windows-con-docker)

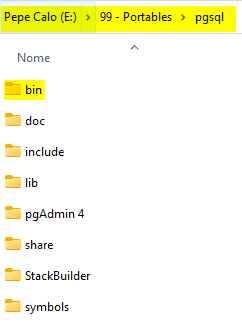
Instalación de PostgreSQL sin privilegios de administrador

Muchas veces (instituto u otro centro educativo, organización,…) es muy poco probable que se tengan privilegios de administrador para instalar cualquier software ajeno, por lo que veremos cómo instalar PostgreSQL sin privilegios de administrador en Windows y Linux.

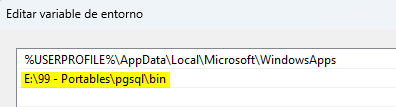
1. Instalación en Windows

Sigue los siguientes pasos para instalar PostgreSQL:

1. Para configurar PostgreSQL, necesitamos [**descargar los binarios de PostgreSQL**](https://www.enterprisedb.com/download-postgresql-binaries): <https://www.enterprisedb.com/download-postgresql-binaries>. Elige el archivo binario de la última versión de PostgreSQL.
2. Crea una **nueva carpeta en un directorio con control total** y extrae estos archivos zip binarios en ella. (Preferiblemente, puedes extraer estos archivos binarios en ubicaciones como la unidad D:\ o E:). La estructura de la carpeta será algo similar a la siguiente:



1. Agrega esta ubicación del **directorio “bin” en la variable PATH en las Variables de entorno de usuario**.



Ya está instalado en el Sistema Operativo. Ahora, necesitamos configurar la base de datos.

Ya está instalado en el Sistema Operativo. Ahora, necesitamos configurar la base de datos.

1.1. Verificación de la instalación

Para verificar si está instalado correctamente, usa los siguientes comandos.

El siguiente comando verifica la **versión del servidor PostgreSQL**:

postgres -V

Obtenemos una salida similar a la siguiente:

postgres (PostgreSQL) 16.2

El siguiente comando verifica la **versión del cliente PostgreSQL**:

psql -V

Obtenemos una salida similar a la siguiente:

psql (PostgreSQL) 16.2

1.2. Creación de una base de datos y asociar un usuario en PostgreSQL

Para **crear una base de datos y asociar un usuario a ésta**, la base de datos se inicializará en la ubicación que hemos especificado (**carpeta de data** en este caso, opción recomendable). La orden es la siguiente:

initdb -D E:\99 - Portables\pgsql\data -U postgres -E utf8

Alguna de las opciones que podemos usar son:

* **-D path/to/db/server/:** informa a initdb para **inicializar la base de datos( cluster de base de datos) en una ubicación concreta** especificada por el usuario. Después de especificar la ubicación, **se creará implícitamente un nuevo directorio y se guardarán aquí todos los archivos de PostgreSQL y sus datos relacionados**.
* **-U NAME** o **–username=NAME**: **crea un usuario con el nombre especificado y con todos los privilegios de superusuario**.
* **-W:** **se utiliza para solicitar explícitamente una contraseña para el nuevo superusuario**.
* **-E:** indica la **codificación que se utilizará para la base de datos**.
* **-A METODO** o **–auth=MÉTODO**: se usa para **especificar el cifrado de la contraseña** para conexiones locales.
* **–auth-local=METODO** se usa para **especificar el cifrado de la contraseña** para conexiones locales por socket.
* **–auth-host=METODO** se usa para **especificar el cifrado de la contraseña** para conexiones de red.
* **–locale=LOCALE:** se usa para **especificar la configuración regional**.

La salida será similar a la siguiente:

E:\99 - Portables\pgsql>initdb -D .\data -U postgres -E utf8

The files belonging to this database system will be owned by user "pepecalo".

This user must also own the server process.

The database cluster will be initialized with locale "Galician\_Spain.1252".

initdb: could not find suitable text search configuration for locale "Galician\_Spain.1252"

The default text search configuration will be set to "simple".

Data page checksums are disabled.

creating directory data ... ok

creating subdirectories ... ok

selecting dynamic shared memory implementation ... windows

selecting default max\_connections ... 100

selecting default shared\_buffers ... 128MB

selecting default time zone ... Europe/Paris

creating configuration files ... ok

running bootstrap script ... ok

performing post-bootstrap initialization ... ok

syncing data to disk ... ok

initdb: warning: enabling "trust" authentication for local connections

initdb: hint: You can change this by editing pg\_hba.conf or using the option -A, or --auth-local and --auth-host, the next time you run initdb.

Success. You can now start the database server using:

pg\_ctl -D ^"^.^\data^" -l logfile start

1.3. Iniciar el sistema gestor de base de datos

1. Inicia el sistema gestor de base de datos ejecutando:

pg\_ctl -D E:\99 - Portables\pgsql\data -l logfile start

La salida será similar a la siguiente:

E:\99 - Portables\pgsql>pg\_ctl -D .\data -l logfile start

waiting for server to start.... done

server started

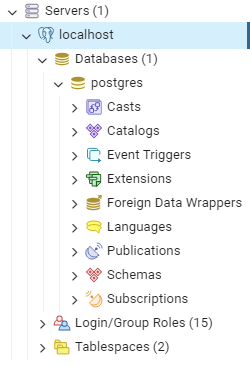
El **puerto por defecto de PostgreSQL es el** **5432**. Si deseas cambiar el puerto, puedes hacerlo en el archivo postgresql.conf en la carpeta data que hemos creado.

1.4. pgAdmin4

Aunque trabajaremos con Dbeaver, una vez que el servidor PostgreSQL esté en funcionamiento, puedes usar pgAdmin4 para administrar la base de datos. Para abrir pgAdmin4, sigue los siguientes pasos:

1. Descarga [**pgAdmin4**](https://www.pgadmin.org/download/): <https://www.pgadmin.org/download/>.

Esta es una herramienta de administración de bases de datos para PostgreSQL y derivados, que **puede ser instalada para un usuario concreto**.

1. Instala pgAdmin4 en la carpeta de tu elección. Por ejemplo, en la carpeta E:\99 - Portables\pgsql\pgAdmin4.
2. Haz doble **clic en la aplicación pgAdmin4** (ruta E:\99 - Portables\pgAdmin 4\runtime) para iniciar el programa (antes era versión Web pero ahora es versión de escritorio).
3. Ahora haz clic en Servers en el lado derecho para crear un nuevo servidor para tu base de datos. Completa los detalles requeridos. Crea una **conexión con el servidor local en el puerto 5432 y con el usuario postgres**:
4. Haz clic en la sección de **Databases para crear una nueva base de datos** para tu trabajo y comenzar a usarla.
5. Para **detener la base de datos**, utiliza el mismo comando utilizado para iniciar la base de datos como se usó arrancarla y sustituye start por stop.

pg\_ctl -D E:\99 - Portables\pgsql\data -l logfile stop

Para usar funciones adicionales de PGSQL como el vaciado (vacuuming), upgrade, restore, etc., es posible que necesites configurar las rutas binarias para ello. Para esto, ve a **File -> Preferences**. Ahora desplázate hacia abajo hasta Paths y **haz clic en Binary Paths**. Especifica las rutas como el directorio de la carpeta bin de instalación de PgSQL:

* EDB Advanced Server Binary Path: E:\99 - Portables\pgsql\bin
* PostgreSQL Binary Path: E:\99 - Portables\pgsql\bin

2. Instalación en Ubuntu sin sudo

[Referencia](https://www.postgresql.org/download/linux/ubuntu/) <https://gist.github.com/usuario/832aceac6998e2f894e5780229920cb5>

En este caso seguiremos la opción de compilación desde el código fuente.

2.1. Compilación e Instalación de PostgreSQL

Asegúrate de tener instalado el siguiente software:

* Un compilador de C/C++, como GCC.
* Otro software de soporte, [consulta aquí](https://www.postgresql.org/docs/current/install-requirements.html).

1. Descarga el código fuente: <https://www.postgresql.org/ftp/source/>.

Puedes usar wget https://www.postgresql.org/ftp/source/v16.2/postgresql-16.2.tar.gz.

* + **Nota:** La versión puede cambiar, por lo que asegúrate de descargar la última versión.

1. Descomprime el código fuente con tar xvzf postgresql-16.2.tar.gz.

tar -xzvf postgresql-[version].tar.gz

1. Navega a la carpeta con cd postgresql-16.2/.

Nota: a veces es necesario tener instaladas las dependencias de compilación de PostgreSQL. Puedes instalarlas con sudo apt-get install build-essential ;-)

sudo apt-get update

sudo apt-get install build-essential libreadline-dev zlib1g-dev flex bison

1. Actualiza la configuración del código fuente y compila:

cd postgresql-[version]

./configure --prefix=$HOME/postgresql

make

make install

 Opcional. Número de núcleos

1. Puedes compilar el código fuente con make world -j <num\_cores\_to\_use>.
   * Verifica el número de núcleos de CPU en tu máquina con cat /proc/cpuinfo | grep processor | wc -l.
   * El argumento world significa que también se compilarán todos los módulos adicionales. Puedes ignorarlo.
   * Este paso puede tardar un tiempo si el computador no es lo suficientemente potente.
2. Compila PostgreSQL con make install-world.

2.2. Configuración de PostgreSQL

1. Añade los binarios de PostgreSQL a la ruta con:

echo 'export PATH="$HOME/postgresql/bin:$PATH"' >> ~/.bash\_profile && source ~/.bash\_profile`.

1. Inicializa la base de datos con initdb -D $HOME/postgresql/data.
2. Inicia la base de datos con pg\_ctl -D $HOME/postgresql/data -l $HOME/postgresql/server.log start.

Ahora deberías tener **PostgreSQL instalado y ejecutándose en tu sistema Ubuntu** sin necesidad de sudo. Puedes **acceder a la línea de comandos de PostgreSQL ejecutando psql y conectar a tu servidor PostgreSQL utilizando el usuario por defecto postgres**. Por ejemplo:

psql -U postgres

Luego podrás comenzar a trabajar con tu base de datos PostgreSQL.

1. Puedes cambiar la contraseña con ALTER USER <username> PASSWORD 'new\_password\_here';.
2. Puedes eliminar la base de datos predeterminada creada con DROP DATABASE postgres;.
3. Puedes crear una base de datos con el mismo nombre que la cuenta del sistema operativo con CREATE DATABASE <username>;.

Puedes **conectarte a PostgreSQL simplemente usando psql**.

3. Instalación en Ubuntu con Docker (sin sudo)

Si no puedes instalar PostgreSQL en tu sistema, puedes usar Docker para ejecutar PostgreSQL en un contenedor.

1. *(Ya instalado en el centro)* Instala Docker siguiendo las instrucciones de la [documentación oficial](https://docs.docker.com/engine/install/ubuntu/).
2. Ejecuta el siguiente comando para descargar la imagen de PostgreSQL:

docker pull postgres

1. Ejecuta el siguiente comando para ejecutar PostgreSQL en un contenedor:

docker run --name nombreContendor -e POSTGRES\_PASSWORD=micontraseña -p 5432:5432 -d postgres

1. Conéctate a PostgreSQL ejecutando el siguiente comando:

docker exec -it nombreContendor psql -U postgres

1. Ahora puedes trabajar con PostgreSQL en tu sistema Ubuntu sin necesidad de instalarlo.

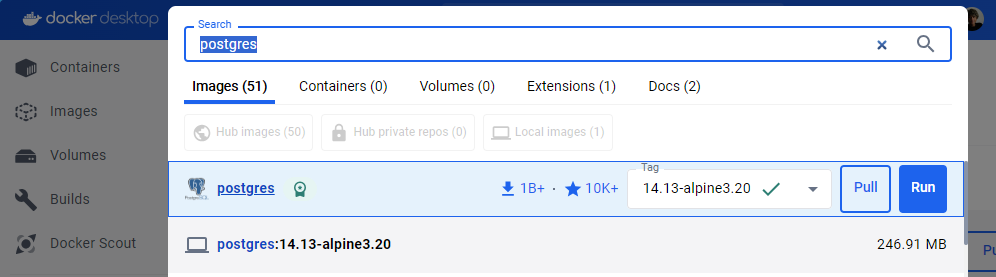
4. Instalación en Windows con Docker

Si no puedes instalar PostgreSQL en tu sistema, puedes usar Docker para ejecutar PostgreSQL en un contenedor.

1. *(Ya instalado en el centro)* Instala Docker siguiendo las instrucciones de la [documentación oficial](https://docs.docker.com/desktop/setup/install/windows-install/).
2. Ejecuta el siguiente comando para descargar la imagen de PostgreSQL:

docker pull postgres

o emplea la interfaz gráfica de Docker: **Docker Desktop**:



Pulsando en Pull descargamos la imagen de PostgreSQL.

[Imagen de Docker Hub](https://hub.docker.com/_/postgres?uuid=d516f26d-9b22-45c5-a9ce-05518be66ba6%0A)

1. Ejecuta el siguiente comando para ejecutar PostgreSQL en un contenedor:

docker run --name nombreContenedor -e POSTGRES\_PASSWORD=contraseña -p 5432:5432 -d postgres

* La opción -p 5432:5432 mapea el puerto 5432 del contenedor al puerto 5432 del host.
* La opción -e POSTGRES\_PASSWORD=contraseña establece la contraseña de la base de datos.
* La opción --name nombreContenedor establece el nombre del contenedor.
* La opción -d ejecuta el contenedor en segundo plano con el modo demonio.

Postgres estará ejecutándose en el contenedor en el puerto 5432.

Si lo haces desde Docker Desktop, puedes hacer clic en Run para ejecutar el contenedor y verlo en la pestaña Containers/Apps. Debes poner la variable de entorno POSTGRES\_PASSWORD con la contraseña que desees.

1. Conéctate a PostgreSQL ejecutando el siguiente comando:

docker exec -it nombreContenedor psql -U postgres

También puedes usar **pgAdmin4** para administrar la base de datos. Para ello, sigue los pasos anteriores para instalar pgAdmin4 en Windows o el Dbveaver.

1. Ahora puedes trabajar con PostgreSQL en tu sistema Windows sin necesidad de instalarlo.

04. PostgreSQL. Características y Operadores.

* [PostgreSQL – Características y Operadores](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/02db/0104postgres01/#postgresql--caracter%C3%ADsticas-y-operadores)
  + [1. Características de PostgreSQL](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/02db/0104postgres01/#1-caracter%C3%ADsticas-de-postgresql)
  + [2. Tipos de Datos en PostgreSQL](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/02db/0104postgres01/#2-tipos-de-datos-en-postgresql)
  + [3. Operadores en PostgreSQL](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/02db/0104postgres01/#3-operadores-en-postgresql)
  + [4. Instalación en Linux](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/02db/0104postgres01/#4-instalaci%C3%B3n-en-linux)
  + [5. Trabajando con Bases de Datos:](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/02db/0104postgres01/#5-trabajando-con-bases-de-datos)
  + [6. Modificación de Tablas](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/02db/0104postgres01/#6-modificaci%C3%B3n-de-tablas)
  + [7. Restauración mediante la línea de comandos](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/02db/0104postgres01/#7-restauraci%C3%B3n-mediante-la-l%C3%ADnea-de-comandos)

PostgreSQL – Características y Operadores

**PostgreSQL** es un **sistema de gestión de bases de datos objeto-relacional potente y de código abierto** que tiene como objetivo ayudar a los desarrolladores a construir aplicaciones y a los administradores a proteger la integridad de los datos y construir entornos tolerantes a fallos. Admite tipos de datos avanzados y características de optimización de rendimiento, como Ms-SQL Server y Oracle.

1. Características de PostgreSQL

* Sistema de gestión de bases de datos de **código abierto**
* Admite **propiedades ACID**
* Técnicas de **indexación diversas**
* **Replicación basada en registros y basada en disparadores SSL**
* Soporte para **JSON**
* Admite **objetos geográficos**
* Compatible **con orientado a objetos y ANSI-SQL 2008**

2. Tipos de Datos en PostgreSQL

| **Numeric** | **Character** | **Date/Time** | **Monetary** | **Binary** |
| --- | --- | --- | --- | --- |
| Boolean | Geométrico | JSON | Enumerado | Búsqueda de texto |
| UUID | Tipos de dirección de red | Compuesto | Identificadores de objeto | Pseudo |
| BitString | XML | Rango | Arrays | pg\_lsn |

* **Datos Numéricos**: smallint, integer, bigint, decimal, numeric, real, serial.
* **Datos de Carácter**: varchar(n), text, char(n).
* **Datos de Fecha/Hora**: timestamp, date, time, interval.
* **Tipo de Datos Monetarios**: money.
* **Tipo de Datos Binarios**: bytea (admite formato hexadecimal y de escape).
* **Tipo de Datos Booleano**: boolean.
* **Tipos de Datos Geométricos**: point, line, box, path, polygon, circle, lseg.
* **Tipos de Datos JSON**: string, number, boolean, null.
* **Tipos de Datos Enumerados**: enum.
* **Tipo de Datos UUID**: uuid (almacena Identificadores Únicos Universales).
* **Tipos de Dirección de Red**: cidr, inet, macaddr.
* **Pseudo Tipos**: any, anyelement, anyarray, anyenum, anyrange, internal, record, trigger, event\_trigger.
* **Tipos de BitString**: bit(n), bit varying(n).
* **Tipos de Datos de Rango**: int4range, int8range, numrange, tsrange (rango de marcas de tiempo), daterange.
* **Tipo de Datos pg\_lsn**: pg\_lsn (almacena Número de Secuencia de Registro).

3. Operadores en PostgreSQL

Un operador manipula elementos de datos individuales y devuelve un resultado. Estas son las palabras reservadas **utilizadas en la cláusula WHERE para realizar operaciones**.

* **Operadores Aritméticos**: +, -, \*, /, %, ^, !
* **Operadores de Comparación**: =, !=, <>, >, <, >=, <=
* **Operadores Lógicos**: AND, NOT, OR
* **Operadores a Nivel de Bits**: &, |

4. Instalación en Linux

a) Para instalar PostgreSQL, ejecuta el siguiente comando:

sudo apt install postgresql

O

sudo apt install postgresql postgresql-contrib

**postgresql-contrib agregará algunas utilidades y funcionalidades adicionales**.

b) Después de la instalación, cambia a la cuenta de Postgres:

sudo -i -u postgres

c) Ahora, puedes **acceder al prompt de Postgres usando el comando psql**.

5. Trabajando con Bases de Datos:

* **CREATE DATABASE**: Se utiliza para crear la base de datos.
* CREATE DATABASE nombre\_base\_de\_datos;
* **CREATE TABLE**: Se utiliza para crear la tabla.
* CREATE TABLE nombre\_tabla
* (columna\_1 tipo\_de\_dato,
* columna\_2 tipo\_de\_dato,
* ...
* columna\_n tipo\_de\_dato);
* **INSERT**: Se utiliza para insertar un nuevo registro (fila) en la tabla.
* INSERT INTO nombre\_tabla (columna\_1, columna\_2 ,...)
* VALUES(valor\_1, valor\_2, ...);
* **SELECT**: Se utiliza para obtener datos de una tabla de la base de datos.
* SELECT
* columna\_1, columna\_2, .. columna\_n
* FROM
* nombre\_tabla;
* La cláusula **WHERE** se utiliza para filtrar registros.
* SELECT
* columna\_1, columna\_2, .. columna\_n
* FROM
* nombre\_tabla
* WHERE
* condición\_1 AND condición\_2;
* **LIMIT**: Se utiliza para limitar el número de registros devueltos por una consulta.
* SELECT
* columna\_1, columna\_2, .. columna\_n
* FROM
* nombre\_tabla
* LIMIT número\_de\_registros;
* **ORDER BY**: Se utiliza para ordenar los registros devueltos por una consulta.
* SELECT
* columna\_1, columna\_2, .. columna\_n
* FROM
* nombre\_tabla
* ORDER BY columna\_1 ASC|DESC, columna\_2 ASC|DESC, ...;
* **OFFSET**: Se utiliza para **omitir un número específico de registros de una consulta**.
* SELECT
* columna\_1, columna\_2, .. columna\_n
* FROM
* nombre\_tabla
* OFFSET número\_de\_registros;

6. Modificación de Tablas

Es posible modificar la estructura de una tabla existente utilizando la instrucción ALTER TABLE. **PostgreSQL admite diversas acciones para realizar con ALTER TABLE**, que se enumeran a continuación:

* **Agregar una columna a una tabla existente**:
* ALTER TABLE nombre\_tabla ADD COLUMN nuevo\_nombre\_columna TIPO;
* **Eliminar una columna de una tabla existente**:
* ALTER TABLE nombre\_tabla DROP COLUMN nombre\_columna;
* **Renombrar una columna de una tabla existente**:
* ALTER TABLE nombre\_tabla RENAME COLUMN nombre\_columna TO nuevo\_nombre\_columna;
* **Cambiar el nombre de una columna de una tabla existente**:
* ALTER TABLE nombre\_tabla ALTER COLUMN nombre\_columna [SET DEFAULT valor | DROP DEFAULT];
* **Cambiar la restricción NOT NULL**:
* ALTER TABLE nombre\_tabla ALTER COLUMN nombre\_columna [SET NOT NULL | DROP NOT NULL];
* **Agregar restricciones CHECK a una columna**:
* ALTER TABLE nombre\_tabla ADD CHECK expresion;
* **Agregar una restricción**:
* ALTER TABLE nombre\_tabla ADD CONSTRAINT nombre\_restriccion definicion\_restriccion;
* **Renombrar una tabla existente**:
* ALTER TABLE nombre\_tabla RENAME TO nuevo\_nombre\_tabla;
* **UPDATE**: Se utiliza para actualizar o modificar datos existentes en la tabla.
* UPDATE nombre\_tabla
* SET columna\_1 = valor\_1,
* columna\_2 = valor\_2, ...
* WHERE
* condición\_1 AND condición\_2;
* **DELETE**: Se utiliza para eliminar fila(s) de la tabla.
* DELETE FROM nombre\_tabla
* WHERE condición;

7. Restauración mediante la línea de comandos

Para restaurar una base de datos mediante la línea de comandos, seguiremos el siguiente procedimiento:

1. En primer lugar, necesitamos **iniciar sesión en el terminal de PostgreSQL a través de la línea de comandos**. Para hacerlo, escriba el siguiente comando:

psql -U <nombre\_usuario>

1. Ahora podemos ver que hemos iniciado sesión correctamente en el terminal cliente psql y **hemos obtenido el indicador de entrada de línea de comandos de PostgreSQL**.
2. Lo haremos esta vez a través del terminal de línea de comandos de PostgreSQL.
3. Ahora **crearemos una base de datos de marcador de posición para nuestro propósito que se utilizará para restaurar la copia de seguridad**. Para hacerlo, ejecute el siguiente script.

CREATE DATABASE BackupDB ENCODING='UTF-8' OWNER='postgres';

1. La base de datos ahora está creada. **Ahora vamos a restaurarla**. Para restaurar la base de datos, **vamos a utilizar el comando pg\_restore suministrado con algunos argumentos**. Es importante tener en cuenta aquí que necesitamos salir del terminal psql para poder ejecutar el comando pg\_restore. **Para salir del terminal psql, escriba “\q”**.
2. Introduce el comando pg\_restore con los siguientes argumentos:

pg\_restore -U postgres -d backupdb -v "D:\Backup.sql"

La explicación detallada de los argumentos para PostgreSQL se puede encontrar en el sitio web oficial de PostgreSQL en la sección de documentación de pg\_restore.

1. Después de la restauración exitosa de la base de datos, veremos que nuestros esquemas se han restaurado junto con las tablas y sus datos.

2.03 Procesamiento de sentencia SQL

En este apartado veremos las interfaces y clases que declara el API JDBC.

