

0101010  
0100101  
1101010

# UD12.1- ACCESO Y MANIPULACIÓN DE BASES DE DATOS RELACIONALES

Programación – 1er DAW/DAM

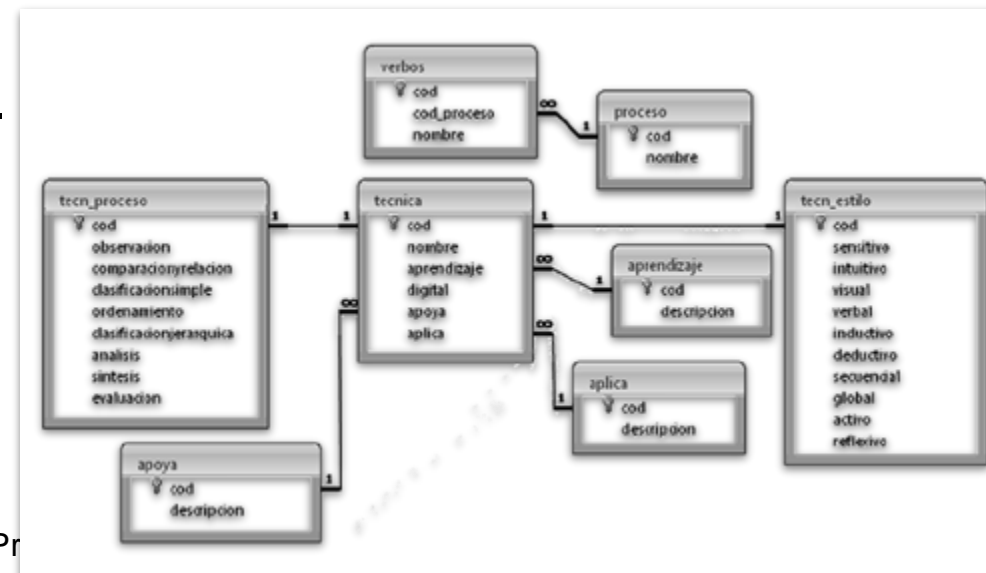
# 0. ÍNDICE

---

- INTRODUCCIÓN
  - ARQUITECTURA
- ESTABLECIMIENTO DE LA CONEXIÓN
  - ESTÁNDARES DE CONEXIÓN
  - DRIVERS JDBC
- COMUNICACIÓN CON EL SGBD
  - CONEXIÓN A BASES DE DATOS
  - CLASES SERVICIO
- EJECUCIÓN DE CONSULTAS
- CONSULTAS DE ACTUALIZACIÓN
- SENTENCIAS PREPARADAS

# 1. INTRODUCCIÓN

- Cuando la **cantidad de datos** a gestionar **crece** y la **estructura** de los mismos se hace **compleja**, su **tratamiento** mediante **ficheros** es ineficiente, susceptible a **errores** y, por tanto, poco recomendable.
- Las **Sistemas Gestores de Bases de Datos (SGBD)** proporcionan una capa de persistencia, **independiente de las aplicaciones**, para el **manejo** eficiente de la **información**.:
  - Mantenimiento de la **integridad**.
  - La **compartición** de datos y el **acceso concurrente** por diferentes aplicaciones.
  - **Atomicidad** de las operaciones.
  - **Políticas de seguridad**
  - ...