01.Procesamiento de sentencias SQL

* [Introducción](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0101procesamientosql/#introducci%C3%B3n)
  + [Características](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0101procesamientosql/#caracter%C3%ADsticas)
  + [Ejemplos de SGBD relacionales](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0101procesamientosql/#ejemplos-de-sgbd-relacionales)
  + [El API JDBC](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0101procesamientosql/#el-api-jdbc)
* [Etapas de procesamiento de sentencias](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0101procesamientosql/#etapas-de-procesamiento-de-sentencias)
  + [1. Establecimiento Connection](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0101procesamientosql/#1-establecimiento-connection)
  + [2. Creación de Statement](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0101procesamientosql/#2-creaci%C3%B3n-de-statement)
  + [3. Ejecución de consultas: execute, executeQuery, executeUpdate](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0101procesamientosql/#3-ejecuci%C3%B3n-de-consultas-execute-executequery-executeupdate)
  + [4. Obtención de objetos ResultSet](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0101procesamientosql/#4-obtenci%C3%B3n-de-objetos-resultset)
  + [5. Cierre de Conexiones](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0101procesamientosql/#5-cierre-de-conexiones)

## Introducción

* La **API Java JDBC (Java Database Connectivity)** permite que las aplicaciones Java se conecten a SGBD relacionales.
* La API JDBC permite **consultar y actualizar, así como procedimientos almacenados u obtener metadatos sobre la base de datos relacionales** (como MySQL, PostgreSQL, MS SQL Server, Oracle, H2 Database, etc.)
* La API Java **JDBC forma parte del SDK**, por lo que está disponible para todas las aplicaciones Java.



* Java proporciona conexión a bases de datos mediante JDBC (Java Database Connection) que proporciona una interface a **muchos tipos de bases de datos**.
* Oculta las diferencias debajo de SQL y **proporciona un conjunto de interfaces** que son una abstracción de la funcionalidad de la base de datos.
* Nos **conectamos desde java con unos controladores (drivers), implementaciones de las interfaces JDBC del API** que pueden haber sido escritos en puro Java, para ser 100% portables, o pueden implicar un componente nativo. Un ejemplo es el puente JDBC-ODBC, que depende del S.O. y sólo se puede ejecutar en Windows.

### Características

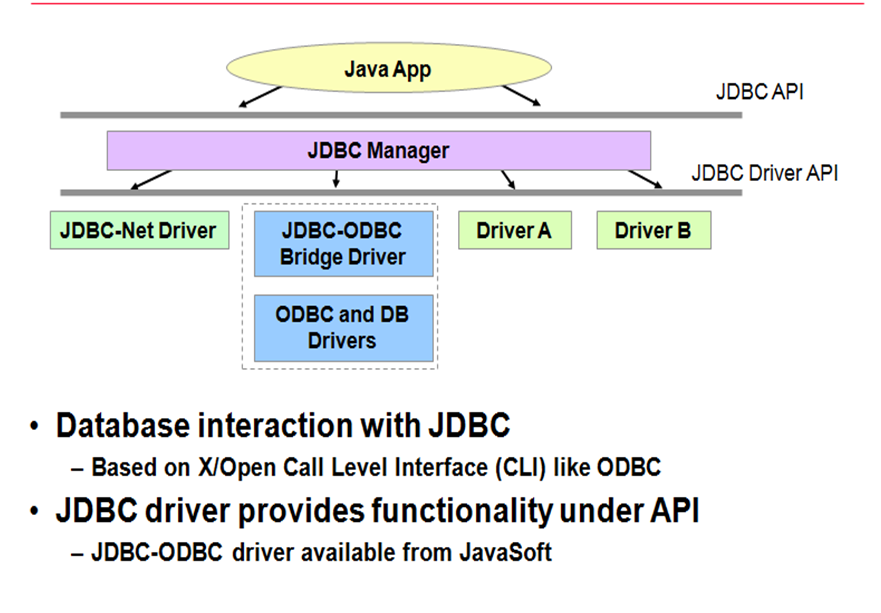
* JDBC es **independiente del Sistema Gestor de BD**.
* JDBC **no es independiente de SQL**: el dialecto de SQL utilizado por diferentes bases de datos varía ligeramente dependiente del SGBD (emplea SQL estándar)
* **NO es para SGBD no relacionales como MongoDB, Cassandra, Dynamo, etc, que tienen su propia biblioteca Java**.

Elementos e interfaces relevantes:

* Driver JDBC.
* Connection.
* Statement.
* PreparedStatement.
* CallableStatement.
* ResultSet.
* Batch Updates.
* Transactions.
* DatabaseMetadata.

### Ejemplos de SGBD relacionales

**SQLite**: <https://sqlite.org/index.html> **HSQLDB**: <https://hsqldb.org/> (HyperSQL database management system) **H2Database** **MariaDB** **PostgreSQL** **Derby** **tinySQL**: <http://priede.bf.lu.lv/ftp/pub/DatuBazes/tinySQL/tinySQL.htm **SmallSQL**: <http://www.smallsql.de/> **Microsoft SQL Server** **Oracle**



import java.sql.\*;

public class EjemploJDBC {

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

Class.forName("org.h2.Driver");

String url = "jdbc:h2:~/prueba"; //URL específica de la base de datos

String usuario = "sa";

String password = "";

try(Connection conexion = DriverManager.getConnection(url, usuario, password)) {

try(Statement st = conexion.createStatement()){

String sql = "select \* from alumno";

try(ResultSet result = st.executeQuery(sql)){ // Cierra automáticamente.

while(result.next()) {

String nome = result.getString(“nome");

long idade = result.getLong(“idade");

}

}

}

} catch (SQLException e) { /\* … \*/ }

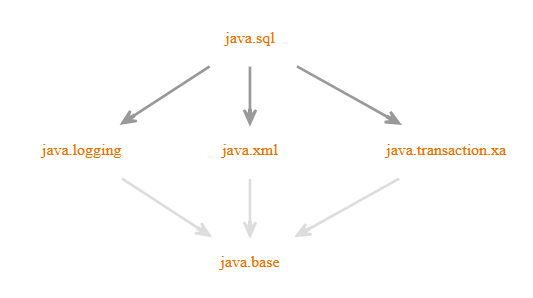
}

}

### El API JDBC

Existe un módulo específico en el API Java SE para trabajar con bases de datos (creado en Java 9):

<https://docs.oracle.com/en/java/javase/21/docs/api/java.sql/module-summary.html>



El paquete principal es [java.sql](https://docs.oracle.com/en/java/javase/21/docs/api/java.sql/java/sql/package-summary.html#versions-heading), pero desde el API JDBC 4.3 **se incluye tanto el**[**java.sql**](https://docs.oracle.com/en/java/javase/21/docs/api/java.sql/java/sql/package-summary.html#versions-heading)**, denominado API principal de JDBC, como el**[**javax.sql**](https://docs.oracle.com/en/java/javase/21/docs/api/java.sql/javax/sql/package-summary.html)**,** denominado API del **paquete opcional JDBC**. Esta API JDBC completa está incluida en Java Standard Edition (Java SE), desde la versión 7.

* [java.sql](https://docs.oracle.com/en/java/javase/21/docs/api/java.sql/java/sql/package-summary.html): API principal para **acceder y procesar datos almacenados en una fuente de datos** (normalmente una base de datos relacional) utilizando el lenguaje de programación Java. Incluye un marco mediante el cual se pueden instalar diferentes controladores de forma dinámica para acceder a diferentes fuentes de datos. Aunque la API JDBC está diseñada principalmente para pasar declaraciones SQL a una base de datos, **permite leer y escribir datos de cualquier fuente de datos con formato tabular**. Existen unas interfaces de lectura/escritura en javax.sql.RowSet, que se puede personalizar para usar y actualizar datos de una hoja de cálculo, un archivo plano o cualquier otra fuente de datos tabulares. Incluye:
  + Clases e interfaces para establecer conexiones a BBD con la clase DriverManager: DriverManager, SQLPermission, Driver, DriverPropertyInfo.
  + Envío de sentencias a la base de datos: Statement, PreparedStatement, CallableStatement, Connection, Savepoint.
  + Obtención y actualización de resultado s de una consulta: ResultSet.
  + Mapeo de tipos SQL a clases e interfaces Java: Array, Blob, Clob, Date, NClob, Ref, RowId, Struct, SQLXML, Time, Timestamp, Types.
  + Mapeo personalizado de tipos SQL definidos por el usuario (UDT) a clases Java: SQLDat, SQLInput, SQLOuput.
  + Metadatos: DatabaseMetaData, ResultSetMetaData, ParameterMetaData.
  + Excepciones: SQLException, SQLWarning, DataTruncation, BatchUpdateException.
* [javax.sql](https://docs.oracle.com/en/java/javase/21/docs/api/java.sql/javax/sql/package-summary.html): API para el acceso y procesamiento de fuentes de datos del lado del servidor desde el lenguaje de programación Java. **Complementa el paquete java.sql y, a partir de la versión 1.4, se incluye en Java Platform, Standard Edition (Java SE)**. Sigue siendo una parte esencial de Java Platform, Enterprise Edition (Java EE).

Los paquetes relacionados:

* [API para transacciones distribuidas](https://docs.oracle.com/en/java/javase/21/docs/api/java.transaction.xa/javax/transaction/xa/package-summary.html): javax.transaction.xa, define el contrato entre el gestor de transaccione y el gestor recursos, lo que permite al administrador de transacciones dar de alta y eliminar objetos de recursos (proporcionados por el controlador del administrador de recursos) en transacciones JTA.
* [java.util.logging](https://docs.oracle.com/en/java/javase/21/docs/api/java.logging/java/util/logging/package-summary.html): proporciona las clases e interfaces de las funciones principales de log de la plataforma Java 2. El objetivo principal del API de logging es respaldar el mantenimiento y servicio del software en los sitios de los clientes.
* [Módulo java.xml](https://docs.oracle.com/en/java/javase/21/docs/api/java.xml/module-summary.html): declara y define el API de Java para **procesamiento XML** (JAXP = Java APIs for XML Processing).

## Etapas de procesamiento de sentencias

En general, para procesar cualquier sentencia SQL con JDBC, sigue estos pasos:

1. Establecer una conexión (Connection)
2. Crea una declaración (Statement)
3. Ejecuta la consulta (executeQuery/executeUpdate/execute)
4. Procesa el objeto ResultSet, en el caso de ser de consulta)
5. Cierra la conexión (de manera automática con try-catch-with-resources)

Por ejemplo, el método, Juego.showTabla el contenido de la tabla Juego:

public static void showTabla(Connection con) throws SQLException {

String query = "select nombre, idDesarrollador, precio, ventas, total from Juego";

try (Statement stmt = con.createStatement()) {

ResultSet rs = stmt.executeQuery(query);

while (rs.next()) {

String nombreJuego = rs.getString("nombre");

int idDesarrollador = rs.getInt("idDesarrollador");

float precio = rs.getFloat("precio");

int ventas = rs.getInt("ventas");

int total = rs.getInt("total");

System.out.println(nombreJuego + ", " + idDesarrollador + ", " + precio +

", " + ventas + ", " + total);

}

} catch (SQLException e) {

// Gestión de la excepción.

}

}

### 1. Establecimiento Connection

Primero, establece una conexión con la fuente de datos que deseas utilizar. Una fuente de datos (Data source) puede ser un sistema de gestión de bases de datos (DBMS), un sistema de archivos heredado u otra **fuente de datos con un controlador JDBC correspondiente**. Esta conexión está representada por un objeto Connection.

### 2. Creación de Statement

Un Statement es una interfaz que representa una sentencia SQL. Si se invocan métodos de consulta (executeQuery) sobre un Statement y generan objetos **ResultSet, una tabla de datos que representa un conjunto de resultados de base de datos**.

Por ejemplo, Juego.showTabla crea un objeto Statement con el siguiente código:

stmt = con.createStatement();

Existen tres tipos diferentes de declaraciones:

* Statement: Se utiliza para implementar sentencias SQL simples **sin parámetros**.
* PreparedStatement: (hereda de Statement.) Se utiliza para compilar previamente **(precompilar) sentencias SQL que pueden contener parámetros de entrada**.
* CallableStatement: (hereda de PreparedStatement.) se utiliza para **ejecutar procedimientos almacenados** que pueden contener **parámetros de entrada y salida**.

### 3. Ejecución de consultas: execute, executeQuery, executeUpdate

Para ejecutar una consulta, llama a un método de tipo execute de Statement. Existen 3 versiones:

* execute: devuelve true si el primer objeto que devuelve la consulta es un **objeto ResultSet**. Se **utiliza este método si la consulta podría devolver uno o más objetos ResultSet**. Después se recuperan los objetos ResultSet devueltos por la consulta llamando repetidamente a Statement.getResultSet.
* executeQuery: **devuelve un objeto ResultSet**.
* executeUpdate: **devuelve un entero** que representa el **número de filas afectadas** por la sentencia SQL, con sentencias SQL **INSERT, DELETE o UPDATE**.

Por ejemplo, Juego.showTabla ejecutó un objeto Statement con el siguiente código:

ResultSet rs = stmt.executeQuery(query);

### 4. Obtención de objetos ResultSet

Para **acceder a los datos de un objeto ResultSet se realiza a través un cursor**, que no es un cursor de la base de datos. Es un **puntero que apunta a una fila de datos en el objeto ResultSet**. Inicialmente, **el cursor se encuentra antes de la primera fila**.

Existe varios métodos definidos en el objeto ResultSet para mover el cursor (next, previous,…)

Por ejemplo, Juego.showTabla llama repetidamente al método ResultSet.next para mover el cursor hacia adelante una fila. Cada vez que llama a next, el método obtiene los datos en la fila donde se encuentra actualmente el cursor:

ResultSet rs = stmt.executeQuery(query);

while (rs.next()) {

String nombreJuego = rs.getString("nombre");

int idDesarrollador = rs.getInt("idDesarrollador");

float precio = rs.getFloat("precio");

int ventas = rs.getInt("ventas");

int total = rs.getInt("total");

System.out.println(nombreJuego + ", " + idDesarrollador + ", " + precio +

", " + ventas + ", " + total);

}

// ...

### 5. Cierre de Conexiones

Siempre que no precisemos más los objetos Connection, Statement o ResultSet, llama a su método close para liberar inmediatamente los recursos que está utilizando.

Es mejor recomendación **emplear una declaración try-with-resources para cerrar automáticamente los objetos Connection, Statement y ResultSet**, independientemente de si ha lanzado una SQLException. (JDBC lanza una SQLException cuando encuentra un error durante una interacción con una fuente de datos.)

Como sabes, una declaración automática de recursos consta de una declaración try y uno o más recursos declarados. Por ejemplo, el método Juego.showTabla cierra automáticamente su objeto Statement, de la siguiente manera:

public static void viewTable(Connection con) throws SQLException {

String query = "select nombre, idDesarrollador, precio, ventas, total from Juego";

try (Statement stmt = con.createStatement()) {

ResultSet rs = stmt.executeQuery(query);

while (rs.next()) {

String nombreJuego = rs.getString("nombre");

int idDesarrollador = rs.getInt("idDesarrollador");

float precio = rs.getFloat("precio");

int ventas = rs.getInt("ventas");

int total = rs.getInt("total");

System.out.println(nombreJuego + ", " + idDesarrollador + ", " + precio +

", " + ventas + ", " + total);

}

} catch (SQLException e) {

JDBCTutorialUtilities.printSQLException(e);

}

}

La siguiente declaración es una declaración try-with-resources, que declara un recurso, stmt, que se cerrará automáticamente cuando el bloque try finalice:

try (Statement stmt = con.createStatement()) {

// ...

}

(El usuario de la base de datos es el usuario por defecto, “sa”, sin comillas, y la contraseña en blanco) Para permitir el uso de nombres en CamelCase en H2 JDBC Driver versión 2, es necesario agregar la propiedad DATABASE\_TO\_UPPER=FALSE en la URL de conexión.

Por ejemplo, si la URL de conexión es jdbc:h2:/test;DATABASE\_TO\_UPPER=FALSE\*. URL: jdbc:h2:ruta\_a\_baseDatos/JuegosH2

 Ejercicio. Crear y transferir datos JSON-BD

Descarga de datos JSON y almacenamiento en una base de datos SQLite. Para ello debes ampliar el ejercicio anterior de JSON.

Al menos debes haber realizado los adaptadores de tipo, las clases del modelo para poder realizar de manera mejor diseñada el apartado i), que hace referencia a BD. Si no está hecho, debes leer el JSON y guardar los datos en la base de datos SQLite creada, con las tablas:

**Plataforma**: idPlataforma, nombre. **Genero**: idGenero, nombre. **Juego**: idJuego, titulo, miniatura (varchar), estado, descripción, url, idGenero (FK), idePlataforma (FK), editor, desarrollador, fecha, urlFreeToGame. **Imagen**: idImagen, idJuego (FK), url, imagen (tipo BLOB). De momento, sólo se guardará la URL a la imagen.

Referencias: <https://www.freetogame.com/api-doc>

Disponemos de un archivo JSON los dos datos de un juego:

* Las plataformas pueden ser: pc, browser, all. (Incorpóralas a la tabla Plataforma)
* Las categorías (géneros) pueden ser:
  + mmorpg, shooter, strategy, moba, racing, sports, social, sandbox, open-world, survival, pvp, pve, pixel, voxel, zombie, turn-based, first-person, third-Person, top-down, tank, space, sailing, side-scroller, superhero, permadeath, card, battle-royale, mmo, mmofps, mmotps, 3d, 2d, anime, fantasy, sci-fi, fighting, action-rpg, action, military, martial-arts, flight, low-spec, tower-defense, horror, mmorts. (Incorpóralas a la tabla Genero)
* La ordenación puede ser: release-date, popularity, alphabetical o relevance

{

"id": 452,

"title": "Call Of Duty: Warzone",

"thumbnail": "https:\/\/www.freetogame.com\/g\/452\/thumbnail.jpg",

"status": "Live",

"short\_description": "A standalone free-to-play battle royale and modes accessible via Call of Duty: Modern Warfare.",

"description": "Call of Duty: Warzone is both a standalone free-to-play battle royale and modes accessible via Call of Duty: Modern Warfare. Warzone features two modes \u2014 the general 150-player battle royle, and \u201cPlunder\u201d. The latter mode is described as a \u201crace to deposit the most Cash\u201d. In both modes players can both earn and loot cash to be used when purchasing in-match equipment, field upgrades, and more. Both cash and XP are earned in a variety of ways, including completing contracts.\r\n\r\nAn interesting feature of the game is one that allows players who have been killed in a match to rejoin it by winning a 1v1 match against other felled players in the Gulag.\r\n\r\nOf course, being a battle royale, the game does offer a battle pass. The pass offers players new weapons, playable characters, Call of Duty points, blueprints, and more. Players can also earn plenty of new items by completing objectives offered with the pass.",

"game\_url": "https:\/\/www.freetogame.com\/open\/call-of-duty-warzone",

"genre": "Shooter",

"platform": "Windows",

"publisher": "Activision",

"developer": "Infinity Ward",

"release\_date": "2020-03-10",

"freetogame\_profile\_url": "https:\/\/www.freetogame.com\/call-of-duty-warzone",

"minimum\_system\_requirements": {

"os": "Windows 7 64-Bit (SP1) or Windows 10 64-Bit",

"processor": "Intel Core i3-4340 or AMD FX-6300",

"memory": "8GB RAM",

"graphics": "NVIDIA GeForce GTX 670 \/ GeForce GTX 1650 or Radeon HD 7950",

"storage": "175GB HD space"

},

"screenshots": [

{

"id": 1124,

"image": "https:\/\/www.freetogame.com\/g\/452\/Call-of-Duty-Warzone-1.jpg"

},

{

"id": 1125,

"image": "https:\/\/www.freetogame.com\/g\/452\/Call-of-Duty-Warzone-2.jpg"

},

{

"id": 1126,

"image": "https:\/\/www.freetogame.com\/g\/452\/Call-of-Duty-Warzone-3.jpg"

},

{

"id": 1127,

"image": "https:\/\/www.freetogame.com\/g\/452\/Call-of-Duty-Warzone-4.jpg"

}

]

}

a) Crea las clases de la aplicación:

* Image: con identificador, URL y ¡un array de bytes con la imagen!.
* Plataforma: enumeración con 3 posibles valores, BROWSER, PC, ALL.
* Game: con identificador, título, miniatura (tipo Image), descripción, url para jugar, género, plataforma (de tipo Plataforma), fecha de realización (LocalDate) y una lista de imágenes.
* BrowserGame: hereda de Game y se trata de un juego para navegador, por lo que su categoría será BROWSER.

b) Haz una sencilla aplicación que, a partir de el JSON de un Game, cree un juego, pero sólo el identificador, el título, la descripción, la URL, … sin miniatura ni la lista de imágenes.

c) Haz que el juego se pueda guardar en un archivo de texto con el nombre: nombre del juego.txt y la versión toString del Game dentro de él. Emplea Java IO, no Files.

d) Como sólo nos interesan los juegos de navegador para jugar en clase mientras Pepe explica, haremos una aplicación que descargue la lista de juegos de:

<https://www.freetogame.com/api/games?platform=browser>

Empleando un InstanceCreator para que asigne la plataforma BROWSER al constructor de Game.

e) Amplía el ejercicio anterior para que también recupere las imágenes, sin los bytes, sólo la url. La miniatura tendrá siempre id igual a 0.

f) Amplía el ejercicio apartado anterior para que guarde recupere también la imagen y la almacene en el array de bytes. Ved nota [1]

g) Usando el API haz una aplicación que pida un identificador de objeto y lo descargue, tanto en un fichero de texto como las imágenes. Previamente debe “deserializar el objeto en el tipo Game”.

h) Si deseas hacer una aplicación gráfica, puedes ver la nota 2, en la que explico cómo crear un ImageIcon a partir de una array de bytes.

i) **Haz diseña una base de datos SQLite con la estructura de los datos del JSON y crea un aplicación que descargue los juegos y los guarde en la base de datos.**:

* Crea la base de datos e introduce los datos “estáticos” de la Plataforma y el Género.
* Realiza un programa que lea los archivos JSON, de la URL: <https://www.freetogame.com/api/game?id=X>, pasándole el id del Juego, desde 1 al número de juegos que consideres. Ten en cuenta que el juego podría no existir devolviendo:

{"status":0,"status\_message":"No game found with that id"}

* Lee los datos del JSON, por medio de un [JsonReader](https://javadoc.io/doc/com.google.code.gson/gson/latest/com.google.gson/com/google/gson/stream/JsonReader.html) (o un [JsonParser](https://javadoc.io/doc/com.google.code.gson/gson/latest/com.google.gson/com/google/gson/JsonParser.html)) y guárdalos en la base de datos para cada juego, teniendo en cuenta que debes realizar los pasos que hemos comentado:
  + Establecer conexión con DriverManage.getConnection.
  + Crear sentencia, Statement.
  + Ejecutar sentencia de tipo executeUpdate(INSERT INTO …).

Para evitar problemas con los caracteres especiales, comillas, etc. usa como base el siguiente ejemplo:

// Se supone que ya hemos creado la conexión y creamos una sentencia preformateada.

// Las ? son los parámetros de la sentencia.

PreparedStatement ps= conexion.prepareStatement("UPDATE Filosofo set nome=? , apelidos=? where idFilosofo=?"); // Sólo se realiza al principio y luego se reutiliza para cada inserción o actualización. CON INSERT sería igual, cambiando la sentencia.

// asignamos los parámetros a la consulta. En nuestro caso serían los valores a insertar en las tablas.

ps.setString(1, "Ludwig Josef Johann");

ps.setString(2, "Wittgenstein");

ps.setLong (3, 1);

int filasAfectadas = ps.executeUpdate(); // típicamente devolverá 1 o 0.