# 1. INTRODUCCIÓN

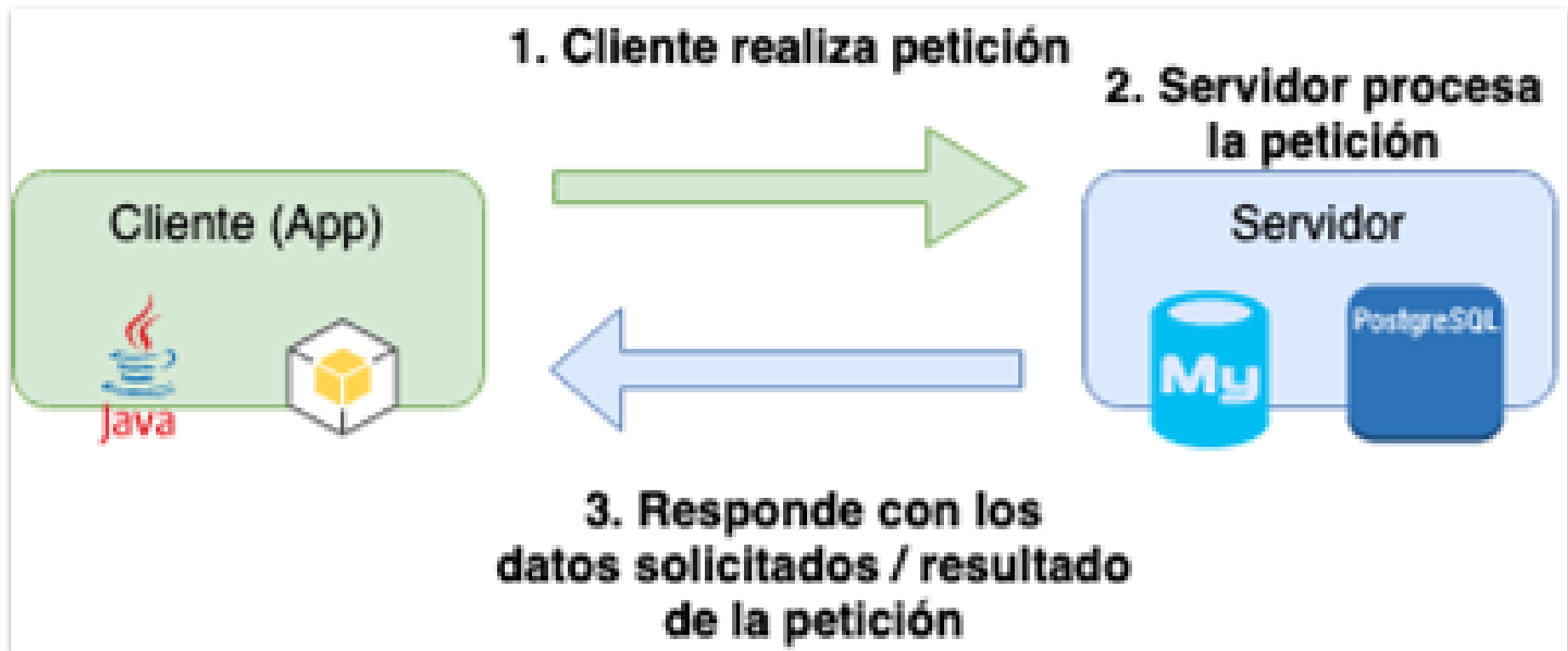
- En función de la forma en la que ofrecen su servicio podemos diferenciar **2 tipos SGBD relacionales**:
  - **Embebidos**: Las bases de datos son gestionadas por la propia aplicación a partir de un conjunto de librerías y la información se guarda en ficheros locales (*SQLite*).
  - **Externos**: Funcionan como un servicio o proceso independiente cuya comunicación se lleva a cabo a través de comunicación local o en red (*Mysql/Mariadb, PostgreSQL,...*).

En cualquier caso, utilizaremos el **lenguaje SQL** para la **definición y manipulación** de la información contenida en la **BD**