// Dentro del bucle podemos volver a insertar nuevos valores sin tener que crear una nueva sentencia.

ps.setString(1, "Bertrand Arthur William");

ps.setString(2, "Russell");

ps.setLong (3, 1);

int filasAfectadas = ps.executeUpdate();

**Nota 1. Guardar una imagen en un array de bytes:**

import java.io.ByteArrayOutputStream;

import java.io.File;

import java.io.FileInputStream;

import java.io.IOException;

public class ImageToBytes {

public static void main(String[] args) {

try {

// 1. Crea un objeto fis de tipo InputStream a la imagen.

// Ya deberías saberlo

// 2. Crea un flujo de salida a un array de bytes:

ByteArrayOutputStream bos = new ByteArrayOutputStream();

byte[] buf = new byte[1024]; // buffer

for (int readNum; (readNum = fis.read(buf)) != -1;) {

// Escribimos en el array de bytes

bos.write(buf, 0, readNum);

}

// Convertimos el flujo de bytes en un array

byte[] bytes = bos.toByteArray();

System.out.println("Imagen convertida a bytes: " + bytes);

} catch (IOException e) {

// …

}

}

}

**Nota 2. ImageIcon a partir de un array de bytes:**

import java.io.FileOutputStream;

import java.io.IOException;

public class BytesToImageFile {

public static void main(String[] args) {

try {

byte[] bytes = new byte[] { 0x00, 0x01, 0x02, …};

// Ya sabes que hay mejores maneras de crear flujos, más

// eficientes, pero a modo de ejemplo.

FileOutputStream fos =

new FileOutputStream("ruta/a/tu/imagen.jpg");

fos.write(bytes);

fos.close(); // mejor, try-catch-with-resources.

System.out.println("Imagen guardada en disco.");

} catch (IOException e) {

// ..

}

}

}

# 02. Conexiones a BD

* [La interface Connection](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0102conexiones/#la-interface-connection)
  + [1. La clase DriverManager](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0102conexiones/#1-la-clase-drivermanager)
  + [2. Especificando URL de Conexión a la Base de Datos](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0102conexiones/#2-especificando-url-de-conexi%C3%B3n-a-la-base-de-datos)
    - [Java DB Database Connection URLs: Derby](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0102conexiones/#java-db-database-connection-urls-derby)
    - [MySQL Connector/J Database URL](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0102conexiones/#mysql-connectorj-database-url)
    - [Otros sistemas gestores de bases de datos](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0102conexiones/#otros-sistemas-gestores-de-bases-de-datos)

## La interface Connection

Como se ha comentado anteriormente, la fuente de datos puede ser cualquier fuente que tenga un controlador JDBC: un sistema de gestión de bases de datos (DBMS), un sistema de archivos heredado u otra fuente de datos.

Existen dos clases principales para establecer la conexión:

* **DriverManager:** clase totalmente implementada que **conecta una aplicación a una fuente de datos**, indicada por medio de una URL de base de datos.  
  Cuando esta clase intenta establecer una conexión por primera vez, **carga automáticamente cualquier controlador JDBC 4.0 (o superior) encontrado dentro del classpath**. Se precisa cargar de manera manual cualquier controlador con JDBC inferior a 4.0
* **DataSource:**: interfaz se prefiere sobre DriverManager porque **permite que los detalles sobre la fuente de datos subyacente sean transparentes para tu aplicación**. Las propiedades de un objeto DataSource se configuran para que represente una fuente de datos concreta. Es el modo de conexión preferido para aplicaciones Java EE.

 DataSource

En principio emplearemos la clase DriverManager en lugar de la clase DataSource porque es más fácil de usar y, en general, las aplicaciones cliente multiplataforma no requieren las características avanzadas de la clase DataSource que inicialmente estaban diseñadas para ser empleadas en aplicaciones Java EE.

### 1. La clase DriverManager

Para establecer la conexión con la clase DriverManager sew invoca al método DriverManager.getConnection. El siguiente método de ejemplo establece una conexión de base de datos:

// Suponemos que este método está declarado dentro de una clase ConnectionManager que tiene

// - Un atributo sxbd de tipo String, para guardar el nombre del SGBD.

// - Un atributo con el nombre del servidor, nomeServidor

// - Un atributo con el número de puerto.

// - Un atributo para el usuario y otro para la contraseña.

public Connection getConnection() throws SQLException {

Connection conn = null;

// Propiedades de la conexión

Properties propiedadesCon = new Properties();

propiedadesCon.put("user", this.userName);

propiedadesCon.put("password", this.password);

if (this.sxbd.equals("mysql")) {

conn = DriverManager.getConnection("jdbc:" + this.sxbd + "://" +

this.nomeServidor + ":" + this.puerto + "/", propiedadesCon);

} else if (this.sxbd.equals("derby")) {

conn = DriverManager.getConnection("jdbc:" + this.sxbd + ":" + this.dbName +

";create=true", propiedadesCon);

}

System.out.println("Conexión establecida con la BD");

return conn;

}

El método DriverManager.getConnection(...) establece una conexión de base de datos. Este método requiere una **URL de base de datos**, que varía según el SGBD/DBMS. El método devuelve un objeto Connection, que representa una conexión con el DBMS o una base de datos específica. Consulta la base de datos a través de este objeto.

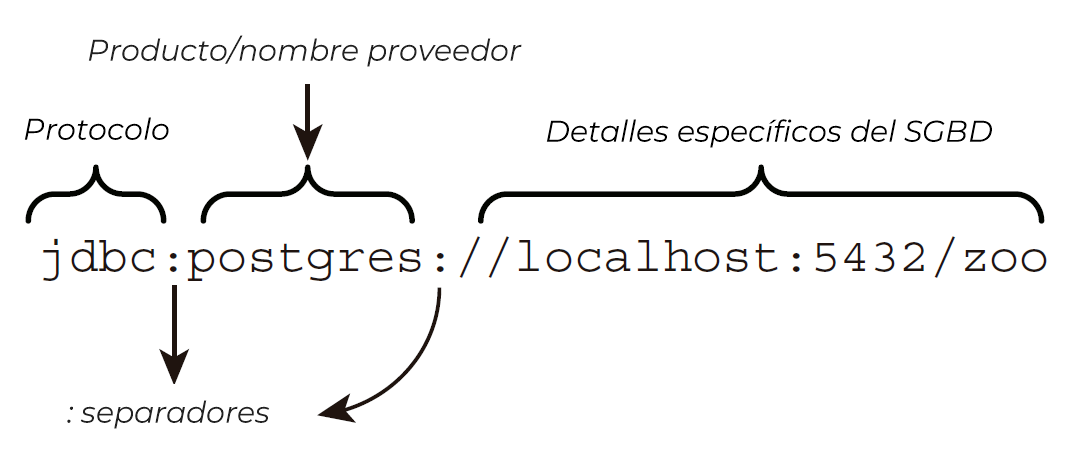
Por ejemplo:

* **MySQL:** jdbc:mysql://localhost:3306/, donde localhost es el nombre del servidor que aloja a la base de datos y 3306 es el número de puerto.
* **Java DB:** jdbc:derby:testdb;create=true, donde testdb es el nombre de la base de datos a la que conectarse y create=true informa al SGBD a crear la base de datos.

Nota:esta URL establece una conexión de base de datos con el ***controlador Java DB Embedded***. Java DB también incluye un controlador Network Client, que utiliza una URL diferente.

El método anterior especifica el nombre de usuario y la contraseña necesarios para acceder al DBMS con un objeto Properties.

Cuando se usa un driver de un proveedor, la documentación informará del subprotocolo que usa, esto es, que pone después de “jdbc:subprotocolo:….” en la URL.



Ejemplos:

**SQLite:** Driver: org.sqlite.JDBC URL: jdbc:sqlite:rutaArchivo

**JDBC-ODBC:** Driver: sun.jdbc.odbc.JdbcOdbcDriver URL: jdbc:odbc:dsn

**MySQL:** Driver: com.mysql.cj.jdbc.Driver URL: jdbc:mysql://localhost:3306/nomeBD

**H2Database:** Driver: org.h2.Driver URL: jdbc:h2:rutaArchivo

**Oracle:** Driver: oracle.jdbc.driver.OracleDriver URL: jdbc:oracle:thin@host:porto:bd

**HSQLDB** Driver: org.hsqldb.jdbc.JDBCDriver (hsqldb.jar) URL: jdbc:hsqldb:file:{folder} URL: jdbc:hsqldb:hsql://{HOST}[:{PORT}]

Nota. URL JDBC y Drivers

* Generalmente, en la URL de la base de datos, también especifica el **nombre de una base de dato**s existente a la que deseas conectarte.Por ejemplo: jdbc:mysql://localhost:3306/jugadores representa la URL de conexión a una base de datos MySQL llamada “jugadores”. En los caso de bases de datos en memoria no se especifica, porque debe ser creada previamente.
* En ve**rsiones anteriores de JDBC a 4.0**, para obtener una conexión, había que cargar previamente el Drive invocando al método estático Class.forName, que debía recoger un objeto de tipo java.sql.Driver.
* Cada **Driver JDBC contiene una o más clases que implementan la interfaz java.sql.Driver**. Los controladores para Java DB son org.apache.derby.jdbc.EmbeddedDriver y org.apache.derby.jdbc.ClientDriver, y el de MySQL Connector/J es com.mysql.cj.jdbc.Driver. En los ejemplo anteriores hemos visto cómo se denominan las implementaciones para otros SGBD que emplearemos durante esta unidad y parte del curso.

Cualquier **Driver JDBC 4.0 o superior** que se encuentre en el classpath (o en las implementaciones o dependencias del proyecto Java) **se carga automáticamente**. (Sin embargo, **debe cargarse de manera explícita cualquier Driver anterior a JDBC 4.0 con el método Class.forName**).

### 2. Especificando URL de Conexión a la Base de Datos

Una URL de conexión de base de datos es una cadena que el controlador JDBC utiliza para conectarse a una base de datos. Puede contener información sobre **dónde buscar la base de datos, el nombre de la base de datos a la que conectarse y propiedades de configuración**.

La **sintaxis exacta de una URL de conexión de base de datos depende del SGBD**:

#### C:\Users\a24aliciama\Pictures\urlconnection (1).pngJava DB Database Connection URLs: Derby

La siguiente es la sintaxis de la URL de conexión de base de datos para Java DB:

jdbc:derby:[subsubprotocol:][databaseName][;attribute=value]\*

* subsubprotocol especifica dónde Java DB debe buscar la base de datos, ya sea en un directorio, en memoria, en un classpath o en un archivo JAR. Típicamente, se omite.
* databaseName es el **nombre de la base de datos** a la que conectarse.
* attribute=value representa una **lista opcional separada por punto y coma de atributos**. Estos atributos permiten informar al SGBD de los parámetros de conexión, incluyendo la:
  + **Creación de la base de datos** indicada en la URL.
  + **Encriptación** de la base de datos.
  + Indicar **directorios** para almacenar información de log y traza.
  + Indicar el **nombre de usuario y contraseña para conectarse a la base de datos**.

Referencias:

Java DB es un distribución de la base de datos Open Source Apache Derby. Desde el 2015, JAvaDB no se incluye en JDK y fue eliminado de JDK 7 y 8 el 17 de julio del 2018. JavaDB ha sido redirigido a Apache Derby, para emplear JavaDB debe usarse la versión del Proyecto Apache Derby.

[Apache Derby](https://db.apache.org/derby/) [Apache Derby: Quick Start](https://db.apache.org/derby/quick_start.html)

#### MySQL Connector/J Database URL

La siguiente es la sintaxis de la URL de conexión de base de datos para MySQL Connector/J:

jdbc:mysql://[host][,failoverhost...][:port]/[database][?propertyName1][=propertyValue1][&propertyName2][=propertyValue2]...

* host:port es el nombre de host y el número de puerto de la computadora que aloja la base de datos. Si no se especifica, los valores predeterminados de host y puerto son 127.0.0.1 y 3306, respectivamente.
* database es el nombre de la base de datos a la que conectarse. Si no se especifica, se realiza una conexión sin una base de datos predeterminada.
* failover es el nombre de una base de datos en espera (MySQL Connector/J admite failover).
* propertyName=propertyValue representa una lista opcional separada por ampersand de propiedades. Estas propiedades te permiten informar a MySQL Connector/J la realización de varias tareas y configuraciones.

Ejemplo con MariaDB:

**El conector más reciente de MariaDB es compatible con MySQL 5.5.3 o superior**, además de con MariaDB.

Se ajustan a las especificaciones de JDBC 4.2.

Class.forName("org.mariadb.jdbc.Driver"); // Sigue funcionando pero no se precisa.

Connection connection = DriverManager.getConnection("jdbc:mariadb://localhost:3306/DB?user=root&password=myPassword");

Todos los parámetros de conexión pueden consultarse en:

<https://mariadb.com/kb/en/about-mariadb-connector-j/>

**Referencias:**

[MariaDB Connector/J](https://mariadb.com/kb/en/about-mariadb-connector-j/) [Descarga de conectores MariaDB](https://mariadb.org/download/?t=mariadb&p=mariadb&r=11.3.0&os=windows&cpu=x86_64&pkg=msi&m=fe_up_pt) [MySQL Documentation](https://dev.mysql.com/doc/) [Conector JDBC de MySQL](https://dev.mysql.com/doc/relnotes/connector-j/en/news-8-2-0.html)

#### Otros sistemas gestores de bases de datos

Cuando se usa un driver de un proveedor, la documentación informará del subprotocolo que usa, esto es, que pone después de “jdbc:subprotocolo:….” en la URL.

**SQLite:** Driver: org.sqlite.JDBC URL: jdbc:sqlite:rutaArchivo

**JDBC-ODBC:** Driver: sun.jdbc.odbc.JdbcOdbcDriver URL: jdbc:odbc:dsn

**MySQL:** Driver: com.mysql.cj.jdbc.Driver URL: jdbc:mysql://localhost:3306/nomeBD

**H2Database:** Driver: org.h2.Driver URL: jdbc:h2:rutaArchivo

**Oracle:** Driver: oracle.jdbc.driver.OracleDriver URL: jdbc:oracle:thin@host:porto:bd

**HSQLDB** Driver: org.hsqldb.jdbc.JDBCDriver (hsqldb.jar) URL: jdbc:hsqldb:file:{folder} URL: jdbc:hsqldb:hsql://{HOST}[:{PORT}]

# 03. Excepciones SQLException

* [Gestión de Excepciones SQLException](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0103excepcionessql/#gesti%C3%B3n-de-excepciones-sqlexception)
  + [1. Captura de excepciones](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0103excepcionessql/#1-captura-de-excepciones)
  + [2. Recuperación de warnings](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0103excepcionessql/#2-recuperaci%C3%B3n-de-warnings)
  + [3. SQLExceptions categorizadas](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0103excepcionessql/#3-sqlexceptions-categorizadas)
  + [4. Otras Subclases de SQLException](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0103excepcionessql/#4-otras-subclases-de-sqlexception)

## Gestión de Excepciones SQLException

Cuando **JDBC encuentra un error** durante una interacción con una base de datos a la que está conectado un objeto Connection, **lanza una instancia de SQLException**.La instancia de SQLException contiene la siguiente información que facilita encontrar la causa del error:

* Una **descripción del error**. Recupera el objeto String que contiene esta descripción **llamando al método SQLException.getMessage()**.
* Un **código SQLState**: códigos y significados estandarizados por ISO/ANSI y Open Group (X/Open), aunque algunos códigos se han reservado para que los definan los proveedores de bases de datos. Este objeto String consta de **cinco caracteres alfanuméricos**. Recupera este código **llamando al método SQLException.getSQLState()**.
* Un **código de error**: valor entero que identifica el error que causó que la instancia de SQLException se lanzara. Su valor y significado son **específicos de la implementación** y podrían ser el código de error real devuelto por la fuente de datos. Recupera el error **llamando al método SQLException.getErrorCode()**.
* Una **causa**. Una instancia de SQLException podría tener una relación causal, que consiste en **uno o más objetos Throwable que causaron que la instancia de SQLException se lanzara**. Para navegar por esta cadena de causas, **llama recursivamente al método SQLException.getCause() hasta que se devuelva un valor nulo**.
* Una referencia a excepciones encadenadas. Si ocurren más de un error, las excepciones se referencian a través de esta cadena. Recupera estas excepciones llamando al método SQLException.getNextException en la excepción que se lanzó.

### 1. Captura de excepciones

El siguiente método, printSQLException, muestra el SQLState, el código de error, la descripción del error y la causa (si la hay) contenidos en SQLException, así como cualquier otra excepción encadenada:

public static void printSQLException(SQLException ex) {

for (Throwable e : ex) {

if (e instanceof SQLException) {

// Método implantado más adelante:

if (!ignoraSQLException(((SQLException)e).getSQLState())) {

// e.printStackTrace(System.err);

System.err.println("Estado SQL: " + ((SQLException)e).getSQLState());

System.err.println("Código error: " + ((SQLException)e).getErrorCode());

System.err.println("Mensaje: " + e.getMessage());

Throwable t = ex.getCause();

while(t != null) {

System.out.println("Causa: " + t);

t = t.getCause(); // LLamada recursiva.

}

}

}

}

}

Por ejemplo, si se invoca una llamada da una tabla que no existe la llamada al método ignoraSQLException, la salida será similar a la siguiente:

Estado SQL: 42Y55

Código error: 30000

Mensaje: 'DROP TABLE' cannot be performed on

'TESTDB.TABLAPRUEBA' because it does not exist.

En lugar de imprimir información de SQLException, podrías en su lugar primero recuperar el SQLState y procesar SQLException en consecuencia. Por ejemplo, el método ignoraSQLException devuelve true si el SQLState es igual al código 42Y55 (y estás utilizando Java DB como tu DBMS), lo que provoca que printSQLException ignore la SQLException:

public static boolean ignoraSQLException(String sqlState) {

if (sqlState == null) {

System.out.println("Este estado SQL no está declarado!");

return false;

}

// X0Y32: Jar file already exists in schema

if (sqlState.equalsIgnoreCase("X0Y32"))

return true;

// 42Y55: Table already exists in schema

if (sqlState.equalsIgnoreCase("42Y55"))

return true;

return false;

}

### 2. Recuperación de warnings

Los objetos **SQLWarning son una subclase de SQLException** que gestiona los **warnings de acceso a la base de datos**.

Los warnings **no detienen la ejecución de una aplicación**, como lo hacen las excepciones; simplemente alertan al usuario de que algo no sucedió según lo planeado. Por ejemplo, una advertencia podría informar que un privilegio que intentaste revocar no se revocó. O una advertencia podría decir que ocurrió un error durante una desconexión solicitada.

Una advertencia **se puede informar en un objeto Connection**, un **objeto Statement** (incluidos los objetos PreparedStatement y CallableStatement) o un **objeto ResultSet**.

Cada **una de estas interfaces (y sus clasesimplementadas) tiene un método getWarnings()**, que se debe invocar para ver la primera advertencia informada en el objeto que llama.

Si **getWarnings devuelve una advertencia, se llamar al método getNextWarning de SQLWarning en ella para obtener cualquier advertencia adicional**. La ejecución de una instrucción borra automáticamente las advertencias de una instrucción anterior, por lo que no se acumulan.

**Si se desea recuperar advertencias informadas en una orden, debe hacerse antes de ejecutar otra instrucción** de cierre.

Por ejemplo, para acceder a cualquier advertencia informada en objetos Statement o ResultSet:

public static void getWarningsFromResultSet(ResultSet rs) throws SQLException {

printWarnings(rs.getWarnings());

}

public static void getWarningsFromStatement(Statement stmt) throws SQLException {

printWarnings(stmt.getWarnings());

}

public static void printWarnings(SQLWarning warning) throws SQLException {

if (warning != null) {

System.out.println("\n---Warning---\n");

while (warning != null) {

System.out.println("Mensaje: " + warning.getMessage());

System.out.println("SQLState: " + warning.getSQLState());

System.out.print("Código de error del proveedor: ");

System.out.println(warning.getErrorCode());

System.out.println("");

warning = warning.getNextWarning();

}

}

La advertencia más común es una advertencia de **DataTruncation**, una **subclase de SQLWarning**. Todos los objetos DataTruncation tienen un **SQLState de 01004**, lo que indica que hubo un problema al leer o escribir datos.

Los métodos de DataTruncation te permiten averiguar en qué columna o parámetro se truncaron los datos, si fue en una operación de lectura o escritura, cuántos bytes deberían haberse transferido y cuántos bytes se transfirieron realmente.

### 3. SQLExceptions categorizadas

Tu controlador JDBC podría lanzar una subclase de SQLException que corresponde a un SQLState común o a un estado de error común que no está asociado con un valor de clase SQLState específico, permitiendo concretar la excepción que produjo error:

* **SQLNonTransientException**
* **SQLTransientException**
* **SQLRecoverableException**
* BatchUpdateException.
* RowSetWarning
* SerialException
* SQLClientInfoException
* **SQLWarning**
* SyncFactoryException
* SyncProviderException

Consulta la última [documentación de Javadoc del paquete java.sql](https://docs.oracle.com/en/java/javase/21/docs/api/java.sql/java/sql/SQLException.html).

### 4. Otras Subclases de SQLException

Las siguientes subclases de SQLException también pueden lanzarse:

* BatchUpdateException se lanza cuando ocurre un error durante una operación de actualización por lotes. Además de la información proporcionada por SQLException, BatchUpdateException proporciona las cuentas de actualización para todas las declaraciones que se ejecutaron antes de que ocurriera el error.
* SQLClientInfoException se lanza cuando no se pueden establecer una o más propiedades de información del cliente en una Connection. Además de la información proporcionada por SQLException, SQLClientInfoException proporciona una lista de propiedades de información del cliente que no se establecieron.

# 04. ResultSet

* [Recuperación y actualización con ResultSet](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0104resultset/#recuperacin-y-actualizacin-con-resultset)
  + [1. Interfaz ResultSet](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0104resultset/#1-interfaz-resultset)
    - [A) Tipos de ResultSet](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0104resultset/#a-tipos-de-resultset)
    - [B) Concurrencia de ResultSet (actualizable o no)](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0104resultset/#b-concurrencia-de-resultset-actualizable-o-no)
    - [C) Retención del Cursor (permanece abiertos cuando se llama al método commit)](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0104resultset/#c-retencin-del-cursor-permanece-abiertos-cuando-se-llama-al-mtodo-commit)
  + [2. Recuperación de valores (por filas)](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0104resultset/#2-recuperacin-de-valores-por-filas)
  + [3. Moviendo el cursor](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0104resultset/#3-moviendo-el-cursor)
  + [4. Actualización de Filas en Objetos ResultSet](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0104resultset/#4-actualizacin-de-filas-con-resultset)

## Recuperación y actualización con ResultSet

Un objeto **ResultSet** mantiene un cursor que apunta a su fila actual de datos.

<https://docs.oracle.com/en/java/javase/21/docs/api/java.sql/java/sql/ResultSet.html>

El siguiente método, showCafes, muestra el contenido de la tabla Cafes y demuestra el uso de objetos [**ResultSet**](https://docs.oracle.com/en/java/javase/21/docs/api/java.sql/java/sql/ResultSet.html) y cursores:

public static void showCafes(Connection con) throws SQLException {

String query = "select nome, idProveedor, precio, ventas, total from Cafe";

try (Statement stmt = con.createStatement()) {

ResultSet rs = stmt.executeQuery(query);

while (rs.next()) {

String nombreCafe = rs.getString("nome");

int idProveedor = rs.getInt("idProveedor");

float precio = rs.getFloat("precio");

int ventas = rs.getInt("ventas");

int total = rs.getInt("total");

System.out.println(nombreCafe + ", " + idProveedor + ", " + precio +

", " + ventas + ", " + total);

}

} catch (SQLException e) {

// Manejo de excepciones

}

}

Un objeto ResultSet es una **tabla de datos que representa un conjunto de resultados de una base de datos**, usualmente mediante la ejecución de una declaración que consulta la base de datos.

Por ejemplo, el método showCafes crea un ResultSet, rs, cuando **ejecuta la consulta a través del objeto**[**Statement**](https://docs.oracle.com/en/java/javase/21/docs/api/java.sql/java/sql/Statement.html), stmt.

Un objeto **ResultSet se puede crear a través de cualquier objeto que implemente la interfaz Statement**:

* [Statement](https://docs.oracle.com/en/java/javase/21/docs/api/java.sql/java/sql/Statement.html)
* [PreparedStatement](https://docs.oracle.com/en/java/javase/21/docs/api/java.sql/java/sql/PreparedStatement.html)
* [CallableStatement](https://docs.oracle.com/en/java/javase/21/docs/api/java.sql/java/sql/CallableStatement.html)
* [RowSet](https://docs.oracle.com/en/java/javase/21/docs/api/java.sql/javax/sql/RowSet.html).

Se accede a los datos en un objeto ResultSet a través de un cursor, que **no es un cursor de base de datos**.

* Un objeto ResultSet es un **puntero que apunta a una fila de datos en el ResultSet**.
* Inicialmente, el cursor **se sitúa antes de la primera fila**.
* El método **ResultSet.next() mueve el cursor a la siguiente fila**. Este método devuelve false si el cursor está **situado después de la última fila**.
* **ResultSet.next() se llama repetidamente al método ResultSet.next()** con un bucle while para iterar a través de todos los datos en el ResultSet.

Veremos a continuación:

* Interfaz **ResultSet**
* Recuperación de valores de columnas de cada fila/registro.
* **Cursores**
* **Actualización** de Filas en Objetos ResultSet
* Uso de Objetos **Statement** para **Actualizaciones batch**
* Inserción de Filas en Objetos ResultSet

### 1. Interfaz ResultSet

La [interfaz ResultSet](https://docs.oracle.com/en/java/javase/21/docs/api/java.sql/java/sql/ResultSet.html) dispone de métodos para recuperar y manipular los resultados de consultas ejecutadas. Pueden crearse objetos ResultSet con funcionalidades y características diferentes:

* **Tipo de cursor**.
* **Concurrencia**.
* **“Retención” del cursor**.

#### A) Tipos de ResultSet

El primer argumento de los métodos createStatement, prepareStatement y prepareCall de Connection es el tipo de ResultSet.

El tipo de un objeto ResultSet **determina el nivel de funcionalidad** en dos aspectos:

* Las **formas en que se puede manipular el cursor** (hacia adelante, hacia atrás, a una posición absoluta y así sucesivamente).
* **Cómo se reflejan los cambios concurrentes** realizados en la fuente de datos (base de datos) mediante el objeto ResultSet: si se reflejan o no y cuándo se reflejan.

La sensibilidad de un objeto ResultSet está determinada por uno de los tres tipos diferentes de ResultSet:

* **TYPE\_FORWARD\_ONLY: el cursor se mueve solo hacia adelante**, desde antes de la primera fila hasta después de la última fila. A veces se recupera fila a fila y no todos los resultados de una vez.
* **TYPE\_SCROLL\_INSENSITIVE: el cursor puede moverse hacia adelante y hacia atrás con respecto a la posición actual**, y puede moverse a una **posición absoluta**. El ResultSet **no es sensible a los cambios realizados en la base de datos** mientras está abierto.
* **TYPE\_SCROLL\_SENSITIVE: el cursor puede moverse hacia adelante y hacia atrás con respecto a la posición actual**, y puede moverse a una **posición absoluta**. El ResultSet **refleja los cambios realizados en la base de datos subyacente** mientras está abierto.

El tipo de ResultSet [**predeterminado**](https://docs.oracle.com/en/java/javase/21/docs/api/java.sql/java/sql/ResultSet.html#TYPE_FORWARD_ONLY)**es TYPE\_FORWARD\_ONLY**.

Nota: No todas las bases de datos y controladores JDBC admiten todos los tipos de ResultSet. El [**método DatabaseMetaData.supportsResultSetType**](https://docs.oracle.com/en/java/javase/21/docs/api/java.sql/java/sql/DatabaseMetaData.html#supportsResultSetType(int))**devuelve true si el tipo de ResultSet especificado es compatible y false en caso contrario**.

#### B) Concurrencia de ResultSet (actualizable o no)

Es el **segundo argumento de la createStatement, prepareStatement o prepareCall** de Connection es la concurrencia.

La concurrencia de un objeto ResultSet determina **qué nivel de funcionalidad de actualización se admite**.

Hay dos niveles de concurrencia:

* **CONCUR\_READ\_ONLY: ResultSet no se puede actualizar**.
* **CONCUR\_UPDATABLE: ResultSet se puede actualizar**.

La concurrencia **predeterminada de ResultSet es CONCUR\_READ\_ONLY**.

Nota: No todos los controladores JDBC y bases de datos admiten la concurrencia. El método DatabaseMetaData.supportsResultSetConcurrency devuelve true si el nivel de concurrencia especificado es compatible con el controlador y false en caso contrario.

Comprobación de si un ResultSet admite determinados niveles de concurrencia, tipo y actualización:

try {

if (con.getMetaData().supportsResultSetType(ResultSet.TYPE\_SCROLL\_INSENSITIVE)) {

System.out.println("Soporta TYPE\_SCROLL\_INSENSITIVE");

} else {

System.out.println("No soporta TYPE\_SCROLL\_INSENSITIVE");

}

if (con.getMetaData().supportsResultSetConcurrency(ResultSet.TYPE\_SCROLL\_INSENSITIVE,

ResultSet.CONCUR\_UPDATABLE)) {

System.out.println("Soporta CONCUR\_UPDATABLE");

} else {

System.out.println("No soporta CONCUR\_UPDATABLE");

}

} catch (SQLException ex) {

System.out.println("Error al obtener metadatos: " + ex.getMessage());

}

El siguiente ejemplo muestra cómo usar un objeto ResultSet cuyo nivel de concurrencia es CONCUR\_UPDATABLE:

Statement stmt = con.createStatement(ResultSet.TYPE\_SCROLL\_INSENSITIVE, ResultSet.CONCUR\_UPDATABLE);

ResultSet rs = stmt.executeQuery("SELECT a, b FROM Tabla1");

// rs será desplazable y no mostrará cambios realizados por otros.

// Será actualizable.

El método actualizarPrecios demuestra cómo usar un objeto ResultSet cuyo nivel de concurrencia es CONCUR\_UPDATABLE:

public void actualizarPrecios(float porcetaje) throws SQLException {

try (Statement stmt = con.createStatement(ResultSet.TYPE\_SCROLL\_SENSITIVE, ResultSet.CONCUR\_UPDATABLE)) {

ResultSet uprs = stmt.executeQuery("SELECT \* FROM Cafe");

while (uprs.next()) {

float f = uprs.getFloat("precio");

uprs.updateFloat("precio", f \* porcetaje);

uprs.updateRow();

}

} catch (SQLException e) {

// Manejo de excepciones

}

}

#### C) Retención del Cursor (permanece abiertos cuando se llama al método commit)

Llamar al [método Connection.commit](https://docs.oracle.com/en/java/javase/21/docs/api/java.sql/java/sql/Connection.html#commit()) (confirmar la transacción) puede cerrar los objetos ResultSet que se hayan creado durante la transacción actual.  
En algunos casos, esto puede no ser el deseado. La propiedad **holdability del ResultSet le da a la aplicación control sobre si los objetos ResultSet (cursores) se cierran cuando se llama a commit**.

Las siguientes constantes de ResultSet se pueden suministrar a los métodos createStatement, prepareStatement y prepareCall de Connection:

* HOLD\_CURSORS\_OVER\_COMMIT: los cursores ResultSet no se cierran; **permanecen abiertos cuando se llama al método commit**. Los cursores retenidos pueden ser ideales si la aplicación utiliza principalmente objetos ResultSet de solo lectura.
* CLOSE\_CURSORS\_AT\_COMMIT: Los objetos ResultSet (cursores) **se cierran cuando se llama al método commit**. Cerrar cursores al llamar a este método puede dar **mejor rendimiento** para algunas aplicaciones.

La **retención predeterminada del cursor varía según el SGBD**.

 Retención predeterminada del cursor

Nota: No todos los controladores JDBC y bases de datos admiten cursores retenibles y no retenibles. El método DatabaseMetaData.supportsResultSetHoldability devuelve true si el nivel de retención especificado es compatible con el controlador y false en caso contrario.

El método admiteRetencion muestra la retención predeterminada del cursor de los objetos ResultSet y si se admiten HOLD\_CURSORS\_OVER\_COMMIT y CLOSE\_CURSORS\_AT\_COMMIT:

public static void admiteRetencion(Connection conn) throws SQLException {

DatabaseMetaData dbMetaData = conn.getMetaData();

System.out.println("ResultSet.HOLD\_CURSORS\_OVER\_COMMIT = " +

ResultSet.HOLD\_CURSORS\_OVER\_COMMIT);

System.out.println("ResultSet.CLOSE\_CURSORS\_AT\_COMMIT = " +

ResultSet.CLOSE\_CURSORS\_AT\_COMMIT);

System.out.println("Retención predeterminada del cursor: " +

dbMetaData.getResultSetHoldability());

System.out.println("¿Admite HOLD\_CURSORS\_OVER\_COMMIT? " +

dbMetaData.supportsResultSetHoldability(

ResultSet.HOLD\_CURSORS\_OVER\_COMMIT));

System.out.println("¿Admite CLOSE\_CURSORS\_AT\_COMMIT? " +

dbMetaData.supportsResultSetHoldability(

ResultSet.CLOSE\_CURSORS\_AT\_COMMIT));

}

### 2. Recuperación de valores (por filas)

ResultSet tiene métodos getXXX (por ejemplo, getBoolean(indice/nombre) y getLong(indice/nombre) para recuperar valores de columna desde la fila actual:

int getInt(int columnIndex/String columnName):

Date getDate(int columnIndex/String columnName);

String getString(int columnIndex/String columnName);

double getDouble(int columnIndex/String columnName);

// ...

* Se pueden **recuperar valores utilizando el número de índice de la columna o el alias o nombre de la columna**.
* El **índice de columna suele ser más eficiente**. Las columnas se numeran **a partir de 1**.
* Para máxima **portabilidad**, las columnas del conjunto de resultados dentro de cada fila **deben leerse en orden de izquierda a derecha, y cada columna debe leerse solo una vez**.

Por ejemplo, el siguiente método, showCafesPorIndice, recupera valores de columna por número:

public static void showCafesPorIndice(Connection con) throws SQLException {

String query = "select nome, idProveedor, precio, ventas, total from Cafe";

try (Statement stmt = con.createStatement()) {

ResultSet rs = stmt.executeQuery(query);

while (rs.next()) {

String nombreCafe = rs.getString(1);

int idProveedor = rs.getInt(2);

float precio = rs.getFloat(3);

int ventas = rs.getInt(4);

int total = rs.getInt(5);

System.out.println(nombreCafe + ", " + idProveedor + ", " + precio +

", " + ventas + ", " + total);

}

} catch (SQLException e) {

// Manejo de excepciones

}

}

Los **parámetros de String de todos los métodos de get no distinguen mayúsculas de minúsculas**.

Una llamada a un método get con String y más de una columna tiene el mismo alias o nombre, **devuelve el valor de la primera columna coincidente**.

La opción de usar una cadena en lugar de un número entero está diseñada para utilizarse cuando las columnas tienen alias o nombres en la consulta SQL que generó el conjunto de resultados.  
Para columnas que no se nombran explícitamente en la consulta (por ejemplo, select \* from Cafe), es **mejor emplear números de columnas**.

 getString con nombres únicos

Si se utilizan nombres de columna, se debe garantizar que se refieran de manera única a las columnas previstas mediante el **uso de alias de columna**, por medio de la cláusula SQL AS en la declaración SELECT.

 getString con para recuperar otros tipos de datos

Nota: se recomienda el **método getString** para recuperar los tipos de SQL CHAR y VARCHAR, pero **es posible recuperar cualquier tipo de SQL básicos con él**. Obtener todos los valores con getString puede ser muy cómodo, pero convierte el valor numérico en un objeto String de Java.

Para **tipos de datos no estándar SQL3 emplea getString**.

## 3. Moviendo el cursor

Se accede a los datos en un objeto ResultSet a través de un **cursor, que apunta a una fila en el objeto ResultSet**.

**Cuando se crea un objeto ResultSet, el cursor se sitúa antes de la primera fila**.

El método showCafes mueve el cursor llamando al método ResultSet.next(). Hay otros métodos disponibles para mover el cursor:

* next: mueve el cursor **hacia adelante una fila**. Devuelve true si el cursor está en una fila y false si se sitúa después de la última fila.
* previous: mueve el cursor **hacia atrás una fila**. Devuelve true si el cursor está en una fila y false si el cursor está antes de la primera fila.
* first: mueve el cursor a la **primera fila en el objeto ResultSet**. Devuelve true si el cursor está en la primera fila y false si el objeto ResultSet no contiene ninguna fila.
* last: mueve el cursor a la **última fila en el objeto ResultSet**. Devuelve true si el cursor está en la última fila y false si el objeto ResultSet no contiene ninguna fila.
* beforeFirst: sitúa el **cursor al comienzo del objeto ResultSet**, antes de la primera fila. Si el objeto ResultSet no contiene ninguna fila, este método no tiene efecto.
* afterLast: sitúa el **cursor al final del objeto ResultSet**, después de la última fila. Si el objeto ResultSet no contiene ninguna fila, este método no tiene efecto.
* relative(int rows): mueve el cursor **en relación con su posición actual**.
* absolute(int row): **sitúa el cursor en la fila especificada por el parámetro row**.

La sensibilidad **predeterminada de un ResultSet es TYPE\_FORWARD\_ONLY**, lo que significa que no se puede desplazar. **No se puede llamar a ninguno de estos métodos que mueven el cursor, excepto next, si el ResultSet no se puede desplazar**.

## 4. Actualización de Filas con ResultSet

No se puede actualizar un objeto ResultSet con TYPE\_FORWARD\_ONLY.

Los **ResultSet que pueden moverse (TYPE\_SCROLL\_SENSITIVE y TYPE\_SCROLL\_INSENSITIVE)** (el cursor puede moverse hacia atrás o a una posición absoluta) **pueden actualizarse**.

* Existen métodos de actualización de campos de ResultSet para todos los tipos de datos SQL: updateBoolean, updateByte, updateShort, updateInt, updateLong, updateFloat, updateDouble, updateBigDecimal, updateString, updateBytes, updateDate, updateTime, updateTimestamp, updateAsciiStream, updateBinaryStream, updateCharacterStream, updateObject.
* Estos métodos actualizan el valor de un campo en la fila actual.
* Una vez actualizado el valor de un campo, se debe llamar al método updateRow para que se haga efectivo el cambio en la base de datos.

El siguiente método, actualizaPrecios, multiplica la columna precio de cada fila por el porcentaje argumentado:

public void actualizaPrecios(float percentage) throws SQLException {

try (Statement stmt =

con.createStatement(ResultSet.TYPE\_SCROLL\_SENSITIVE, ResultSet.CONCUR\_UPDATABLE)) {

ResultSet uprs = stmt.executeQuery("SELECT \* FROM Cafe");

while (uprs.next()) {

float f = uprs.getFloat("precio");

uprs.updateFloat("precio", f \* percentaje);

uprs.updateRow();

}

} catch (SQLException e) {

// Manejo de excepciones

}

}

En el ejemplo:

* El campo ResultSet.TYPE\_SCROLL\_SENSITIVE crea un objeto ResultSet cuyo cursor puede moverse.
* El campo **ResultSet.CONCUR\_UPDATABLE crea un objeto ResultSet que se puede actualizar**. Si no se especifica, el objeto ResultSet es de solo lectura.
* El método ResultSet.updateFloat(campo, valor) actualiza la columna especificada (en este ejemplo, precio) con el valor float especificado en la fila donde está posicionado el cursor. ResultSet contiene varios métodos actualizadores que te permiten actualizar valores de columnas de varios tipos de datos. Para actualizar **debe llamarse al método ResultSet.updateRow()**.

# 05. Actualizaciones por lotes con ResultSet

* [1. Órdenes Batch (por lotes) sobre un Statement](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0105resultsetbatch/#1-%C3%B3rdenes-batch-por-lotes-sobre-un-statement)
* [2. Actualización batch parametrizada](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0105resultsetbatch/#2-actualizaci%C3%B3n-batch-parametrizada)
* [3. Excepciones en listas batch](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0105resultsetbatch/#3-excepciones-en-listas-batch)

## 1. Órdenes Batch (por lotes) sobre un Statement

* Los objetos Statement, PreparedStatement y CallableStatement tienen una lista de órdenes batch asociadas que podemos añadir con el [método addBatch(String s)](<https://docs.oracle.com/en/java/javase/21/docs/api/java>. sql/java/sql/Statement.html#addBatch(java.lang.String)).
* **No puede contener una declaración que produzca un ResultSet**, como una declaración SELECT.
* La **lista de procesos Batch sólo puede contener declaraciones que produzcan una tarea de actualización**
* (UPDATE, INSERT, etc.) o de tipo DLL (CREATE TABLE, DROP TABLE, ALTER TABLE, etc.).
* La **lista se asocia con un objeto Statement en su creación (método addBatch) y está inicialmente vacía**.
* **Se pueden añadir sentencias SQL a esta lista con el método addBatch y vaciarla con el método clearBatch**.
* Al terminar de añadir órdenes batch se invoca al **método executeBatch** para enviarlas todas a la base de datos para que **se ejecuten como una unidad o lote**.

Por ejemplo, el siguiente método, batchUpdate, añade cuatro filas a la tabla Cafe con una **actualización por lotes (batch)**:

public void batchUpdate() throws SQLException {

con.setAutoCommit(false); // deshabilita el modo de autocommit

try (Statement stmt = con.createStatement()) {

stmt.addBatch("INSERT INTO Cafe " +

"VALUES('Amaretto', 49, 9.99, 0, 0)");

stmt.addBatch("INSERT INTO Cafe " +

"VALUES('Avellana', 49, 9.99, 0, 0)");

stmt.addBatch("INSERT INTO Cafe " +

"VALUES('Amaretto\_decaf', 49, 10.99, 0, 0)");

stmt.addBatch("INSERT INTO Cafe " +

"VALUES('Avellana\_decaf', 49, 10.99, 0, 0)");

int[] updateCounts = stmt.executeBatch();

con.commit();

} catch (BatchUpdateException b) {

//

} catch (SQLException ex) {

//

} finally {

con.setAutoCommit(true);

}

}

La línea siguiente deshabilita el modo de autocommit para el objeto Connection con, de modo que **la transacción NO se comprometerá ni se revertirá automáticamente cuando se llame al método executeBatch**.

con.setAutoCommit(false);

Para permitir un manejo de errores correcto, **siempre debes deshabilitar el modo de autocommit antes de comenzar una actualización batch**.

Para enviar las órdenes SQL que se agregaron a la lista y que se ejecuten como un lote (st.executeBatch()):

int[] updateCounts = stmt.executeBatch();

* stmt **utiliza el método executeBatch para enviar el lote** de INSERT, no el método executeUpdate, que envía sólo una orden y devuelve un sólo recuento de actualización.
* El SGBD **ejecuta las órdenes en el orden en que se agregaron** a la lista, por lo que primero agregará la fila de valores para “Amaretto”, “Avellana”,…
* Si los cuatro comandos se ejecutan correctamente, el método **stmt.executeBatch() devuelve el recuento de actualización para cada orden SQL** en el orden en que se ejecutó. Los recuentos de actualización que indican cuántas filas afectó cada comando se almacenan en el array updateCounts (puedes llamarle cómo quieras).
* Si los cuatro comandos en el lote se ejecutan correctamente, updateCounts contendrá cuatro valores, en este caso con 1 porque una inserción afecta a una fila. La lista de comandos asociados con stmt ahora estará vacía porque los cuatro comandos agregados anteriormente se enviaron a la base de datos cuando stmt llamó al método executeBatch.
* Se puede **vaciar explícitamente esta lista de comandos** en cualquier momento con el **método clearBatch**.
* El método Connection.commit hace que el lote de actualizaciones en la tabla Cafe sea permanente. Este método debe llamarse de manera explícita, porque se deshabilitó el modo de autocommit previamente para esta conexión.

Volvemos a habilitar el modo autocommit para el objeto Connection:

con.setAutoCommit(true);

Es importante para que se vuelva a hacer commit de manera automática, y evitamos tener que llamar a commit.

## 2. Actualización batch parametrizada

También es posible realizar una actualización en lote parametrizada:

con.setAutoCommit(false);

PreparedStatement pstmt = con.prepareStatement(

"INSERT INTO Cafe VALUES( " +

"?, ?, ?, ?, ?)");

pstmt.setString(1, "Amaretto");

pstmt.setInt(2, 49);

pstmt.setFloat(3, 9.99);

pstmt.setInt(4, 0);

pstmt.setInt(5, 0);

pstmt.addBatch();

pstmt.setString(1, "Avellana");

pstmt.setInt(2, 49);

pstmt.setFloat(3, 9.99);

pstmt.setInt(4, 0);

pstmt.setInt(5, 0);

pstmt.addBatch();

// ... y así sucesivamente para cada nuevo

// tipo de café

int[] updateCounts = pstmt.executeBatch();

con.commit();

con.setAutoCommit(true);

## 3. Excepciones en listas batch

**BatchUpdateException hereda de SQLException** y obtenerse una [excepción de tipo BatchUpdateException](https://docs.oracle.com/en/java/javase/21/docs/api/java.sql/java/sql/BatchUpdateException.html) se invoca al método executeBatch si:

(1) Una de las declaraciones SQL que añadida al lote produce un ResultSet (por lo general, una consulta)

(2) Una de las declaraciones SQL no se ejecuta correctamente por alguna otra razón.

Recuerda que **no se debe agregar una consulta (una declaración SELECT) a un lote** de comandos SQL porque el método executeBatch, que devuelve un array de contador de actualizaciones, espera un recuento de actualización de cada declaración SQL que se ejecute correctamente:

* INSERT INTO, UPDATE, DELETE, que devuelven el número de filas afectadas.
* CREATE TABLE, DROP TABLE, ALTER TABLE, que devuelven 0. executeBatch.

Una BatchUpdateException **contiene un array de recuentos de actualización** similar al array devuelto por el método executeBatch. En ambos casos, los recuentos de actualización están en el mismo orden que los comandos que los produjeron.  
Esto nos sirve para saber **cuántos comandos en el lote se ejecutaron correctamente y cuáles son**. Por ejemplo, si cinco comandos se ejecutaron correctamente, el array contendrá cinco números: el primero será el recuento de actualización para el primer comando, el segundo será el recuento de actualización para el segundo comando y así sucesivamente.

El método, printBatchUpdateException, imprime toda la información de SQLException más los recuentos de actualización contenidos en un objeto BatchUpdateException.  
Dado que BatchUpdateException.getUpdateCounts devuelve un array de int, el código usa un bucle for para imprimir cada uno de los recuentos de actualización:

public static void printBatchUpdateException(BatchUpdateException b) {

System.err.println("----BatchUpdateException----");

System.err.println("SQLState: " + b.getSQLState());

System.err.println("Message: " + b.getMessage());

System.err.println("Vendor: " + b.getErrorCode());

System.err.print("Update counts: ");

int[] updateCounts = b.getUpdateCounts(); // array de int con los recuentos de actualización

for (int i = 0; i < updateCounts.length; i++) {

System.err.print(updateCounts[i] + " ");

}

}

 Batch vs transacción

**SQL Batch:**

a) **SQL Batch es una colección de sentencias** que deben ejecutarse **sin garantía de éxito o fracaso**.

b) El procesamiento por lotes significa que las cosas **se sitúan en una cola** y se procesan cuando se alcanza cierta cantidad de elementos o cuando ha transcurrido cierto período de tiempo. Se puede deshacer/retroceder en esto.