# 1.1 ARQUITECTURA I

- Los SGBD proporcionan una arquitectura **cliente-servidor**



# 1.1 ARQUITECTURA II

- **Arquitectura**

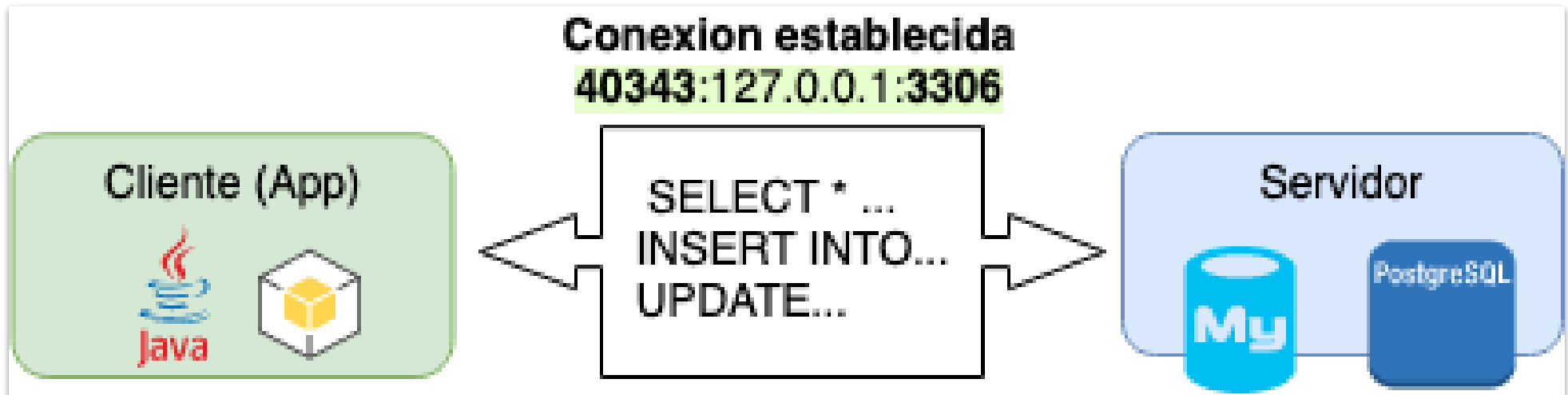
- Cliente y servidor **no** necesariamente **estarán en la misma máquina**.
- Para poder comunicarnos con el servidor deberemos de establecer una **conexión previa** con el mismo.



# 1.1 ARQUITECTURA III

- Arquitectura

- Utilizaremos la **conexión** establecida para realizar todas las **comunicaciones** posteriores (consultas y manipulación de datos)