**SQL Transaction:**

a) La Transacción SQL es una colección de sentencias que están **garantizadas para tener éxito o fallar totalmente**. Las transacciones no completarán la mitad de los comandos y luego fallarán en el resto; si uno falla, todos fallan.

b) La transacción es como un **procesamiento en tiempo real que te permite deshacer/retroceder cambio**s.

En las TRANSACCIONES, es similar al lote, pero **tienes la opción de “cancelarla”**.

Por ejemplo, si el banco procesa tu solicitud de depósito y luego descubre que no tienes suficiente dinero en tu cuenta para cubrir el depósito, el banco puede cancelar la transacción y devolverte el cheque. El banco no puede hacer esto con el procesamiento por lotes.

# 06. Prepared Statement

* [Sentencias “preparadas” (PreparedStatement)](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0106preparedst/#sentencias-preparadas-preparedstatement)
  + [1. Características de PreparedStatement](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0106preparedst/#1-caracter%C3%ADsticas-de-preparedstatement)
  + [2. Creación de PreparedStatement](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0106preparedst/#2-creaci%C3%B3n-de-preparedstatement)
    - [2.1. setTipoDato(columna, valor) de PreparedStatement](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0106preparedst/#21-settipodatocolumna-valor-de-preparedstatement)
  + [3. Ejecución de sentencias con PreparedStatement: executeUpdate, executeQuery y execute.](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0106preparedst/#3-ejecuci%C3%B3n-de-sentencias-con-preparedstatement-executeupdate-executequery-y-execute)
  + [4. Valores devueltos por executeUpdate](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0106preparedst/#4-valores-devueltos-por-executeupdate)

## Sentencias “preparadas” (PreparedStatement)

La interface PreparedStatement hereda de Statement y **representa una sentencia SQL precompilada**.

### 1. Características de PreparedStatement

* En la mayoría de los casos **se recomienda el uso de PreparedStatement** para enviar sentencias SQL a la base de datos.
* **Una vez compilada, la sentencia preparada se puede ejecutar varias veces**.
* PreparedStatement es **más eficientes que las sentencias Statement** cuando se ejecutan varias veces, ya que **la sentencia SQL se analiza y se compila solo una vez**.
* **PreparedStatement también son útiles cuando se ejecutan consultas dinámicas**, ya que **permiten la separación de la sentencia SQL de los parámetros**.

En cuanto al uso, la diferencia principal de un objeto PreparedStatement es que, a diferencia de un objeto Statement, **se le proporciona una declaración SQL cuando se crea.**  
En la mayoría de los casos esta declaración SQL **se envía al SGBD de inmediato, donde se compila**. Como resultado, el objeto PreparedStatement contiene una declaración SQL que ha sido precompilada. Cuando se ejecuta el PreparedStatement, el SGB puede ejecutar la declaración SQL del PreparedStatement sin tener que compilarla primero.

* Es la opción **idónea para declaraciones SQL que toman parámetros**, pues se puede usar la misma declaración y suministrar diferentes valores cada vez que se ejecuta.
* La principal ventaja es que **evita la inyección SQL, pues los parámetros se pasan por separado de la consulta SQL**.

 Inyeccion SQL

La inyección SQL es una técnica **para explotar maliciosamente aplicaciones que utilizan datos proporcionados por el cliente en declaraciones SQL**. Los atacantes engañan al motor SQL para ejecutar comandos no deseados al suministrar una entrada de cadena especialmente diseñada, obteniendo así acceso no autorizado a una base de datos para ver o manipular datos restringidos:

<https://es.wikipedia.org/wiki/Inyecci%C3%B3n_SQL>

Las sentencias preparadas siempre tratan los datos proporcionados por el cliente como contenido de un parámetro y nunca como parte de una declaración SQL.

Ejemplo:

public void updateVentas(HashMap<String, Integer> ventasPorSemana) throws SQLException {

String updateString = "update Cafe set ventas = ? where nome = ?"; // Actualización de ventas.

String updateStatement = "update Cafe set total = total + ? where nome = ?"; // Actualización del total.

try (PreparedStatement updateVentas = con.prepareStatement(updateString);

PreparedStatement updateTotal = con.prepareStatement(updateStatement)) {

con.setAutoCommit(false);

for (Map.Entry<String, Integer> e : ventasPorSemana.entrySet()) {

updateVentas.setInt(1, e.getValue().intValue());

updateVentas.setString(2, e.getKey());

updateVentas.executeUpdate(); // Actualización de ventas.

updateTotal.setInt(1, e.getValue().intValue());

updateTotal.setString(2, e.getKey());

updateTotal.executeUpdate(); // Incremento del total.

con.commit();

}

} catch (SQLException e) {

// Gestión de excepciones.

if (con != null) {

try {

System.err.print("La transacción se está revirtiendo");

con.rollback();

} catch (SQLException excep) {

// Gestión de excepciones.

}

}

}

}

[Método entrySet](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/HashMap.html#entrySet())

### 2. Creación de PreparedStatement

Lo siguiente crea un objeto PreparedStatement que toma dos parámetros de entrada:

String updateString = "update Cafe " + "set ventas = ? where nome = ?";

PreparedStatement updateVentas = con.prepareStatement(updateString);

#### 2.1. setTipoDato(columna, valor) de PreparedStatement

PreparedStatement tiene métodos para asignar valores a las ? de la sentencia SQL para cada tipo de dato.  
Por ejemplo:

updateVentas.setInt(1, e.getValue().intValue());

updateVentas.setString(2, e.getKey());

**clearParameters:**

Después de darle un valor a un parámetro se retiene ese valor hasta que se restablece a otro valor o se llama al método clearParameters.

// cambia la columna ventas de Buñuelos

//fila a 100

updateVentas.setInt(1, 100); // Si no se cambia el valor, se mantendrá en 100.

updateVentas.setString(2, "Buñuelos");

updateVentas.executeUpdate();

// cambia la columna ventas de Tortitas americanas a 100

// (el primer parámetro se quedó en 100, y el segundo

// parámetro se restableció a "Tortitas americanas")

updateVentas.setString(2, "Tortitas americanas");

updateVentas.executeUpdate();

**Uso de bucles para asignar valores:**

Se puede facilitar la codificación mediante el uso de un bucle para asignar valores para los parámetros de entrada.

El método updateVentas utiliza un bucle for-each para establecer repetidamente valores en los objetos PreparedStatement updateVentas y updateTotal:

for (Map.Entry<String, Integer> e : ventasPorSemana.entrySet()) {

updateVentas.setInt(1, e.getValue().intValue());

updateVentas.setString(2, e.getKey());

// ...

}

El método updateVentas toma un argumento, HashMap. Cada elemento en el argumento HashMap contiene el nombre y la cantidad vendida durante la semana actual.  
El bucle for-each itera a través de cada elemento del [HashMap](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/HashMap.html).

### 3. Ejecución de sentencias con PreparedStatement: executeUpdate, executeQuery y execute.

Al igual que con los objetos Statement, para ejecutar un objeto PreparedStatement pude invocar:

* **executeQuery** si la consulta devuelve solo un ResultSet (como una declaración SQL SELECT).
* **executeUpdate** si la consulta no devuelve un ResultSet (como una declaración SQL UPDATE o INSERT).
* **execute** si la consulta podría devolver más de un objeto ResultSet.

En updateVentas(HashMap<String, Integer>) son sentencias UPDATE, por lo que usa executeUpdate:

updateVentas.setInt(1, e.getValue().intValue());

updateVentas.setString(2, e.getKey());

updateVentas.executeUpdate();

updateTotal.setInt(1, e.getValue().intValue());

updateTotal.setString(2, e.getKey());

updateTotal.executeUpdate();

con.commit();

Nota: Al principio de updateVentas, el modo de confirmación automática se establece en false:

con.setAutoCommit(false);

En consecuencia, ninguna declaración SQL se confirma hasta que se llama al **método commit**.  
Más adelante veremos cómo realizar transacciones.

### 4. Valores devueltos por executeUpdate

El valor de **devuelto para executeUpdate es un valor int que indica cuántas filas de una tabla se actualizaron**.  
Por ejemplo:

updateVentas.setInt(1, 50);

updateVentas.setString(2, "Tortitas americanas");

int n = updateVentas.executeUpdate();

// n = 1 porque se cambió una fila.

Esa actualización **afecta a una fila en la tabla, por lo que n es igual a 1**.

Cuando el método **executeUpdate se utiliza para ejecutar** **una declaración DDL** (lenguaje de definición de datos), como en la creación de una tabla, **devuelve el valor int de 0**.  
Por ejemplo:

// n = 0

int n = executeUpdate(crearTablaCafe); // Devuelve º filas afectadas.

Cuando el **valor de devuelto por executeUpdate es 0,** puede significar:

1. La declaración ejecutada fue una declaración de actualización que no afectó a ninguna fila.
2. La declaración ejecutada fue una declaración DDL.

# 07. Transacciones

* [Transacciones](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0107transacciones/#transacciones)
  + [1. Desactivación de Auto-Commit](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0107transacciones/#1-desactivacin-de-auto-commit)
  + [2. Commit de transacciones](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0107transacciones/#2-commit-de-transacciones)
  + [3. Puntos de Guardado](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0107transacciones/#3-puntos-de-guardado)
  + [4. Liberación de Puntos de Guardado](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0107transacciones/#4-liberacin-de-puntos-de-guardado)
  + [5. Método rollback:](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0107transacciones/#5-mtodo-rollback-)
  + [6. Utilizando Transacciones en la integridad de los datos](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0107transacciones/#6-utilizando-transacciones-en-la-integridad-de-los-datos)
    - [Nivel de aislamiento de transacción](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0107transacciones/#nivel-de-aislamiento-de-transaccin)

## Transacciones

En muchos casos de uso, es posible que desee ejecutar varias declaraciones SQL como una unidad de trabajo. Por ejemplo, supongamos que tiene una aplicación que actualiza los datos de una tabla y luego actualiza los datos de otra tabla. Desea asegurarse de que ambas actualizaciones se realicen correctamente o que no se realice ninguna de ellas.

La **ejecución de varias declaraciones SQL como una unidad de trabajo se denomina transacción**.

### 1. Desactivación de Auto-Commit

* **Por defecto**, una conexión JDBC está en **modo de auto-commit**.
* Cada **sentencia SQL se trata como una transacción y se confirma automáticamente** justo después de ejecutarse.
* **Para permitir que dos o más sentencias se agrupen en una transacción** se debe **desactivar el modo de auto-commit**.
* **Ninguna sentencia SQL se confirma hasta que se llame explícitamente al método commit**.
* Todas las sentencias ejecutadas después de la llamada al método commit se incluyen en la transacción actual y se confirman juntas como una unidad.

(Para ser más preciso, el valor predeterminado es que una declaración SQL se confirme cuando se completa, no cuando se ejecuta. Sin embargo, en casi todos los casos, una declaración se completa y, por lo tanto, se confirma, justo después de ejecutarse.)

Desactivación de de auto-commit:

con.setAutoCommit(false);

### 2. Commit de transacciones

Ninguna declaración SQL se confirma hasta que se llame al método commit:

public void updateVentas(HashMap<String, Integer> ventasPorSemana) throws SQLException {

try (PreparedStatement psVentas = con.prepareStatement("update Producto set ventas = ? where nome = ?");

PreparedStatement psTotal = con.prepareStatement("update Producto set total = total + ? where nome = ?")) {

con.setAutoCommit(false); // Deshabilita el modo de autocommit

for (Map.Entry<String, Integer> e : ventasPorSemana.entrySet()) {

psVentas.setInt(1, e.getValue().intValue());

psVentas.setString(2, e.getKey());

psVentas.executeUpdate();

psTotal.setInt(1, e.getValue().intValue());

psTotal.setString(2, e.getKey());

psTotal.executeUpdate();

con.commit(); // Confirmación

}

} catch (SQLException e) {

// Gestión de excepciones.

if (con != null) {

try {

System.err.print("La transacción se está revirtiendo");

con.rollback(); // Revierte la transacción

} catch (SQLException excep) {

// Gestión de excepciones.

}

}

}

}

* Las dos declaraciones preparadas psVentas y psTotal se confirman juntas cuando se llama al método commit().
* Cada vez que se llama al método commit (ya sea automáticamente cuando se habilita el modo de auto-commit o explícitamente cuando se deshabilita), todos los cambios se vuelven permanentes.
* La declaración con.setAutoCommit(true); habilita el modo de auto-commit, cada declaración se confirma automáticamente cuando se completa.

 Desactivación y activación de auto-commit

**Es recomendable desactivar el modo de auto-commit únicamente durante el modo de transacción**.  
De esta manera, se **evita mantener bloqueos de base de datos para múltiples declaraciones**, lo que aumenta la probabilidad de conflictos con otros usuarios.

### 3. Puntos de Guardado

El método Connection.setSavepoint establece un objeto Savepoint (punto de guardado) dentro de la transacción actual.  
El método Connection.rollback se sobrecarga para aceptar un argumento Savepoint, un punto de guardado dentro de la transacción actual.

El siguiente método, Producto.modificarPreciosPorPoncertaje, aumenta el precio de un café en particular por un porcentaje, porcentaje. Sin embargo, si el nuevo precio es mayor que un precio especificado, precioMaximo, entonces el precio se revierte al precio original:

public void modificarPreciosPorPoncertaje(String nombre, float porcentaje, float precioMaximo)

throws SQLException {

con.setAutoCommit(false);

ResultSet rs = null;

String precioQuery = "SELECT nombre, precio FROM Producto WHERE nombre = ?";

String updateQuery = "UPDATE Producto SET precio = ? WHERE nombre = ?";

try (PreparedStatement psPrecio = con.prepareStatement(precioQuery, ResultSet.TYPE\_SCROLL\_INSENSITIVE, ResultSet.CONCUR\_READ\_ONLY);

PreparedStatement updatePrice = con.prepareStatement(updateQuery)) {

Savepoint puntoSalvar = con.setSavepoint(); // Creación de punto de guardado

psPrecio.setString(1, nombre);

if (!psPrecio.execute()) { // Si no hay resultados

System.out.println("No puedo encontrar el producto con nombre: " + nombre);

} else {

rs = psPrecio.getResultSet();

rs.first(); // sitúa el cursor en la primera fila

float precioAnterior = rs.getFloat("precio");

float precioNuevo = precioAnterior + (precioAnterior \* porcentaje);

System.out.printf("Precio anterio de %s es $%.2f%n", nombre, precioAnterior);

System.out.printf("Nuevo precio de %s es $%.2f%n", nombre, precioNuevo);

System.out.println("Realizando actualización...");

updatePrice.setFloat(1, precioNuevo);

updatePrice.setString(2, nombre);

updatePrice.executeUpdate();

System.out.println("\nProducto después de actualización:");

Producto.verTabla(con); // Ver tabla (debe implantarse)

if (precioNuevo > precioMaximo) { // Si supera el máximo se hace un rollback al punto de guardado

System.out.printf("El nuevo precio, $%.2f, es mayor que el precio máximo, $%.2f. " +

"Revertiendo la transacción...%n", precioNuevo, precioMaximo);

con.rollback(puntoSalvar);

System.out.println("\nProducto después de revertir:");

Producto.viewTable(con); // Ver tabla (debe implantarse)

}

con.commit(); // Commit de la transacción

}

} catch (SQLException e) {

// Gestión de excepciones.

} finally {

con.setAutoCommit(true);

}

}

Se especifica que el cursor ResultSet generado por psPrecio se cierra cuando se llama al método commit. Ten en cuenta que si el SGBD no admite ResultSet.CLOSE\_CURSORS\_AT\_COMMIT, ésta se ignora:

psPrecio = con.prepareStatement(consulta, ResultSet.CLOSE\_CURSORS\_AT\_COMMIT);

El método comienza creando un Savepoint con la siguiente instrucción:

Savepoint puntoSalvar = con.setSavepoint();

El método verifica si el nuevo precio es mayor que el valor de precioMaximo. Si es así, el método deshace la transacción hasta el punto de guardado con la siguiente instrucción:

con.rollback(puntoSalvar);

Cuando el método realiza la transacción llamando al método Connection.commit, no comprometerá ninguna fila cuyo Savepoint asociado haya sido deshecho; comprometerá todas las demás filas actualizadas.

### 4. Liberación de Puntos de Guardado

El método Connection.releaseSavepoint toma un objeto Savepoint como parámetro y lo elimina de la transacción actual.

Después de que se ha liberado un punto de guardado, intentar hacer referencia en una operación de deshacer provoca que se lance una SQLException.

Cualquier punto de guardado que se haya creado en una transacción se libera automáticamente y se vuelve inválido cuando la transacción se confirma o cuando se revierte por completo la transacción.  
Deshacer una transacción hasta un punto de guardado libera automáticamente y vuelve inválidos cualquier otro punto de guardado que se haya creado después del punto de guardado en cuestión.

### 5. Método rollback:

* Llamar al método **rollback termina una transacción y devuelve los valores que se modificaron a sus valores anteriores**.
* Si se intenta ejecutar una o más declaraciones en una transacción y se obtiene un SQLException, debe invocarse al método rollback para finalizar la transacción y comenzarla de nuevo.

Capturar un **SQLException indica que hay errores, pero no te dice qué se ha comprometido o no**.  
Debido a que no no se puede saber qsi se ha completado alguna sentencia, llamar al método rollback es la única forma de estar seguro.

El método Producto.updateVentas demuestra una transacción e incluye un bloque catch que invoca al método rollback. Si la aplicación continúa y utiliza los resultados de la transacción, esta llamada al método rollback en el bloque catch evita el uso de datos posiblemente incorrectos.

### 6. Utilizando Transacciones en la integridad de los datos

* Las **transacciones** pueden ayudar a **preservar la integridad de los datos** en una tabla.
* El uso de transacciones proporciona algún nivel de **protección contra conflictos** que surgen cuando **dos usuarios acceden a datos al mismo tiempo**.
* Para **evitar conflictos** durante una transacción, **un SGBD utiliza bloqueos**:
  + Mecanismo para **bloquear el acceso de otros a los datos que está siendo accedido por la transacción**. (Ten en cuenta que en el modo de *autocommit*, donde cada declaración es una transacción, los bloqueos se mantienen solo para una declaración).
* Un **bloqueo permanece vigente hasta que la transacción se confirma o se revierte**.

 Bloqueos

**Los bloqueos pueden causar problemas de rendimiento**.

Por ejemplo, un DBMS podría bloquear una fila de una tabla hasta que las actualizaciones en ella se hayan confirmado. El efecto de este bloqueo sería evitar que un usuario obtenga una lectura sucia, es decir, leer un valor antes de que se haga permanente. (Acceder a un valor actualizado que no se ha confirmado se considera una lectura sucia porque es posible que ese valor se revierta a su valor anterior. Si lees un valor que luego se revierte, habrás leído un valor no válido.)

#### Nivel de aislamiento de transacción

**Cómo se establecen los bloqueos** está determinado por lo que se llama un **nivel de aislamiento de transacción**, que puede variar desde no admitir transacciones en absoluto hasta admitir transacciones que imponen reglas de acceso muy estrictas.

Un ejemplo de un nivel de aislamiento de transacción es TRANSACTION\_READ\_COMMITTED, que no permitirá que se acceda a un valor hasta después de que se haya confirmado.  
En otras palabras, si el nivel de aislamiento de la transacción se establece en TRANSACTION\_READ\_COMMITTED, el DBMS no permite lecturas sucias.

La interfaz **Connection incluye cinco valores que representan los niveles de aislamiento de transacción**:

| **Nivel de Aislamiento** | **Transacciones** | **Lecturas Sucias** | **Lecturas No Repetibles** | **Lecturas Fantasmales** |
| --- | --- | --- | --- | --- |
| TRANSACTION\_NONE | No admitido | No aplicable | No aplicable | No aplicable |
| TRANSACTION\_READ\_COMMITTED | Admitido | Prevenido | Permitido | Permitido |
| TRANSACTION\_READ\_UNCOMMITTED | Admitido | Permitido | Permitido | Permitido |
| TRANSACTION\_REPEATABLE\_READ | Admitido | Prevenido | Prevenido | Permitido |
| TRANSACTION\_SERIALIZABLE | Admitido | Prevenido | Prevenido | Prevenido |

* Una **lectura no repetible** ocurre cuando la transacción A recupera una fila, la transacción B actualiza posteriormente la fila, y la transacción A vuelve a recuperar la misma fila. **La transacción A recupera la misma fila dos veces pero ve datos diferentes**.
* Una **lectura fantasma** ocurre cuando la transacción A recupera un conjunto de filas que cumplen con una condición dada, la transacción B inserta o actualiza posteriormente una fila de manera que ahora cumple con la condición en la transacción A, y la transacción A repite más tarde la recuperación condicional. La transacción A ahora ve una fila adicional. A esta fila se le denomina fantasma.

No necesitas hacer nada respecto al **nivel de aislamiento de la transacción**; puedes **usar el predeterminado para el SGBD empleado**.  
El **nivel de aislamiento de transacción predeterminado depende del SGBD**. Por ejemplo, para Java DB, es TRANSACTION\_READ\_COMMITTED. JDBC te permite averiguar a qué nivel de aislamiento de transacción está configurado tu DBMS (usando el método getTransactionIsolation de Connection) y también te permite establecerlo en otro nivel (usando el método setTransactionIsolation de Connection).

 Niveles de aislamiento de transacción

Nota: **es muy probable que un controlador JDBC no admita todos los niveles de aislamiento de transacción**.  
Si un controlador no admite el nivel de aislamiento especificado en una invocación de setTransactionIsolation, el controlador puede sustituir un nivel de aislamiento de transacción más alto y restrictivo.  
Si un controlador no puede sustituir un nivel de transacción más alto, se produce una SQLException.

Utiliza el método DatabaseMetaData.supportsTransactionIsolationLevel para determinar si el controlador admite o no un nivel dado.

# 08. Claves Generadas

* [1. Introducción](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0108clavesgeneradas/#1-introduccin)
* [2. Configuración y creación de la tabla](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0108clavesgeneradas/#2-configuracin-y-creacin-de-la-tabla)
* [3. Statement.RETURN\_GENERATED\_KEYS y getGeneratedKeys()](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0108clavesgeneradas/#3-statementreturn_generated_keys-y-getgeneratedkeys--)
* [4. Devolver Columnas Específicas](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0108clavesgeneradas/#4-devolver-columnas-especficas)

## 1. Introducción

Muchas veces necesitamos **obtener el valor de una clave primaria generada automáticamente después de insertar un registro en la base de datos**. Mediante JDBC podemos obtener el valor de la clave primaria generada automáticamente después de insertar un registro en la base de datos.

## 2. Configuración y creación de la tabla

A m odo de ejemplo, poder ejecutar consultas SQL, utilizaremos una base de datos H2 en memoria:

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

<version>2.1.214</version>

</dependency>

También utilizaremos una tabla muy sencilla con dos columnas:

public class JdbcInsertId {

private static Connection connection;

public static void inicio() throws Exception {

connection = DriverManager.getConnection("jdbc:h2:mem:dbConClaves", "sa", "");

connection.createStatement().execute("create table Persona (id bigint auto\_increment, nome varchar(255))");

}

public static void borrado() throws SQLException {

connection.createStatement()

.execute("drop table Persona");

connection.close();

}

// ...

}

Que se conecta a la base de datos en memoria “dbConClaves” y crea una tabla llamada “Persona”.

## 3. Statement.RETURN\_GENERATED\_KEYS y getGeneratedKeys()

Una forma de obtener las claves después de la generación automática es pasar Statement.RETURN\_GENERATED\_KEYS al método prepareStatement():

String QUERY = "insert into Persona (nome) values (?)";

try (PreparedStatement statement = conexion.prepareStatement(QUERY, Statement.RETURN\_GENERATED\_KEYS)) {

statement.setString(1, "Otto");

int filasInsertadas = statement.executeUpdate();

if (filasInsertadas > 0) {

// ...

} else {

// ...

}

// ...

} catch (SQLException e) {

// manejar la excepción relacionada con la base de datos de manera apropiada

}

Después de preparar y ejecutar la consulta, se **puede llamar al método getGeneratedKeys() en PreparedStatement para obtener el id**:

try (ResultSet claves = statement.getGeneratedKeys()) {

if(claves.next()) {

// ...

long id = claves.getLong(1);

} else {

// ...

}

}

Que llama al método next() para mover el cursor del resultado con las claves generadas.  
El método getLong() obtiene obtener la primera columna como long.

Además, también es posible utilizar la misma técnica con Statements normales:

try (Statement statement = conexion.createStatement()) {

String query = "insert into Persona (nome) values ('Otto')";

int filasAfectadas = statement.executeUpdate(query, Statement.RETURN\_GENERATED\_KEYS);

if(filasAfectadas > 0) {

// ...

} else {

// ...

}

try (ResultSet claves = statement.getGeneratedKeys()) {

if(claves.next()) {

// ...

long id = claves.getLong(1);

// ...

} else {

// ...

}

}

}

Debe emplearse try-with-resources de manera extensiva para permitir que el compilador limpie los resultados.

## 4. Devolver Columnas Específicas

Podemos hacer que **devuelva columnas específicas después de emitir una consulta**. Para hacer eso, solo tenemos que pasar un array de nombres de columna:

try (PreparedStatement statement = conexion.prepareStatement(QUERY, new String[] { "id" })) {

statement.setString(1, "Otto");

int filasAfectadas = statement.executeUpdate();

if (filasAfectadas > 0) {

// ...

} else {

// ...

}

// ...

}

En el ejemplo anterior devuelve el valor de la columna id después de ejecutar la consulta dada.

Similar al ejemplo anterior, podemos obtener el id después:

try (ResultSet claves = statement.getGeneratedKeys()) {

if(claves.next()) {

// ...

long id = claves.getLong(1);

} else {

// ...

}

}

Podemos utilizar el mismo enfoque con Statements simples:

try (Statement statement = conexion.createStatement()) {

int filasAfectadas = statement.executeUpdate("insert into Persona (nome) values ('Otto')", new String[] { "id" });

if(filasAfectadas > 0) {

// ...

} else {

// ...

}

try (ResultSet claves = statement.getGeneratedKeys()) {

if(claves.next()) {

// ...

long id = claves.getLong(1);

// ...

} else {

// ...

}

}

}

# 09. Objectos grandes

* [1. Introducción](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0109largeobjects/#1-introducci%C3%B3n)
* [1. Uso de LOB (Objetos de Gran Tamaño)](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0109largeobjects/#1-uso-de-lob-objetos-de-gran-tama%C3%B1o)
  + [1. Añadir un CLOB la Base de Datos](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0109largeobjects/#1-a%C3%B1adir-un-clob-la-base-de-datos)
  + [2. Recuperando valores CLOB](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0109largeobjects/#2-recuperando-valores-clob)
  + [3. Agregando y recuperado Objetos BLOB](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0109largeobjects/#3-agregando-y-recuperado-objetos-blob)
    - [setBlob de PreparedStatement](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0109largeobjects/#setblob-de-preparedstatement)
    - [setBinaryStream de PreparedStatement](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0109largeobjects/#setbinarystream-de-preparedstatement)
    - [setBytes de PreparedStatement](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0109largeobjects/#setbytes-de-preparedstatement)
  + [4. Liberando Recursos Retenidos por Objetos Grandes](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0109largeobjects/#4-liberando-recursos-retenidos-por-objetos-grandes)

## 1. Introducción

Los **objetos grandes (LOB, Large Objects)** son **objetos de datos** que pueden tener un tamaño variable y que se almacenan en una base de datos. Se utilizan para almacenar datos como **imágenes, sonidos, videos y documentos de texto**.

Por lo general, las bases de datos almacenan los datos de la siguiente forma: las columnas se agrupan en filas que, a su vez, se apilan en bloques de datos. La información en cada bloque de datos está asociada a una fila y los bloques de datos consumen así menos espacio en la base de datos.

* Las bases de datos **tratan de otro modo los objetos de datos de mayor tamaño**. Los LOB superan en tamaño a las entradas convencionales de las bases de datos y no se encuentran estructurados.
* **En la mayoría de los casos, se almacenan en un lugar distinto**.  
  La base de datos sólo crea en la posición que corresponda una referencia a la ubicación de almacenamiento.

Existen dos tipos de LOB:

* **BLOB**: un BLOB es un tipo de dato que almacena un elemento grande de datos **en código binario**.
* **CLOB**: un CLOB (***Character Large Objects***) almacena **cadenas largas de caracteres**. Es un término acuñado por los desarrolladores de la base de datos de Oracle.

Nota: otros sistemas de gestión de bases de datos utilizan también otros términos para denominar los objetos grandes: MySQL/MariaDB y PostgreSQL los denominan ***TEXT***.

Tipos de datos para almacenar objetos LOB (Large Objects) tanto binarios como de texto en diferentes Sistemas de Gestión de Bases de Datos (SGBDR), junto con sus tamaños típicos:

1. **MariaDB:**
   * **Binario (BLOB):** BLOB o LONGBLOB (hasta 4 GB).
   * **Texto (CLOB):** TEXT o LONGTEXT (hasta 4 GB).
2. **H2:**
   * **Binario (BLOB):** BLOB (64TB)
   * **Texto (CLOB):** CLOB (64TB).
3. **SQLite:**
   * **Binario (BLOB):** No hay un tipo específico para BLOB, se pueden usar tipos de datos TEXT o BLOB (hasta 2 GB).
   * **Texto (CLOB):** No hay un tipo específico para CLOB, se pueden usar tipos de datos TEXT o BLOB (hasta 2 GB).
4. **PostgreSQL:**
   * **Binario (BLOB):** BYTEA o OID (hasta 1 GB).
   * **Texto (CLOB):** TEXT o VARCHAR (sin límite declarado, prácticamente limitado por el tamaño de la tabla).
5. **Oracle:**
   * **Binario (BLOB):** BLOB (hasta 128 TB).
   * **Texto (CLOB):** CLOB (hasta 128 TB).
6. **MS SQL Server:**
   * **Binario (BLOB):** VARBINARY(MAX) o IMAGE (hasta 2 GB en VARBINARY(MAX) y hasta 4 GB en IMAGE).
   * **Texto (CLOB):** VARCHAR(MAX) o TEXT (hasta 2 GB en VARCHAR(MAX) y hasta 2 GB en TEXT).

MariaDB y MySQL tienen cuatro tipos de datos de texto y LOB:

* **TINYTEXT**: un texto de longitud máxima de 255 caracteres.
* **TEXT**: un texto de longitud máxima de 65.535 caracteres (64KB).
* **MEDIUMTEXT**: un texto de longitud máxima de 16.777.215 caracteres.
* **LONGTEXT**: un texto de longitud máxima de 4.294.967.295 caracteres (4GB).
* **BLOB**: un BLOB es un tipo de dato que almacena un elemento grande de datos **en código binario** (6GB).
* **MEDIUMBLOB**: un BLOB de longitud máxima de 16.777.215 bytes.
* **LONGBLOB**: un BLOB de longitud máxima de 4.294.967.295 bytes. (4GB).
* **TINYBLOB**: un BLOB de longitud máxima de 255 bytes.

H2 tiene dos tipos de datos de texto:

* **CLOB/CHARACTER LARGE OBJECT**: un texto de longitud máxima de 2GB.
* **BLOB/BINARY LARGE OBJECT**: un BLOB es un tipo de dato que almacena un elemento grande de datos **en código binario**.
* **BINARY**: un BLOB de longitud máxima de 2GB.
* **VARBINARY**: un BLOB de longitud máxima de 2GB.
* **LONGVARBINARY**: un BLOB de longitud máxima de 2GB.

## 1. Uso de LOB (Objetos de Gran Tamaño)

Los **objetos grandes Java, como Blob, Clob y NClob** pueden gestionarse desde Java **sin tener que traer los datos del servidor de la base de datos al cliente**.

Muchas implementaciones representan una instancia de estos tipos de datos con un localizador **(puntero) al objeto en la base de datos**.

Debido a que un objeto **BLOB, CLOB o NCLOB puede ser muy grande**, el **uso de punteros mejora el rendimiento**. Sin embargo, algunas implementaciones gestionan (y cargan) completamente objetos grandes en cliente.

Para traer un BLOB, CLOB o NCLOB de SQL al programa cliente, se emplean métodos en las **interfaces de Java**[**Blob**](https://docs.oracle.com/en/java/javase/21/docs/api/java.sql/java/sql/Blob.html)**,**[**Clob**](https://docs.oracle.com/en/java/javase/21/docs/api/java.sql/java/sql/Clob.html)**y**[**NClob**](https://docs.oracle.com/en/java/javase/21/docs/api/java.sql/java/sql/NClob.html).

### 1. Añadir un CLOB la Base de Datos

La [interface PreparedStatement](https://docs.oracle.com/en/java/javase/23/docs/api/java.sql/java/sql/PreparedStatement.html) tiene métodos para asignar valores a las ? de la sentencia SQL para cada tipo de dato. Para CLob se utiliza el método [setClob](https://docs.oracle.com/en/java/javase/21/docs/api/java.sql/java/sql/PreparedStatement.html#setClob(int,java.io.Reader)):

void setClob(int parameterIndex, Reader reader);

void setClob(int parameterIndex, Reader reader, long length);

void setClob(int parameterIndex, Clob x);

// o

void setCharacterStream(int parameterIndex, Reader reader);

void setCharacterStream(int parameterIndex, Reader reader, int length);

void setCharacterStream(int parameterIndex, Reader reader, long length);

//Insertar valores

PreparedStatement pstmt = con.prepareStatement("INSERT INTO Table(nombre, descripcion) VALUES (?, ?)");

pstmt.setString(1, "nombre de ejemplo");

pstmt.setClob(2, Files.newBufferedReader(Paths.get("E:\\descripcion.txt")));

pstmt.executeUpdate();

**Clob:**

El siguiente extracto de addDescripcionProducto agrega un valor SQL CLOB a la tabla Producto. El objeto Java Clob clobDescripcion contiene el contenido del archivo especificado por nomeArquivo.

public void addDescripcionProducto(String nome, String nomeArquivo) throws SQLException {

// Cfreación del objeto Clob:

Clob clobDescripcion = this.con.createClob();

try (PreparedStatement pstmt = this.con.prepareStatement("INSERT INTO Producto VALUES(?,?)");

Writer clobWriter = clobDescripcion.setCharacterStream(1);){

// setCharacterStream devuelve un objeto Writer y recibe un entero que indica la posición inicial del Clob.

String str = this.readFile(nomeArquivo, clobWriter); // Lee el conteido del archivo.

System.out.println("Escribo el texto: " + clobWriter.toString());

// Si el archivo es demasiado grande, se puede escribir en el Clob en trozos.

clobDescripcion.setString(1, str);

System.out.println("Longitud del clob: " + clobDescripcion.length());

pstmt.setString(1, nome);

pstmt.setClob(2, clobDescripcion); // Se añade el Clob al PreparedStatement.

pstmt.executeUpdate();

} catch (SQLException sqlex) {

// Gestión de excepciones.

} catch (Exception ex) {

System.out.println("Excepción no esperada: " + ex.toString());

}

}

private String readFile(String nomeArquivo, Writer writer) throws IOException {

try (BufferedReader br = new BufferedReader(new FileReader(nomeArquivo))) {

String nextLine = "";

StringBuffer sb = new StringBuffer();

while ((nextLine = br.readLine()) != null) {

System.out.println("Escribiendo: " + nextLine);

writer.write(nextLine);

sb.append(nextLine);

}

// Convertir el contenido en una cadena

String datosClob = sb.toString();

// devolución de los datos.

return datosClob;

}

}

a) **Creación de un objeto Clob**:

Clob clobDescripcion = this.con.createClob();

b) **Recuperación del flujo** (en este caso, un objeto Writer llamado clobWriter) que se utiliza para escribir un flujo de caracteres en el objeto Java Clob clobDescripcion. El método readFile escribe este flujo de caracteres; el flujo proviene del archivo especificado por la cadena nomeArquivo. El argumento del método 1 indica que el objeto Writer comenzará a escribir el flujo de caracteres al principio del valor Clob:

Writer clobWriter = clobDescripcion.setCharacterStream(1);

El método readFile lee el archivo línea por línea especificado por el archivo nomeArquivo y lo escribe en el objeto Writer especificado por writer:

private String readFile(String nomeArquivo, Writer writer) throws IOException {

try (BufferedReader br = new BufferedReader(new FileReader(nomeArquivo))) {

String nextLine = "";

StringBuffer sb = new StringBuffer();

while ((nextLine = br.readLine()) != null) {

System.out.println("Escribiendo: " + nextLine);

writer.write(nextLine);

sb.append(nextLine);

}

// Convertir el contenido en una cadena

String clobData = sb.toString();

// Devolver los datos.

return clobData;

}

}

c) Creación de un objeto PreparedStatement pstmt que inserta el objeto Java Clob clobDescripcion en Producto:

String sql = "INSERT INTO Producto VALUES(?,?)";

Clob clobDescripcion = this.con.createClob();

try (PreparedStatement pstmt = this.con.prepareStatement(sql);

// ...

) {

// ...

pstmt.setString(1, nome);

pstmt.setClob(2, clobDescripcion);

pstmt.executeUpdate();

// ...

}

### 2. Recuperando valores CLOB

El método getDescripcion recupera el valor SQL CLOB almacenado en la columna descripcion de Producto de la fila cuyo valor de columna nome es igual al valor de la cadena especificada por el parámetro nome:

public String getDescripcion(String nome, int numeroCaracteres) throws SQLException {

String descripcion = null;

Clob clobDescripcion = null;

String sql = "select descripcion from Producto where nome = ?";

try (PreparedStatement pstmt = this.con.prepareStatement(sql)) {

pstmt.setString(1, nome);

ResultSet rs = pstmt.executeQuery();

if (rs.next()) {

clobDescripcion = rs.getClob(1);

System.out.println("Lonxitude do Clob: " + clobDescripcion.length());

}

descripcion = clobDescripcion.getSubString(1, numeroCaracteres);

} catch (SQLException sqlex) {

// Tratamiento de excepciones.

} catch (Exception ex) {

System.out.println("Excepción: " + ex.toString());

}

return descripcion;

}

Recupera el valor Java Clob del objeto ResultSet rs:

clobDescripcion = rs.getClob(1);

Recuperación de una subcadena del objeto clobDescripcion.  
La subcadena comienza en el primer carácter del valor de clobDescripcion y tiene hasta el número de caracteres consecutivos especificados en numeroCaracteres, donde numeroCaracteres es un entero.

descripcion = clobDescripcion.getSubString(1, numeroCaracteres);

### 3. Agregando y recuperado Objetos BLOB

#### setBlob de PreparedStatement

PreparedStatement tiene métodos para asignar valores a las ? de la sentencia SQL para cada tipo de dato.

void setBlob(int parameterIndex, InputStream inputStream)

void setBlob(int parameterIndex, InputStream inputStream, long length)

void setBlob(int parameterIndex, Blob x)

// o

void setBinaryStream(int parameterIndex, InputStream x);

void setBinaryStream(int parameterIndex, InputStream x, int length);

void setBinaryStream(int parameterIndex, InputStream x, long length);

Vamos a verlo con el método setBlob que recoge un objeto Blob.

Agregar y recuperar objetos SQL BLOB es similar a agregar y recuperar objetos CLOB.  
Se precisa crear un blob, para ello se utiliza el método [createBlob](https://docs.oracle.com/en/java/javase/21/docs/api/java.sql/java/sql/Connection.html#createBlob--).

Y el método [el método Blob.setBinaryStream](https://docs.oracle.com/en/java/javase/21/docs/api/java.sql/java/sql/Blob.html#setBinaryStream(long)) para **recuperar un objeto OutputStream para escribir el valor BLOB**.

El siguiente extracto de addImagenProducto agrega un valor SQL BLOB a la tabla Producto. El objeto Java Blob blobImagen contiene el contenido del archivo especificado por nomeArquivo.

public void addImagenProducto(String nome, String nomeArquivo) throws SQLException {

// Creación del objeto Blob:

Blob blobImagen = this.con.createBlob();

try (PreparedStatement pstmt = this.con.prepareStatement("INSERT INTO Producto VALUES(?,?)");

OutputStream blobOutputStream = blobImagen.setBinaryStream(1);){

// setBinaryStream devuelve un objeto OutputStream y recibe un entero que indica la posición inicial del Blob.

byte[] bytes = this.readFile(nomeArquivo, blobOutputStream); // Lee el conteido del archivo.

System.out.println("Escribo el texto: " + blobOutputStream.toString());

// Si el archivo es demasiado grande, se puede escribir en el Blob en trozos.

blobImagen.setBytes(1, bytes);

System.out.println("Longitud del blob: " + blobImagen.length());

pstmt.setString(1, nome);

pstmt.setBlob(2, blobImagen); // Se añade el Blob al PreparedStatement.

pstmt.executeUpdate();

} catch (SQLException sqlex) {

// Gestión de excepciones.

} catch (Exception ex) {

System.out.println("Excepción no esperada: " + ex.toString());

}

}

#### setBinaryStream de PreparedStatement

Otro modo de realizarlo es por medio del método [setBinaryStream](https://docs.oracle.com/en/java/javase/21/docs/api/java.sql/java/sql/PreparedStatement.html#setBinaryStream(int,java.io.InputStream)):

Este método esta sobrecargado y se puede utilizar de varias formas:

public void setBinaryStream(int index, InputStream is)

public void setBinaryStream(int index, InputStream is, int length)

public void setBinaryStream(int index, InputStream is, long length)

Por ejemplo:

//Insertar valores

PreparedStatement pstmt = con.prepareStatement("INSERT INTO Table(nombre, imagen) VALUES (?, ?)");

pstmt.setString(1, "imagen de ejemplo");

FileInputStream fin = new FileInputStream("E:\imagenes\otto.jpg");

pstmt.setBinaryStream(2, fin);

pstmt.execute();

#### setBytes de PreparedStatement

Otra forma de realizarlo es por medio del método [setBytes](https://docs.oracle.com/en/java/javase/21/docs/api/java.sql/java/sql/PreparedStatement.html#setBytes(int,byte%5B%5D)):

void setByte(int parameterIndex, byte x)

void setBytes(int parameterIndex, byte[] x)

Ejemplo:

//Insertar valores

PreparedStatement pstmt = con.prepareStatement("INSERT INTO Table(nombre, imagen) VALUES (?, ?)");

pstmt.setString(1, "imagen de ejemplo");

pstmt.setBytes(2, Files.readAllBytes(Paths.get("E:\imagenes\otto.jpg")));

pstmt.execute();

### 4. Liberando Recursos Retenidos por Objetos Grandes

Los objetos Java Blob, Clob y NClob siguen siendo válidos durante al menos la duración de la transacción en la que se crearon. Esto podría resultar en que una aplicación se quede sin recursos durante una transacción de larga duración. Las aplicaciones pueden **liberar los recursos de Blob, Clob y NClob invocando su método free**:

Clob aClob = con.createClob();

int numWritten = aClob.setString(1, val);

aClob.free();

# 11. RowSet

* [Introducción](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0111rowset/#introducci%C3%B3n)
  + [1. La interface RowSet](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0111rowset/#1-la-interface-rowset)
  + [2. Ventajas de RowSet](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0111rowset/#2-ventajas-de-rowset)
  + [3. La interfaz JdbcRowSet](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0111rowset/#3-la-interfaz-jdbcrowset)
    - [Implementación](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0111rowset/#implementaci%C3%B3n)
    - [Métodos principales de la interfaz JdbcRowSet que no heredan de RowSet](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0111rowset/#m%C3%A9todos-principales-de-la-interfaz-jdbcrowset-que-no-heredan-de-rowset)

## Introducción

Un objeto JDBC RowSet **hereda de la interface ResultSet**, almacenando **datos en forma de tabla** de una manera que lo hace más flexible y fácil de usar que un ResultSet.

El API ha definido **5 interfaces RowSet para algunos de los usos más comunes de un RowSet** con implementaciones de referencia estándar disponibles para estas interfaces RowSet. Las implementaciones de referencia estándar son **JdbcRowSet, CachedRowSet, , WebRowSet, JoinRowSet y FilteredRowSet**.

* **JdbcRowSet** es un objeto RowSet que proporciona **conectividad de base de datos a través de una conexión JDBC**.
* **CachedRowSet** es un objeto RowSet que almacena datos en caché en la **memoria del cliente**.
* **WebRowSet** es un objeto RowSet que proporciona **conectividad de base de datos a través de una conexión JDBC y puede escribirse en un flujo de datos XML**.
* **JoinRowSet** es un objeto RowSet que proporciona conectividad de base de datos a través de una conexión JDBC y puede **unirse a datos de varias fuentes de datos**.
* **FilteredRowSet** es un objeto RowSet que proporciona conectividad de base de datos a través de una conexión JDBC y **puede filtrar filas**.
* Es posible escribir las implementaciones personalizadas de [la interface javax.sql.RowSet](https://docs.oracle.com/en/java/javase/21/docs/api/java.sql/javax/sql/RowSet.html), o heredar de las implementaciones de las cinco interfaces RowSet.
* Para la mayoría de los casos **las implementaciones de referencia estándar cubren todas las necesidades**.

### 1. La interface RowSet

[RowSet es una interfaz](https://docs.oracle.com/en/java/javase/21/docs/api/java.sql/javax/sql/RowSet.html) en Java que se encuentra en el módulo java.sql (no confundir RowSet con ResultSet).

RowSet está en paquete javax.sql, mientras que ResultSet está en java.sql.

La instancia de RowSet es el componente JavaBean porque tiene propiedades y un mecanismo de notificación de JavaBean.  
Se **introdujo en JDK5**. Un JDBC RowSet **proporciona una forma de almacenar los datos en forma tabular**.  
Hace que los datos sean más flexibles y más fáciles que un ResultSet.  
La **conexión entre el objeto RowSet y la fuente de datos se mantiene durante todo su ciclo de vida**.

COmo hemos comentado, RowSets se clasifican en cinco categorías según cómo estén implementados, que se enumeran a continuación:

* JdbcRowSet
* CachedRowSet
* WebRowSet
* FilteredRowSet
* JoinRowSet

### 2. Ventajas de RowSet

* Es **fácil y flexible de usar**.
* **Por defecto, es desplazable y puede actualizarse**, mientras que ResultSet, por defecto, solo es válido para operaciones hacia adelante y de solo lectura.
* Es un **componente JavaBean** (tiene **propiedades y eventos**), por lo que es fácil de usar en cualquier entorno y herramienta de desarrollo, además de ser compatible con la notificación de JavaBean:
  + Las propiedades de un objeto RowSet se pueden establecer y obtener mediante los métodos set y get y se ven en lo IDEs.
  + Los eventos de un objeto RowSet se pueden registrar y recibir mediante los métodos add y remove y se ven en lo IDEs:
  + Los eventos se producen cuando se mueve el cursor, se actualiza una fila, se elimina una fila, se inserta una fila, etc.
    - RowSetEvent se genera cuando se produce un cambio en el objeto RowSet.
    - RowSetMetaDataEvent se genera cuando se produce un cambio en el objeto RowSetMetaData.
    - RowSetWarningEvent se genera cuando se produce un cambio en el objeto RowSetWarning.
* **Puede ser serializado**.
* La **interfaz JDBC RowSet hereda de RowSet**. Es un contenedor para el objeto **ResultSet que añade características**.