## 2. ESTABLECIMIENTO DE LA CONEXIÓN

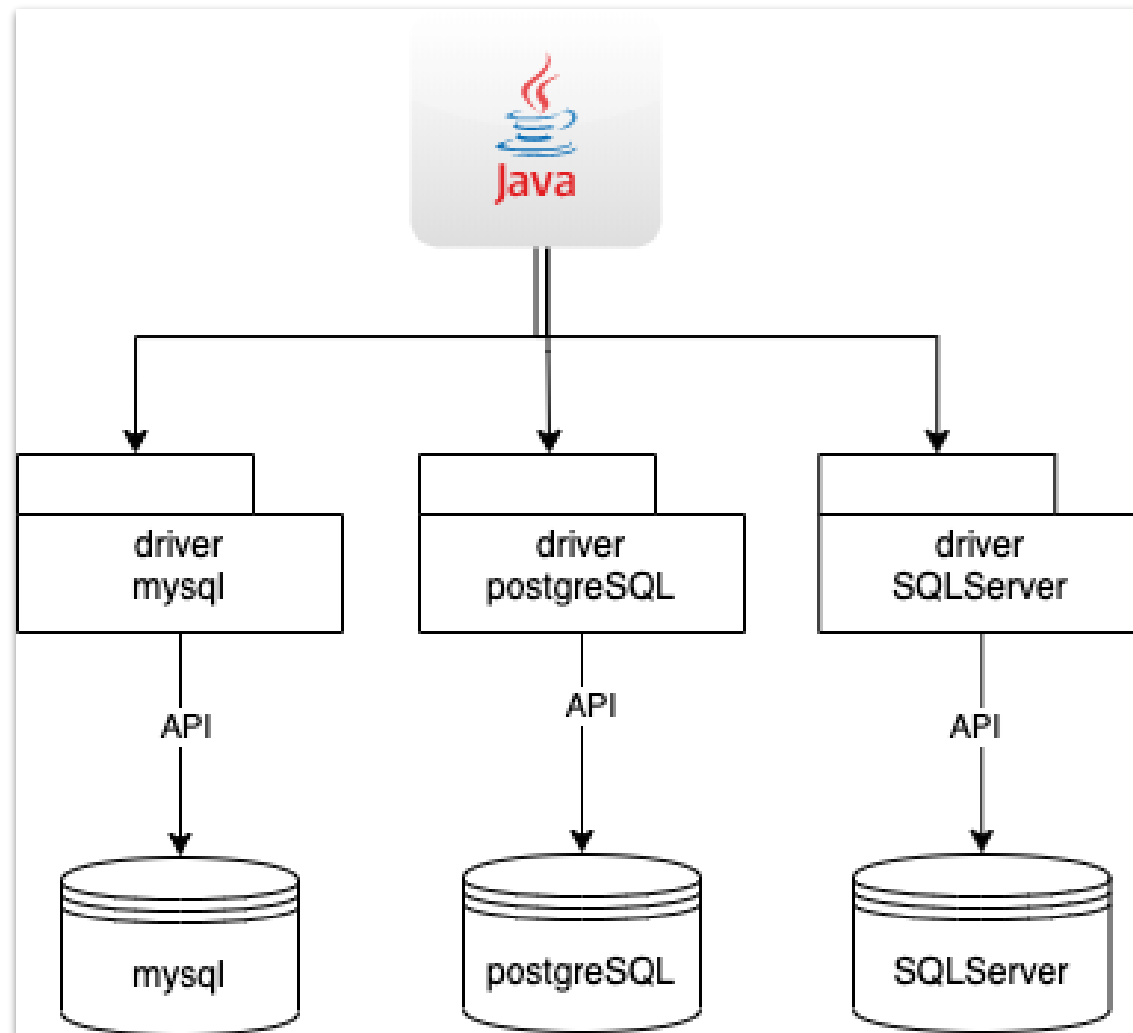
- Necesitamos **mecanismos estándar** que nos permitan **conectar** las **aplicaciones** con los **sistemas gestores de bases de datos**.
  - Cada **SGBD** dispone de **su propia API** (*Application Programming Interface*) que nos permite comunicarnos con él.
  - Los fabricantes proporcionan **drivers específicos** para cada plataforma.





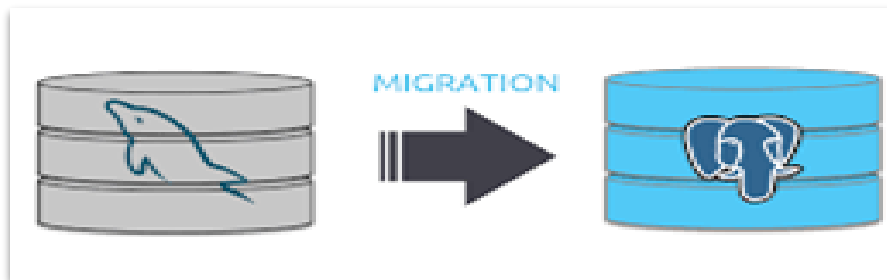
## 2. ESTABLECIMIENTO DE LA CONEXIÓN

### Esquema simple de conexión



## 2.1. ESTÁNDARES DE CONEXIÓN

- **Problema:** En la actualidad, existen una gran variedad de **SGBD** y no todos cumplen el **estándar SQL**.
  - En algunas ocasiones necesitamos **conectarnos a diferentes Sistemas** Gestores de Bases de Datos en la misma aplicación
  - A lo largo de la vida de una aplicación puede ser necesaria la **migración** de los **datos** entre **diferentes SGBD**.