Métodos comunes a todas las implementaciones de RowSet:

* setUrl(String url): [establece la URL](https://docs.oracle.com/en/java/javase/21/docs/api/java.sql/javax/sql/RowSet.html#setUrl(java.lang.String)) de la base de datos a la que se conectará el objeto RowSet.
* setUsername(String name): [establece el nombre de usuario](https://docs.oracle.com/en/java/javase/21/docs/api/java.sql/javax/sql/RowSet.html#setUsername(java.lang.String)) que se utilizará para conectarse a la base de datos.
* setPassword(String password): [establece la contraseña](https://docs.oracle.com/en/java/javase/21/docs/api/java.sql/javax/sql/RowSet.html#setPassword(java.lang.String)) que se utilizará para conectarse a la base de datos.
* setCommand(String cmd): [establece el comando SQL](https://docs.oracle.com/en/java/javase/21/docs/api/java.sql/javax/sql/RowSet.html#setCommand(java.lang.String)) que se utilizará para obtener los datos de la base de datos.
* setInt(int parameterIndex, int x): [establece el valor de un parámetro](https://docs.oracle.com/en/java/javase/21/docs/api/java.sql/javax/sql/RowSet.html#setInt(int,int)) de tipo int en la consulta SQL que se utilizará para obtener los datos de la base de datos.
* setString(int parameterIndex, String x): [establece el valor de un parámetro](https://docs.oracle.com/en/java/javase/21/docs/api/java.sql/javax/sql/RowSet.html#setString(int,java.lang.String)) de tipo String en la consulta SQL que se utilizará para obtener los datos de la base de datos.
* …
* execute(): [ejecuta el comando SQL](https://docs.oracle.com/en/java/javase/21/docs/api/java.sql/javax/sql/RowSet.html#execute--) que se ha establecido con el método setCommand.
* next(): [mueve el cursor](https://docs.oracle.com/en/java/javase/21/docs/api/java.sql/javax/sql/RowSet.html#next--) a la siguiente fila de la tabla de resultados.
* getInt(int columnIndex): [obtiene el valor](https://docs.oracle.com/en/java/javase/21/docs/api/java.sql/javax/sql/RowSet.html#getInt(int)) de la columna especificada como un int en la fila actual de la tabla de resultados.
* getString(int columnIndex): [obtiene el valor](https://docs.oracle.com/en/java/javase/21/docs/api/java.sql/javax/sql/RowSet.html#getString(int)) de la columna especificada como un String en la fila actual de la tabla de resultados.
* close(): [cierra el objeto RowSet](https://docs.oracle.com/en/java/javase/21/docs/api/java.sql/javax/sql/RowSet.html#close--).
* addRowSetListener(RowSetListener listener): [registra un objeto RowSetListener](https://docs.oracle.com/en/java/javase/21/docs/api/java.sql/javax/sql/RowSet.html#addRowSetListener(javax.sql.RowSetListener)) para que reciba eventos de este objeto RowSet.
* removeRowSetListener(RowSetListener listener): [deshace el registro de un objeto RowSetListener](https://docs.oracle.com/en/java/javase/21/docs/api/java.sql/javax/sql/RowSet.html#removeRowSetListener(javax.sql.RowSetListener)) para que deje de recibir eventos de este objeto RowSet.
* …

### 3. La interfaz JdbcRowSet

public interface JdbcRowSet extends RowSet, Joinable

Para conectar RowSet con la base de datos, la interfaz RowSet proporciona métodos para configurar las propiedades de JavaBean:

void setURL(String url);

void setUserName(String user\_name);

void setPassword(String password);

Sólo se precisa crear un objeto JdbcRowSet:

JdbcRowSet rowSet = RowSetProvider.newFactory().createJdbcRowSet();

// 1. Base de datos Oracle

rowSet.setUrl("jdbc:oracle:thin:@localhost:1521:xe");

// 2. El nombre de usuario se establece personalmente como - root

rowSet.setUsername("root");

// 3. La contraseña se establece personalmente como - pass

rowSet.setPassword("pass");

// 4. Consulta

rowSet.setCommand("select \* from Students");

#### Implementación

Supongamos que tenemos una tabla llamada Estudiante en la base de datos con datos:

| **idEstudiante** | **nombre** | **nota** |
| --- | --- | --- |
| 1 | otto | 92 |
| 2 | xoel | 90 |
| 3 | marco | 80 |
| 4 | xoan | 82 |

Implementación de JdbcRowSet y recuperación de los registros:

// Programa Java para ilustrar RowSet en JDBC

// Importación de base de datos

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.ResultSet;

import java.sql.Statement;

import javax.sql.RowSetEvent;

import javax.sql.RowSetListener;

import javax.sql.rowset.JdbcRowSet;

import javax.sql.rowset.RowSetProvider;

// Clase principal

class RowSetDemo {

// Método principal

public static void main(String args[]) {

// Bloque Try para verificar excepciones

try {

// Carga y registro de controladores

Class.forName("oracle.jdbc.driver.OracleDriver");

// Creación de un RowSet

JdbcRowSet rowSet = RowSetProvider.newFactory().createJdbcRowSet();

// Configuración de URL, nombre de usuario, contraseña

rowSet.setUrl("jdbc:oracle:thin:@localhost:1521:xe");

rowSet.setUsername("root");

rowSet.setPassword("pass");

// Creación de una consulta

rowSet.setCommand("select \* from Estudiante");

// Ejecución de la consulta

rowSet.execute();

// Procesamiento de los resultados

while (rowSet.next()) {

// Comandos de impresión y visualización

System.out.println("idEstudiante: " + rowSet.getInt(1));

System.out.println("nombre: " + rowSet.getString(2));

System.out.printf("nota: %.1f", rowSet.geInt(3)/10.);

}

}

// Bloque catch para manejar las excepciones

catch (Exception e) {

// Imprimir y mostrar la excepción junto con

// el número de línea usando el método printStackTrace()

e.printStackTrace();

}

}

}

**Salida:**

idEstudiante: 1

nombre: otto

nota: 9.2

idEstudiante: 2

nombre: xoel

nota: 9.0

idEstudiante: 3

nombre: marco

nota: 8.0

idEstudiante: 4

nombre: xoan

nota: 8.2

#### Métodos principales de la interfaz JdbcRowSet que no heredan de RowSet

Ejemplo de selección parametrizada JdbcRowSet:

JdbcRowSetImpl jrs = new JdbcRowSetImpl();

jrs.setCommand("SELECT \* FROM TITLES WHERE TYPE = ?");

jrs.setURL("jdbc:myDriver:myAttribute");

jrs.setUsername("cervantes");

jrs.setPassword("sancho");

jrs.setString(1, "BIOGRAPHY");

jrs.execute();

* commit(): [hace que todos los cambios realizados](https://docs.oracle.com/en/java/javase/21/docs/api/java.sql.rowset/javax/sql/rowset/JdbcRowSet.html#commit()) en este objeto RowSet desde la última llamada al método commit sean permanentes y los escribe en la base de datos.
* getAutoCommit(): [obtiene el estado de autocommit](https://docs.oracle.com/en/java/javase/21/docs/api/java.sql.rowset/javax/sql/rowset/JdbcRowSet.html#getAutoCommit()) de este objeto RowSet.
* getRowSetWarnings(): [obtiene la primera advertencia](https://docs.oracle.com/en/java/javase/21/docs/api/java.sql.rowset/javax/sql/rowset/JdbcRowSet.html#getRowSetWarnings()) informada por este objeto RowSet.
* getShowDeleted(): [obtiene el estado de la propiedad showDeleted](https://docs.oracle.com/en/java/javase/21/docs/api/java.sql.rowset/javax/sql/rowset/JdbcRowSet.html#getShowDeleted()) de este objeto RowSet.
* rollback(): [deshace todos los cambios realizados](https://docs.oracle.com/en/java/javase/21/docs/api/java.sql.rowset/javax/sql/rowset/JdbcRowSet.html#rollback()) en este objeto RowSet desde la última llamada al método commit y los descarta.
* rollback(Savepoint s): [deshace todos los cambios realizados](https://docs.oracle.com/en/java/javase/21/docs/api/java.sql.rowset/javax/sql/rowset/JdbcRowSet.html#rollback(java.sql.Savepoint)) en este objeto RowSet desde la última llamada al método commit y los descarta.
* setAutoCommit(boolean autoCommit): [establece el modo de autocommit](https://docs.oracle.com/en/java/javase/21/docs/api/java.sql.rowset/javax/sql/rowset/JdbcRowSet.html#setAutoCommit(boolean)) para este objeto RowSet.
* setShowDeleted(boolean b): [establece la propiedad showDeleted](https://docs.oracle.com/en/java/javase/21/docs/api/java.sql.rowset/javax/sql/rowset/JdbcRowSet.html#setShowDeleted(boolean)) de este objeto RowSet en el valor booleano especificado.

# 15. Ejercicios y archivos de la unidad

* [Boletines de ejercicios](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0115boletines/#boletines-de-ejercicios)
* [Archivos de la unidad](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0115boletines/#archivos-de-la-unidad)
* [Ejercicio 1. Gestión Biblioteca](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0115boletines/#ejercicio-1-gesti%C3%B3n-biblioteca)
  + [Estructura de la base de datos](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0115boletines/#estructura-de-la-base-de-datos)
  + [ConnectionManager](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0115boletines/#connectionmanager)
  + [Clase Book implementa Serializable.](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0115boletines/#clase-book-implementa-serializable)
    - [Atributos](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0115boletines/#atributos)
    - [Métodos](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0115boletines/#m%C3%A9todos)
  + [Clase Contido](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0115boletines/#clase-contido-)
  + [Interfaz DAO<T>](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0115boletines/#interfaz-daot)
  + [Clase BookDAO implementa DAO<BooK>](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0115boletines/#clase-bookdao-implementa-daobook)
    - [Atributos](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0115boletines/#atributos-1)
  + [Métodos](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0115boletines/#m%C3%A9todos-1)
* [### Clase ContidoDAO implementa DAO<Contido>](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0115boletines/#-clase-contidodao-implementa-daocontido)
  + [AppBiblioteca](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0115boletines/#appbiblioteca)
    - [Interface gráfica](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0115boletines/#interface-gr%C3%A1fica)

## Boletines de ejercicios

 Ejercicios

* [boletin02.01jdbc.pdf](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0115boletines.files/boletin02.01jdbc.pdf) (946 )

## Archivos de la unidad

La base de datos de ejemplo para los ejercicios de esta unidad es la siguiente (Está incorporado en el fichero anexo):

-- PUBLIC."Book" definition

-- Drop table

-- DROP TABLE PUBLIC."Book";

CREATE TABLE PUBLIC."Book" (

"idBook" INTEGER NOT NULL AUTO\_INCREMENT,

"isbn" CHARACTER VARYING(13) NOT NULL,

"titulo" CHARACTER VARYING(255) NOT NULL,

"autor" CHARACTER VARYING(255),

"anho" INTEGER,

"disponible" BOOLEAN DEFAULT TRUE,

"portada" BINARY LARGE OBJECT,

CONSTRAINT BOOK\_PK PRIMARY KEY ("idBook")

);

CREATE UNIQUE INDEX "IdBookPK" ON PUBLIC."Book" ("idBook");

CREATE INDEX "IdxBookISBN" ON PUBLIC."Book" ("isbn");

CREATE INDEX "IdxBookTitle" ON PUBLIC."Book" ("titulo");

CREATE UNIQUE INDEX PRIMARY\_KEY\_1 ON PUBLIC."Book" ("idBook");

-- PUBLIC."Contido" definition

-- Drop table

-- DROP TABLE PUBLIC."Contido";

CREATE TABLE PUBLIC."Contido" (

"idContido" INTEGER NOT NULL AUTO\_INCREMENT,

"idBook" INTEGER NOT NULL,

"contido" CHARACTER LARGE OBJECT,

CONSTRAINT "Contido\_PK" PRIMARY KEY ("idContido"),

CONSTRAINT FK\_ID\_BOOK PRIMARY KEY ("idBook","idBook")

);

CREATE INDEX FK\_ID\_BOOK\_INDEX\_9 ON PUBLIC."Contido" ("idBook");

CREATE UNIQUE INDEX PRIMARY\_KEY\_9 ON PUBLIC."Contido" ("idContido");

-- PUBLIC."Contido" foreign keys

ALTER TABLE PUBLIC."Contido" ADD CONSTRAINT FK\_ID\_BOOK FOREIGN KEY ("idBook") REFERENCES PUBLIC."Book"("idBook") ON DELETE CASCADE ON UPDATE CASCADE;

 Bases de datos

* [bdbiblioteca.zip](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0115boletines.files/bdbiblioteca.zip) (5 )
* [bdbibliotecaUpperCase.zip](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0115boletines.files/bdbibliotecaUpperCase.zip) (5 )

Las bases de datos están incorporadas en el fichero, así como los datos y los textos e imágenes de ejemplo:

 Proyectos

* [AppBiblioteca.zip](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/03procesandosql/0115boletines.files/AppBiblioteca.zip) (80 )

## Ejercicio 1. Gestión Biblioteca

Queremos desarrollar una **aplicación para una biblioteca y necesitamos interactuar con una base de datos que contiene información sobre los libros que tenemos en nuestra colección de libros**.

Para ello, vamos a crear:

* **Clase Book**: representa la entidad libro.
* **Clase BookDAO**: permite realizar operaciones CRUD (Create, Read, Update y Delete) sobre la tabla Book en la base de datos.
* **Clase ConnectionManager**: para la gestión y obtención de las conexiones a la base de datos de una manera eficiente. Emplearemos el **patrón Singleton** para el gestor de conexiones, que en la primera versión tendrá una única conexión, pero que podremos convertir en un conjunto/pool de conexiones.

### Estructura de la base de datos

Está formada por una única **tabla, Book**. La tabla Contido no se usará de momento.

Tabla Book

| **Columna** | **Tipo de dato** | **Descripción** |
| --- | --- | --- |
| idBook | int | Identificador único del **ejemplar** del libro |
| isbn | varchar(13) | Identificador del libro |
| titulo | varchar(255) | Título del libro |
| autor | varchar(255) | Autor del libro |
| anho | int | Año de publicación del libro |
| disponible | boolean | Indica si el libro está disponible |
| portada | Blob | Portada del libro en formato binario |
| dataPublicacion | Date | Fecha de publicación |

Script de creación de la tabla

CREATE TABLE PUBLIC."Book" (

"idBook" INTEGER NOT NULL AUTO\_INCREMENT,

"isbn" CHARACTER VARYING(13) NOT NULL,

"titulo" CHARACTER VARYING(255) NOT NULL,

"autor" CHARACTER VARYING(255),

"anho" INTEGER,

"disponible" BOOLEAN DEFAULT TRUE,

"portada" BINARY LARGE OBJECT,

"dataPublicacion" DATE,

CONSTRAINT BOOK\_PK PRIMARY KEY ("idBook")

);

CREATE UNIQUE INDEX "IdBookPK" ON PUBLIC."Book" ("idBook");

CREATE INDEX "IdxBookISBN" ON PUBLIC."Book" ("isbn");

CREATE INDEX "IdxBookTitle" ON PUBLIC."Book" ("titulo");

CREATE UNIQUE INDEX PRIMARY\_KEY\_93A ON PUBLIC."Book" ("idBook");

CREATE TABLE PUBLIC.Contido (

idContido INTEGER NOT NULL AUTO\_INCREMENT,

idBook INTEGER NOT NULL,

contido CHARACTER LARGE OBJECT,

CONSTRAINT Contido\_PK PRIMARY KEY (idContido)

);

CREATE INDEX FK\_ID\_BOOK\_INDEX\_9 ON PUBLIC.Contido (idBook);

CREATE UNIQUE INDEX PRIMARY\_KEY\_9 ON PUBLIC.Contido (idContido);

-- PUBLIC.Contido foreign keys

ALTER TABLE PUBLIC.Contido ADD CONSTRAINT FK\_ID\_BOOK FOREIGN KEY (idBook) REFERENCES PUBLIC.Book(idBook) ON DELETE CASCADE ON UPDATE CASCADE;

 DATABASE\_TO\_UPPER=FALSE

Para permitir nombres en CamelCase en H2 JDBC Driver versión 2, agrega la propiedad DATABASE\_TO\_UPPER=FALSE en la URL de conexión.

* **Driver**: "org.h2.Driver" (no se precisa)
* **URL**: "jdbc:h2:rutaBaseDatosSinExtensión;DB\_CLOSE\_ON\_EXIT=TRUE;FILE\_LOCK=NO;DATABASE\_TO\_UPPER=FALSE"

### ConnectionManager

Mediante el **patrón Singleton**, crea una clase ConnectionManager o LibraryConnectionManager con las siguientes características:

* **Atributos**:
  + Instancia de la propia clase como **atributo privado y final: instance**.
  + **Atributo privado de tipo Connection que se crea al invocar el método getConnection**.
* **Recomendaciones**:
  + Puedes hacerlo con **Thread-Safe y doble comprobación**, pero no es relevante para este caso inicial.
  + Usa **constantes privadas para URL, usuario y contraseña, privadas**.

### Clase Book implementa Serializable.

* **Constructores**:
  + Book()
  + Book(String isbn, String title, String author, Short year, Boolean available)
  + Book(Integer idBook, String isbn, String title, String author, Short year, Boolean available, byte[] portada)

#### Atributos

| **Atributo** | **Tipo** | **Descripción** |
| --- | --- | --- |
| idBook | Long | Identificador único del libro |
| isbn | String | Código ISBN del libro |
| titulo | String | Título del libro |
| autor | String | Autor del libro |
| anho | Short | Año de publicación |
| disponible | Boolean | Disponibilidad del libro |
| portada | byte[] | Portada en formato binario |
| dataPublicacion | LocalDate | Data de publicación |

#### Métodos

* **Getters y setters** para cada atributo. Los setters devuelven una referencia al propio objeto.
* **setPortada(File f)**: asigna una portada desde un archivo.
* **setPortada(String f):** asigna una portada desde una ruta.
* **Image getImage():** devuelve un objeto de tipo java.awt.Image si la portada no es nula:
* ByteArrayInputStream flujo = new ByteArrayInputStream(portada);
* ImageIO.read(flujo);
* **equals y hashCode**: dos libros **son iguales si tienen el mismo ISBN.** Además, el método hashcode debe devolver un valor coherente con el método equals (**todos los objetos iguales deben tener, al menos el mismo hashCode**).
* **toString**: Devuelve el título, el autor y el año. Si no está disponible, añade un asterisco.

### Clase Contido

package com.pepinho.ad.biblioteca.model;

import java.sql.Connection;

import java.util.Objects;

public class Contido {

private Long idContido;

private Long idBook;

private String contido;

public Contido() {

}

public Contido(Long idBook, String contido) {

this.idBook = idBook;

this.contido = contido;

}

public Contido(Long idContido, Long idBook) {

this.idContido = idContido;

this.idBook = idBook;

}

public Contido(Long idContido, Long idBook, String contido) {

this.idContido = idContido;

this.idBook = idBook;

this.contido = contido;

}

public Long getIdContido() {

return idContido;

}

public void setIdContido(Long idContido) {

this.idContido = idContido;

}

public Long getIdBook() {

return idBook;

}

public void setIdBook(Long idBook) {

this.idBook = idBook;

}

public String getContido() {

return contido;

}

public void setContido(String contido) {

this.contido = contido;

}

@Override

public int hashCode() {

return 97 \* 7 + Objects.hashCode(this.idContido);

}

@Override

public boolean equals(Object obj) {

if (obj == null || !(obj instanceof Contido other)) return false;

return Objects.equals(this.idContido, other.idContido);

}

@Override

public String toString() {

return idContido + " (" + idBook + "): " + contido;

}

}

### Interfaz DAO<T>

Esta interface será implantada por todas aquellas clases DAO que trabajen con objetos con imágenes. Los nombres de los métodos son totalmente descriptivos:

T get(long id);

List<T> getAll();

void save(T t);

void update(T t);

void delete(T t);

boolean deleteById(long id);

List<Integer> getAllIds();

void updateLOB(T book, String f);

void updateLOBById(long id, String f);

void deleteAll();

### Clase BookDAO implementa DAO<BooK>

Tiene como **atributo final un objeto de tipo Connection**, con, que recoge como argumento el constructor:

public BookDAO(Connection con) {

this.con = con;

}

#### Atributos

* Objeto Connection, pasado al constructor.

### Métodos

1. **get(long idBook)**: devuelve un objeto **Book** con la información del libro que tiene el identificador pasado como parámetro.
2. **getAll()**: devuelve una lista de todos los libros almacenados en la base de datos.
3. **save(Book book)**: crea un nuevo registro en la tabla **Book** con la información del libro pasado como parámetro. Importante: además, debe guardar el idBook en el objeto, por lo que es necesario obtener el ID del registro insertado.
   * Debe crearse la sentencia con la opción Statement.RETURN\_GENERATED\_KEYS.
4. **update(Book book)**: actualiza la información del registro correspondiente al libro pasado como parámetro.
5. **delete(Book book)**: elimina el registro correspondiente al libro con el identificador del libro pasado como parámetro.
6. **deleteById(long idBook)**: elimina el registro correspondiente al libro con el identificador pasado como parámetro.
7. **getAllIds()**: devuelve una lista con los ids de todos los libros de la base de datos.
8. **updateLOB(Book b, String f)**:actualiza el libro en la base de datos con el contenido del archivo recogido como parámetro. Usa setBinaryStream.
   * Ayuda: puedes emplear el método setBinaryStream de PreparedStatement), previamente habiendo leído los bytes.
9. **updateLOBByID(long b, String f)**: actualiza el libro con el id recogido como por parámetro con el contenido del archivo recogido como parámetro.
10. **deleteAll()**: borra todos los libros.

Debes implantar la gestión de sentencias de esta la clase **BookDAO por medio de try-with-resources para manejar los cierres de los Statement y los ResultSet de consultas automáticamente**.

**La conexión no debe cerrarse**, pues debe permanecer abierta para futuros usos.

### Clase ContidoDAO implementa DAO<Contido>

### AppBiblioteca

a) Haz una **aplicación que haga uso el ConnectionManager para obtener una conexión y se la pase al constructor de BooKDAO**. Crea varios libros y añádelos a la base de datos, incluyendo las portadas de los libros desde la aplicación.

* Usa ConnectionManager para obtener una conexión y pasarla al constructor de BookDAO.
* Crea varios libros y añádelos a la base de datos.