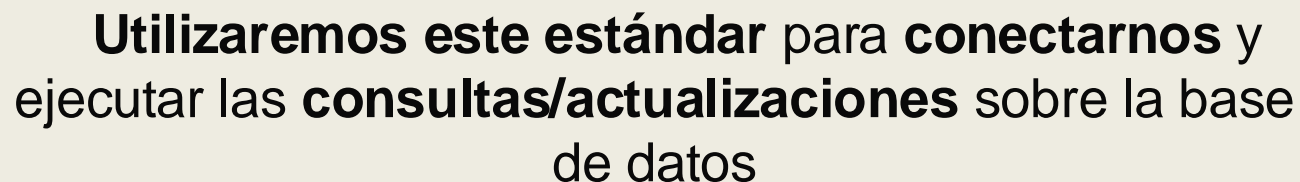


¿**Analizamos cada driver** y modificamos nuestra aplicación cada vez que se **cambie el SGBD**?

## 2.1. ESTÁNDARES DE CONEXIÓN

**Solución:** Establecer estándares para gestionar las conexiones con los **SGBD**.

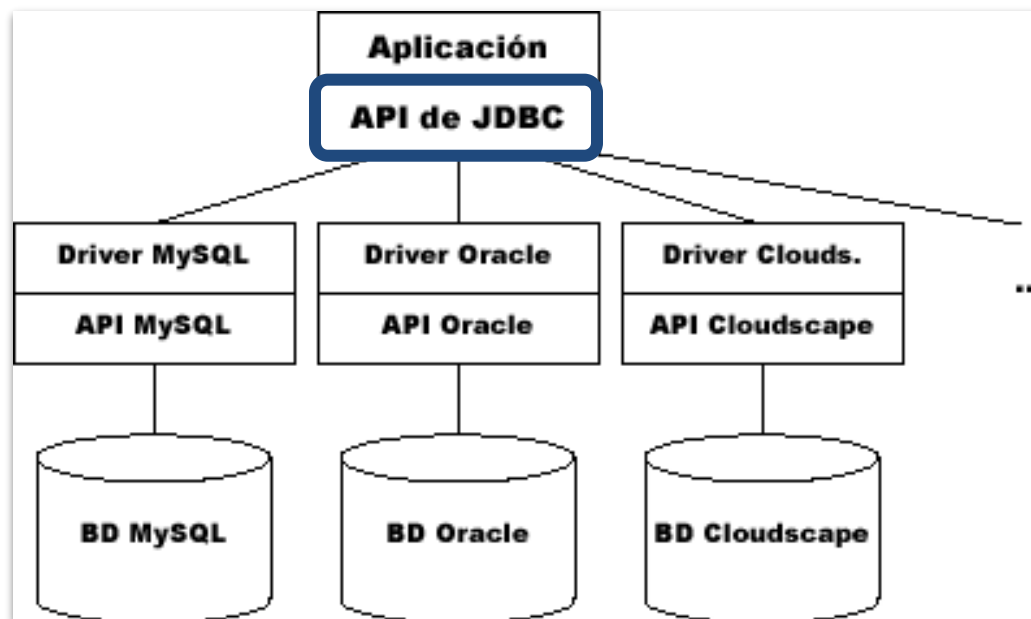
- Los **fabricantes de SGBD** y las **aplicaciones clientes sólo** necesitan **conocer** dicho **estándar** para comunicarse. Podemos destacar los siguientes:
  - **ODBC** (Open DataBase Connectivity): Estándar de acceso a base de datos desarrollado por SQL Access Group (SAG).
  - **JDBC** (Java Database Connectivity): Estándar para la conexión de base de datos en el **lenguaje Java**



Utilizaremos este estándar para **conectarnos** y ejecutar las **consultas/actualizaciones** sobre la base de datos

## 2.1.2 JDBC (Java Database Connectivity)

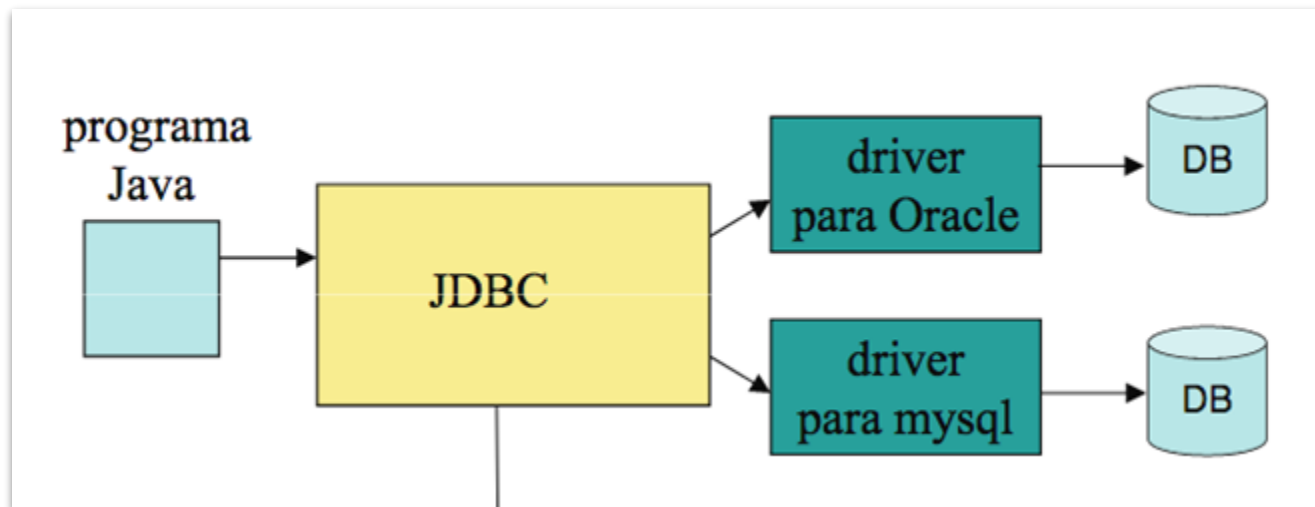
- Conjunto de interfaces y manejadores de conexión que permite a un **programa java** ejecutar **instrucciones SQL** dentro de bases de datos relacionales.
  - El **driver JDBC** instalado en el cliente **convierte** la **petición** proveniente del lenguaje **Java** al **protocolo** específico que entiende cada **SGBD**.



Cada fabricante proporciona un driver jdbc que deberá ser utilizado por el cliente

## 2.2 DRIVERS JDBC

- El JDK no incluye los drivers JDBC para llevar a cabo las conexiones con los diferentes **Sistemas Gestores de Bases de Datos**.



## 2.2 DRIVERS JDBC

- Tenemos diferentes opciones para incluir las librerías del **driver JDBC**:
  - Instalarlo a nivel de **Sistema operativo**
  - *apt-get install libpostgresql-jdbc-java libmariadb-java*
  - **Descargar las librerías** compiladas del fabricante y **enlazarlas al proyecto**.
    - <https://jdbc.postgresql.org/>
    - <https://dev.mysql.com/downloads/connector/j/>
    - <https://mariadb.com/downloads/connectors/>
- Utilizar el **gestor de dependencias** de Maven

**Utilizaremos** esta opción en nuestros **proyectos**

## 2.2 DRIVERS JDBC

- o Utilizando el **gestor de dependencias** de Maven

```
<dependency>
  <groupId>org.postgresql</groupId>
  <artifactId>postgresql</artifactId>
  <version>42.7.3</version>
</dependency>
```

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>8.0.33</version>
</dependency>
```