Ejemplo de libros:

INSERT INTO PUBLIC.Book (isbn, titulo, autor, anho, disponible, portada)

VALUES ('9780307277672', 'Cien años de soledad', 'Gabriel García Márquez', 1967, TRUE, NULL);

INSERT INTO PUBLIC.Book (isbn, titulo, autor, anho, disponible, portada)

VALUES ('9780743273565', 'Harry Potter y la piedra filosofal', 'J.K. Rowling', 1997, TRUE, NULL);

INSERT INTO PUBLIC.Book (isbn, titulo, autor, anho, disponible, portada)

VALUES ('9780307277672', 'Cien años de soledad', 'Gabriel García Márquez', 1967, TRUE, NULL);

INSERT INTO PUBLIC.Book (isbn, titulo, autor, anho, disponible, portada)

VALUES ('9780743273565', 'Harry Potter y la piedra filosofal', 'J.K. Rowling', 1997, TRUE, NULL);

VALUES ('9780307959474', 'The Sense of an Ending', 'Julian Barnes', 2011, TRUE, NULL);

INSERT INTO PUBLIC.Book (isbn, titulo, autor, anho, disponible, portada)

VALUES ('9780307386672', 'No Country for Old Men', 'Cormac McCarthy', 2005, TRUE, NULL);

INSERT INTO PUBLIC.Book (isbn, titulo, autor, anho, disponible, portada)

VALUES ('9781400064168', 'The Road', 'Cormac McCarthy', 2006, TRUE, NULL);

INSERT INTO PUBLIC.Book (isbn, titulo, autor, anho, disponible, portada)

VALUES ('9780099590088', 'The Noise of Time', 'Julian Barnes', 2016, TRUE, NULL);

INSERT INTO PUBLIC.Book (isbn, titulo, autor, anho, disponible, portada)

VALUES ('9780307277672', 'All the Pretty Horses', 'Cormac McCarthy', 1992, TRUE, NULL);

INSERT INTO PUBLIC.Book (isbn, titulo, autor, anho, disponible, portada)

VALUES ('9780099590088', 'Levels of Life', 'Julian Barnes', 2013, TRUE, NULL);

#### Interface gráfica

Crea una aplicación siguiendo la estructura de MVC con el modelo de datos anterior. Puedes crear una vista o emplear la proporcionada en los apuntes:

# Proyecto 1ª evaluación

* [Introducción](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/99proyecto/#introducci%C3%B3n)
* [Persistencia de datos](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/99proyecto/#persistencia-de-datos)
  + [Ejemplo ViewModel con Retrofit](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/99proyecto/#ejemplo-viewmodel-con-retrofit)
    - [1. Dependencias](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/99proyecto/#1-dependencias)
    - [2. Modelo de datos](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/99proyecto/#2-modelo-de-datos)
    - [3. Configuración Retrofit](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/99proyecto/#3-configuraci%C3%B3n-retrofit)
    - [4. Creación el repositorio](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/99proyecto/#4-creaci%C3%B3n-el-repositorio)
    - [5. Creación del ViewModel](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/99proyecto/#5-creaci%C3%B3n-del-viewmodel)
    - [6. Conectar el ViewModel con la vista](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/99proyecto/#6-conectar-el-viewmodel-con-la-vista)
  + [Ejemplo con Jetpack Compose y StateFlow](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/99proyecto/#ejemplo-con-jetpack-compose-y-stateflow)
  + [Jetpack Compose y ViewModel](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/99proyecto/#jetpack-compose-y-viewmodel)
    - [1. Dependencias](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/99proyecto/#1-dependencias-1)
    - [2. Creación del ViewModel](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/99proyecto/#2-creaci%C3%B3n-del-viewmodel)
    - [3. Creación de la UI con Compose](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/99proyecto/#3-creaci%C3%B3n-de-la-ui-con-compose)
    - [Ventajas de usar StateFlow con Compose](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/99proyecto/#ventajas-de-usar-stateflow-con-compose)
* [Referencias](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/99proyecto/#referencias)
  + [Room Persistence Library](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/99proyecto/#room-persistence-library)
  + [MVVM Architecture](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/99proyecto/#mvvm-architecture)
  + [Retrofit](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/02accesobd/99proyecto/#retrofit)

## Introducción

El objeto del proyecto es **diseñar e implantar un modelo de datos adecuado para una aplicación**, tanto para dispositivos móviles como para escritorio.

El resultado final será un **modelo de datos** compuesto por varias clases, los **adaptadores** necesarios para trabajar con archivos o la base de datos y una **interfaz de usuario** que permita interactuar con los datos.

El **lenguaje de programación empleado será Kotlin o Java**, según se considere más adecuado para el proyecto elegido.

 Interoperatividad entre Kotlin y Java

Kotlin es un lenguaje que se puede utilizar de forma interoperable con Java, lo que significa que **se pueden mezclar ambos lenguajes en un mismo proyecto**. Esto **permite a los desarrolladores aprovechar las ventajas de ambos lenguajes y utilizarlos en conjunto en sus aplicaciones**.

Incluso en Android Studio, se puede **convertir código Java a Kotlin de forma automática** o **crear el modelo de datos en Java y el resto de la aplicación en Kotlin**.

Hazlo como mejor te convenga, pero **asegúrate de que el código sea coherente y siga las convenciones de codificación del lenguaje elegido**.

El proyecto debe cumplir la siguiente estructura:

1. **Persistencia de datos**: se debe implantar la persistencia de datos en la aplicación, **utilizando archivos o una base de datos según se considere más apropiado para el proyecto elegido**.
   * El **formato de los archivos** puede ser:
     + Binario.
     + Texto.
     + Archivo de texto en formato JSON (locales o remotos).
   * La **base de datos** puede ser: Postgres, H2, SQLite o cualquier otra base de datos que consideres adecuada.
   * En el caso de **aplicaciones móviles**, se debe tener en cuenta la gestión de la persistencia de datos en el contexto de una aplicación móvil, utilizando el patrón de acceso a datos adecuado para la plataforma: **ViewModel** (para la gestión de la UI), [**LiveData**](https://developer.android.com/topic/libraries/architecture/livedata?hl=es-419) o [**StateFlow**](https://developer.android.com/kotlin/flow/stateflow-and-sharedflow?hl=es-419) (para la observación de datos), **Room** (con SQLite y patrón Repository), **Retrofit** (para acceso a servicios web), etc.
   * También pueden utilizarse plataformas **BaaS (Backend as a Service)** como **Firebase**, **Supabase**, etc.

Para Firebase, se puede emplear la librería de Firebase para Android, que proporciona una API sencilla para interactuar con la base de datos en tiempo real y el sistema de autenticación de Firebase:

<https://developer.android.com/studio/write/firebase> <https://firebase.google.com/docs/android/setup>

1. **Modelo de datos**: se debe un modelo de datos que represente la información que se va a manejar en la aplicación. Este modelo debe ser coherente y adecuado para el propósito de la aplicación, siguiendo los estándares de nombrado en Kotlin o Java. Además, debe **incluir los adaptadores de Gson en el caso de emplear JSON**.
2. **Patrones de acceso a datos**: se deben implementar los patrones de acceso a datos necesarios para interactuar con los datos almacenados. Esto puede incluir la **creación de clases de acceso a datos, adaptadores, o cualquier otro componente necesario para gestionar la persistencia de datos**.
   * Como se ha comentado anteriormentte, en el caso de aplicaciones móviles, se debe tener en cuenta la gestión de la persistencia de datos en el contexto de una aplicación móvil, **utilizando el patrón de acceso a datos adecuado** para la plataforma: **ViewModel** (para la gestión de la UI), [**LiveData**](https://developer.android.com/topic/libraries/architecture/livedata?hl=es-419) o [**StateFlow**](https://developer.android.com/kotlin/flow/stateflow-and-sharedflow?hl=es-419) (para la observación de datos), Room (con SQLite y patrón Repository), Retrofit (para acceso a servicios web), etc.
   * En el caso de aplicaciones de escritorio, se debe tener en cuenta la gestión de la persistencia de datos en el contexto de una aplicación de escritorio, utilizando el patrón de acceso a datos adecuado para la plataforma: DAO (Data Access Object), JDBC (Java Database Connectivity), etc.
3. **Interfaz de usuario**: aunque la parte importante es la parte del modelo y del patrón de arquitectura empleado, se debe implementar una interfaz de usuario que permita interactuar con los datos almacenados.
   * En el caso de aplicaciones móviles, se debe tener en cuenta la gestión de la interfaz de usuario en el contexto de una aplicación móvil, utilizando los componentes de la interfaz de usuario adecuados para la plataforma: Activities, Fragments, Views, etc.
   * En el caso de aplicaciones de escritorio, se debe tener en cuenta la gestión de la interfaz de usuario en el contexto de una aplicación de escritorio, utilizando los componentes de la interfaz de usuario adecuados para la plataforma: JFrame, JPanel, JDialog, etc. No es necesario que siga el patrón MVC, aunque se valorará positivamente si se ha seguido.

## Persistencia de datos

Para la persistencia de datos, se debe elegir una de las siguientes opciones:

1. **Persistencia en archivos**: se pueden utilizar archivos para almacenar los datos de la aplicación. Los archivos pueden ser de **texto, binarios o JSON**, y pueden ser almacenados localmente en el dispositivo o de forma remota en un servidor. También p**ueden emplearse API Rest libres o disponibles en la red**, como los ejemplos con los que hemos trabajado en clase. Ejemplos:
   * Almacenar los datos de una lista de tareas pendientes en un archivo de texto o JSON.
   * Almacenar los datos de una lista de contactos en un archivo binario.
   * Consulta de API de películas o libros para obtener información y/o almacenarla localmente.

Existen muchas APIs públicas, por ejemplo:

* <https://github.com/public-apis/public-apis>
* [The movie DB](https://developers.themoviedb.org/3/getting-started/introduction)
* [Google Books API](https://developers.google.com/books/docs/overview)
* [API de Spotify](https://developer.spotify.com/documentation/web-api/)
* [News API](https://newsapi.org/docs/get-started)
* [API de OpenWeatherMap](https://openweathermap.org/api)
* [API de GitHub](https://docs.github.com/en/rest)
* [API de X/Twitter](https://developer.x.com/en/docs)
* [API de NASA](https://api.nasa.gov/)
* [API iTunes](https://itunes.apple.com/search?media=music&term=bach) ;-) (<https://performance-partners.apple.com/search-api>)

En el **caso de emplear JSON deben crearse serializadores o deserializadores personalizados por medio de Gson**.

En Android debe emplearse la clase File para la gestión de archivos y en Java la clase FileReader y FileWriter.

Para acceso a las API Rest, **en Android se empleará Retrofit y en Java la clase HttpURLConnection, usando la clases con buffer**.

Para Android el API Rest debe estar separada de la vista por medio del [patrón MVVM](https://developer.android.com/topic/libraries/architecture/viewmodel?hl=es-419)(Model-View-ViewModel) y en Java por medio del patrón DAO (Data Access Object):

<https://developer.android.com/topic/libraries/architecture/viewmodel?hl=es-419> <https://developer.android.com/codelabs/basic-android-kotlin-training-livedata?hl=es-419#0>

En la actualidad se emplea el patrón MVVM en Android, que se basa en la separación de la lógica de negocio de la vista, permitiendo una mayor modularidad y reutilización del código. En Java se emplea el patrón DAO (Data Access Object) para la separación de la lógica de acceso a datos de la lógica de negocio.

Para la retención de datos en Android por medio de MVVM se emplea la clase ViewModel y LiveData para la observación de datos. En Java se emplea la clase DAO para la gestión de la persistencia de datos. Sin embargo, existe un aproximación más actual para

### Ejemplo ViewModel con Retrofit

Ejemplo ViewModel con Retrofit en una aplicación Android siguiendo el patrón **MVVM** (Model-View-ViewModel).

Arquitectura de la aplicación:

├── model

│ └── User.kt

├── network

│ ├── ApiService.kt

│ └── RetrofitClient.kt

├── repository

│ └── UserRepository.kt

└── viewmodel

└── UserViewModel.kt

├── UserActivity.kt

└── activity\_user.xml

View

|

ViewModel

|

Repository

|

Retrofit

|

API

#### 1. Dependencias

Primero, asegúrate de tener las siguientes dependencias en tu archivo build.gradle:

dependencies {

implementation("androidx.lifecycle:lifecycle-viewmodel-ktx:2.8.7")

implementation("com.squareup.retrofit2:retrofit:2.11.0")

implementation("com.squareup.retrofit2:converter-gson:2.11.0")

implementation("com.squareup.okhttp3:logging-interceptor:4.9.1")

}

#### 2. Modelo de datos

Define una clase de datos que represente la respuesta de la API:

data class User(

val id: Int,

val name: String,

val email: String

)

#### 3. Configuración Retrofit

Configura Retrofit para realizar las llamadas a la API:

interface ApiService {

@GET("users")

suspend fun getUsers(): List<User>

// Si se le quiere pasar un paámetro a la API

@GET("users/{id}")

suspend fun getUser(@Path("id") id: Int): User

}

// Configuración de Retrofit

object RetrofitClient {

// Esta API es un ejemplo, puedes reemplazarla por la que necesites

// https://jsonplaceholder.typicode.com/ es un servicio de prueba gratuito

private const val BASE\_URL = "https://jsonplaceholder.typicode.com/"

val apiService: ApiService by lazy {

val retrofit = Retrofit.Builder()

.baseUrl(BASE\_URL)

.addConverterFactory(GsonConverterFactory.create()) // Deberías personalizarlo según tus necesidades

.build()

retrofit.create(ApiService::class.java)

}

}

#### 4. Creación el repositorio

El repositorio se encarga de obtener los datos de la API:

class UserRepository {

private val apiService = RetrofitClient.apiService

suspend fun getUsers(): List<User> {

return apiService.getUsers() // Podrían gestionarse errores

}

}

#### 5. Creación del ViewModel

El ViewModel se comunica con el repositorio y expone los datos a la vista:

class UserViewModel : ViewModel() {

private val userRepository = UserRepository()

private val \_users = MutableLiveData<List<User>>()

val users: LiveData<List<User>> get() = \_users

init {

fetchUsers()

}

private fun fetchUsers() {

viewModelScope.launch { // Corrutina para llamada a la API en el hilo del viewModel

try {

val userList = userRepository.getUsers()

\_users.postValue(userList) // notificar a la vista que los datos han cambiado

} catch (e: Exception) {

// Manejar el error

}

}

}

}

#### 6. Conectar el ViewModel con la vista

Finalmente, conecta el ViewModel con tu actividad o fragmento:

class UserActivity : AppCompatActivity() {

private lateinit var userViewModel: UserViewModel

override fun onCreate(savedInstanceState: Bundle?) {

super.onCreate(savedInstanceState)

setContentView(R.layout.activity\_user)

userViewModel = ViewModelProvider(this).get(UserViewModel::class.java)

userViewModel.users.observe(this, Observer { users ->

// Actualizar la UI con la lista de usuarios

})

}

}

Este es un ejemplo básico para empezar. Puedes expandirlo añadiendo manejo de errores más robusto, pruebas unitarias, y otras mejoras según tus necesidades.

Ejemplo de uso de Retrofit en Android con MVVM con corrutinas y LiveData para una API de películas:

/\* El servicio de Retrofit se declara con una interfaz que define los métodos de \* la API

\* En este caso getPopularMovies() para obtener las películas populares devuelve un objeto de tipo Response<PeliculaResponse>, que permite gestionar la respuesta de la API y errores.

\* \*/

interface PeliculasService {

@GET("movie/popular")

suspend fun getPeliculasPopulares(

@Query("key") claveAPI: String

): Response<PeliculaResponse>

}

// El repositorio de películas se encarga de llamar al servicio de Retrofit y gestionar la respuesta (no he creado la clase intermedia RetrofitClient)

// A diferencia del caso anterior en el que se emplea un objeto RetrofitClient, aquí se inyecta el servicio en el constructor del repositorio

class PeliculaRepository(private val servicioPeliculas: PeliculasService) {

// El identificado "suspend" indica que la función debe ser llamada desde una corrutina

// y que no bloqueará el hilo principal

suspend fun getPelicualsPopulares(): Response<PeliculaResponse> {

return servicioPeliculas.getPeliculasPopulares("tuvalordelaclave")

}

}

// La respuesta de la API se mapea a un objeto de datos

data class PeliculaResponse(val results: List<Movie>)

// Objeto de datos de película

data class Movie(val title: String, val overview: String, val posterPath: String)

// El ViewModel de películas se encarga de gestionar la lógica de la vista y la llamada a la API

// y de exponer los datos a la vista

class MovieViewModel(private val peliculaRepository: PeliculaRepository) : ViewModel() {

private val \_popularMovies = MutableLiveData<List<Movie>>()

val popularMovies: LiveData<List<Movie>> = \_popularMovies

fun getPopularMovies() {

viewModelScope.launch { // Corrutina para llamada a la API en el hilo del viewModel

val response = peliculaRepository.getPelicualsPopulares()

if (response.isSuccessful) {

\_popularMovies.value = response.body()?.results

}

}

}

}

La actividad principal debe tener un objeto de la clase MovieViewModel y una lista de objetos de la clase Movie, además de las vistas de la interfaz de usuario:

class MainActivity : AppCompatActivity() {

private lateinit var movieViewModel: MovieViewModel

override fun onCreate(savedInstanceState: Bundle?) {

super.onCreate(savedInstanceState)

setContentView(R.layout.activity\_main)

// Inicializar el ViewModel y pasarle una instancia del repositorio:

val retrofit = Retrofit.Builder()

.baseUrl("https://api.themoviedb.org/3/")

.addConverterFactory(GsonConverterFactory.create())

.build()

val peliculasService = retrofit.create(PeliculasService::class.java)

val peliculaRepository = PeliculaRepository(peliculasService)

movieViewModel = ViewModelProvider(this, MovieViewModelFactory(peliculaRepository)).get(MovieViewModel::class.java)

movieViewModel.getPopularMovies()

// Observar los cambios en la lista de películas populares

movieViewModel.popularMovies.observe(this, Observer { movies ->

// Actualizar la interfaz de usuario con la lista de películas

val adapter = MovieAdapter(movies)

recyclerView.adapter = adapter

})

}

}

Más información sobre Retrofit en Android: [Retrofit](https://square.github.io/retrofit/) Más información sobre corrutinas en Android: [Corrutinas](https://developer.android.com/kotlin/coroutines) Más información sobre LiveData en Android: [LiveData](https://developer.android.com/topic/libraries/architecture/livedata)

 StateFlow, FLow vs LiveData

En la actualidad, **se recomienda el uso de StateFlow en lugar de LiveData para la observación de datos en Android, ya que StateFlow es más flexible y permite una gestión más eficiente de los datos en la aplicación**. Sin embargo, el uso de LiveData sigue siendo válido y es una opción viable para la observación de datos en Android.

<https://medium.com/@codzure/livedata-vs-stateflow-the-battle-of-the-observables-730f846be812>

StateFlow y LiveData tienen similitudes. Ambas **son clases contenedoras de datos observables y siguen un patrón similar cuando se usan en la arquitectura de una app**.

Sin embargo, ten en cuenta que StateFlow y LiveData se comportan de manera diferente:

* StateFlow **requiere que se pase un estado inicial al constructor**, mientras que LiveData, no.
* LiveData.observe() **cancela automáticamente el registro del consumidor cuando la vista pasa al estado STOPPED**, mientras que la recopilación de StateFlow o cualquier otro flujo, no deja de recopilar automáticamente. Para obtener el mismo comportamiento, debes recopilar el flujo desde un bloque Lifecycle.repeatOnLifecycle.

1. **Persistencia en base de datos**: se pueden utilizar bases de datos para almacenar los datos de la aplicación. Las bases de datos pueden ser **locales o remotas**, y pueden ser de **distintos tipos (SQLite, Postgres, MySQL, etc.)**. También se pueden utilizar **servicios de bases de datos en la nube**. Ejemplos:
   * Almacenar los datos de una lista de tareas pendientes en una base de datos SQLite.
   * Almacenar los datos de una lista de contactos en una base de datos Postgres.
   * Consulta de API de películas o libros para obtener información y almacenarla en una base de datos local o remota.

* En Android se empleará Room con SQLite o, menos recomendable, driver JDBC compatibles con Android, y en Java JDBC con MySQL o Postgres.
* En Android se empleará el patrón Repository para la gestión de la base de datos y en Java el patrón DAO (Data Access Object).
* En Android se empleará el patrón MVVM para la gestión de la interfaz de usuario y en Java el patrón MVC.

### Ejemplo con Jetpack Compose y StateFlow

Con **Jetpack Compose**, el enfoque cambia un poco, pero los principios de usar ViewModel y LiveData (o StateFlow) siguen siendo útiles para manejar el estado de la UI de manera reactiva y eficiente.

### Jetpack Compose y ViewModel

En Jetpack Compose, puedes seguir usando ViewModel para manejar la lógica de negocio y el estado de la UI. Sin embargo, en lugar de LiveData, **es común usar State y StateFlow para una integración más fluida con Compose**.

#### 1. Dependencias

Debes tener las siguientes dependencias en tu archivo build.gradle:

dependencies {

implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:2.5.1"

implementation "androidx.activity:activity-compose:1.7.0"

implementation "androidx.compose.ui:ui:1.4.0"

implementation "androidx.compose.material:material:1.4.0"

implementation "androidx.compose.ui:ui-tooling-preview:1.4.0"

implementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:1.6.4"

}

Con kotlin llevarán paréntesis y con groovy no. Asegúrate de que las versiones sean las más recientes.

#### 2. Creación del ViewModel

Define tu ViewModel usando StateFlow:

class UserViewModel : ViewModel() {

private val \_users = MutableStateFlow<List<User>>(emptyList())

val users: StateFlow<List<User>> get() = \_users

init {

fetchUsers()

}

private fun fetchUsers() {

viewModelScope.launch {

try {

val userList = apiService.getUsers()

\_users.value = userList

} catch (e: Exception) {

// Manejar el error

}

}

}

}

#### 3. Creación de la UI con Compose

Usa **collectAsState para observar los cambios en StateFlow y actualizar la UI**:

@Composable

fun UserScreen(userViewModel: UserViewModel = viewModel()) {

val users by userViewModel.users.collectAsState()

LazyColumn {

items(users) { user ->

Text(text = user.name)

}

}

}

@Composable

fun MyApp() {

MaterialTheme {

UserScreen()

}

}

class MainActivity : ComponentActivity() {

override fun onCreate(savedInstanceState: Bundle?) {

super.onCreate(savedInstanceState)

setContent {

MyApp()

}

}

}

#### Ventajas de usar StateFlow con Compose

* **Reactividad**: StateFlow es una API de flujo de datos que es inherentemente reactiva, lo que se integra perfectamente con el paradigma de Compose.
* **Integración fluida**: collectAsState convierte un StateFlow en un State de Compose, lo que **facilita la actualización de la UI en respuesta a cambios de datos**.
* **Manejo del ciclo de vida**: Al igual que LiveData, StateFlow respeta el ciclo de vida de los componentes de la UI, evitando fugas de memoria y actualizaciones innecesarias.

**Puedes usar LiveData con Jetpack Compose, StateFlow ofrece una integración más natural y eficiente con el paradigma declarativo de Compose**.

## Referencias

### Room Persistence Library

* <https://developer.android.com/training/data-storage/room?hl=es-419> (Cómo guardar datos en una base de datos local usando Room)
* <https://developer.android.com/jetpack/androidx/releases/room?hl=es-419>

### MVVM Architecture

* <https://developer.android.com/topic/libraries/architecture/viewmodel?hl=es-419> (ViewModel and State in Compose)
* <https://developer.android.com/topic/libraries/architecture/viewmodel?hl=es-419> (ViewModel overview)
* <https://developer.android.com/topic/libraries/architecture/livedata?hl=es-419> (LiveData overview)
* <https://developer.android.com/kotlin/flow/stateflow-and-sharedflow?hl=es-419> (StateFlow y SharedFlow)
* <https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/-state-flow/> (StateFlow API)
* <https://cursokotlin.com/mvvm-en-android-con-kotlin-livedata-y-view-binding-android-architecture-components/> (MVVM en Android con Kotlin)

### Retrofit

* [Consuming a REST API using Retrofit2 with the MVVM Pattern in Android](https://learntodroid.com/consuming-a-rest-api-using-retrofit2-with-the-mvvm-pattern-in-android/)
* [Ejemplo de aplicación Android con Retrofit, MVVM, ViewModel, LiveData y Corrutinas](tps://github.com/ShashankBale/Retrofit-with-MVVM)
* <https://velog.io/@rakuleeinc/Android-ViewModel-DataBinding-Retrofit-RecyclerView-Example>
* [REtrofit with Kotlin Flow](https://outcomeschool.com/blog/retrofit-with-kotlin-flow)