```
<dependency>
  <groupId>org.mariadb.jdbc</groupId>
  <artifactId>mariadb-java-client</artifactId>
  <version>3.4.0</version>
</dependency>
```

### 3. COMUNICACIÓN CON EL SGBD (JDBC)

- La API JDBC está formada por dos packages: `java.sql` y `javax.sql`.
  - `Java.sql`. Contiene las clases e interfaces esenciales: clases `Driver`, `Connection`, `Statement`, `ResultSet`, `PreparedStatement`, `CallableStatement`, principalmente. La tendremos que **importar siempre**.
  - `Javax.sql`. Clases más especializadas: `RowSet`, `DataSource`, `PooledConnection`, ..., si las necesitamos en algún **caso particular**.

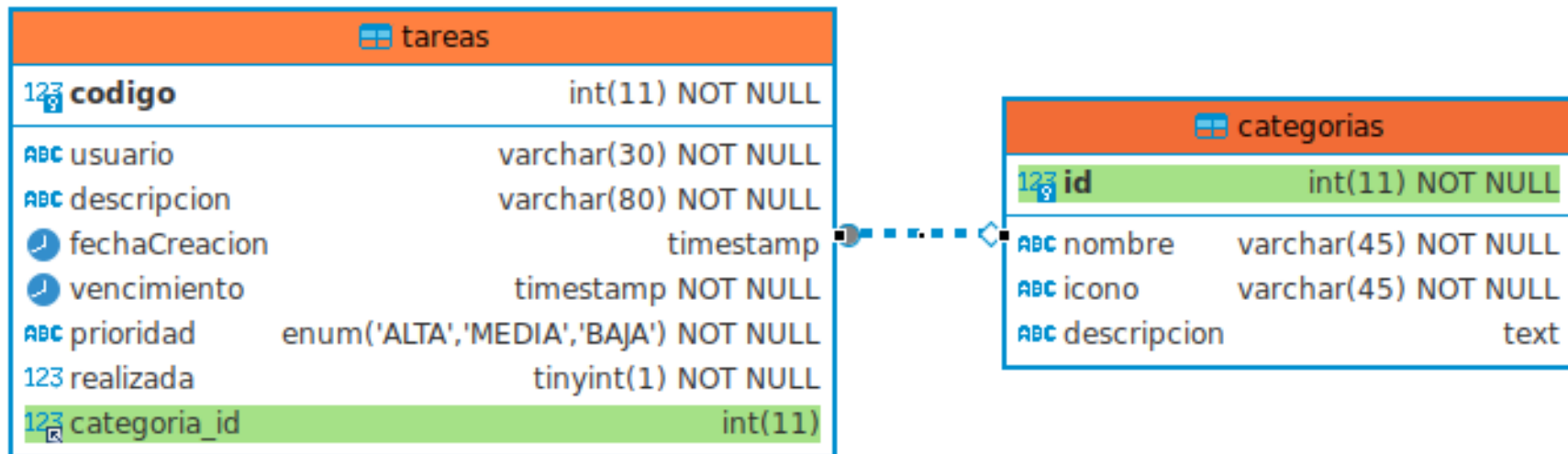


### 3. COMUNICACIÓN CON EL SGBD (JDBC)

- El acceso a la información contenida en una base de datos implica **diferentes fases**:
  1. **Seleccionar la base de datos** con la que queremos comunicar.
  2. **Establecer la conexión con la base de datos.**
  3. Llevar a cabo las **consultas de lectura / escritura.**
  4. **Cerrar la conexión:** debemos tener en cuenta que tanto el SGBD como el SO soportan un límite máximo de conexiones.

## 3.1 SELECCIONAR LA BASE DE DATOS

- La base de datos utilizada en todos los ejemplos y actividades previas de esta presentación está accesible a través de [este enlace](#).

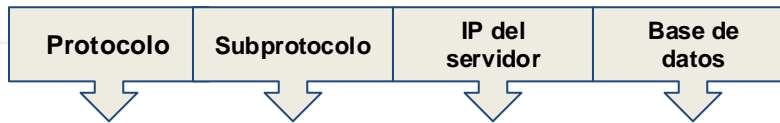


# ACTIVIDAD PREVIA

## ANTES DE EMPEZAR:

- Importa la base de datos `tareas_db` en tu SGBD
- Crea un proyecto `prog-ud12-classwork` como un proyecto *Spring Boot* (créalo igual que en la ud10, aplicando las mismas dependencias que en aquella unidad).
- Una vez creado, añade al fichero `pom.xml` la dependencia del conector para `MySQL` o `Mariadb` indicada en la diapositiva 15.

## 3.2 CONEXIÓN CON LA BASE DE DATOS



```
try {  
    String url = "jdbc:mysql://localhost/tareas_db";  
    Connection con = DriverManager.getConnection(url, "username", "password");  
} catch (SQLException e) {  
    e.printStackTrace();  
}
```

- El encargado de **abrir la conexión con el SGBD** es el `DriverManager`, a través del método `getConnection()` que requiere 3 parámetros:
  - `url`: Identifica la BD a la que nos queremos conectar y las opciones de conexión
  - `username` y `password` de conexión con suficientes privilegios para ejecutar las acciones deseadas sobre la BD.

## 3.2 EJEMPLO I . CONEXIÓN A UNA BD

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class Example1Controller {

    @GetMapping("/example1")
    @ResponseBody
    public String connectionExample() {

        try {
            String dbURL = "jdbc:mysql://localhost/tareas_db";

            Connection connection = DriverManager.getConnection(
                dbURL, "rhidalgo-web", "123456789");

            return "La conexión se ha establecido con éxito";
        } catch (SQLException e) {
            return e.getMessage();
        }
    }
}
```

Si la conexión no puede ser establecida, se lanzará una **excepción** (en este caso se informa al **usuario del error** que la ha causado)

# ACTIVIDAD PREVIA

**Actividad 1 :** Copia el código de la diapositiva anterior para establecer una conexión con la base de datos **tareas\_db**.

Deberías de ver el mensaje de confirmación



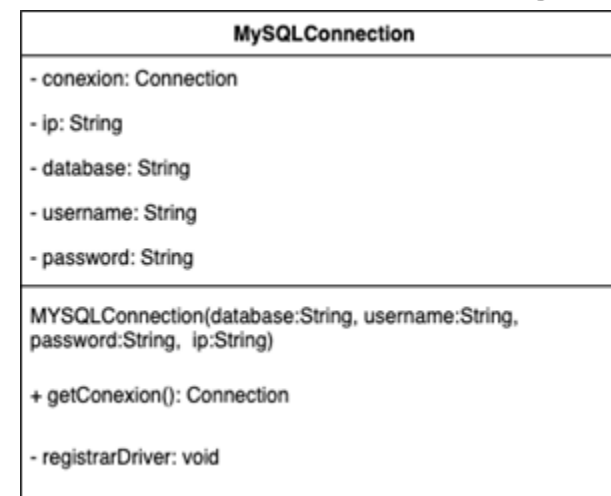
La conexión se ha establecido con éxito

## 3.3 CLASES SERVICIO @Service

- Los **servicios** son **objetos** que proporcionan algún **tipo de tarea global** que será utilizada desde diferentes partes de la aplicación y de la cual queremos que se mantenga una única instancia.
- Algunos ejemplos son: servicio de envío de mails, servicio de **conexión a una base de datos**,...

A este tipo de clases deberemos añadirle la anotación **@Service** al comienzo para poder inyectar este objeto cuando se necesite dentro de alguna clase de nuestra aplicación *Spring Boot*.

Crearemos una clase  
**MySQLConnection**  
para utilizarla cada vez  
que necesitemos  
conectarnos a la BD



## 3.3 CLASES SERVICIO

@Service

```
public class MySqlConnection {
    private static Connection connection;
    private String ip;
    private String database;
    private String userName;
    private String password;
```

Crearemos esta clase dentro de un paquete (al mismo nivel que controllers) llamado **services**

```
public MySqlConnection() {
    this.ip = "127.0.0.1";
    this.database = "tareas_db";
    this.userName = "batoi";
    this.password = "1234";
}
```

Estos datos se deben adaptar

```
public Connection getConnection() {
    if (connection == null || connection.isClosed()) {
        try {
            String dbURL = "jdbc:mysql://" + ip + "/" + database;
            Connection connection = DriverManager.getConnection(dbURL, userName, password);
            MySqlConnection.connection = connection;
            System.out.println("Conexión válida: " + connection.isValid(20));
        } catch (SQLException exception) {
            throw new RuntimeException(exception.getMessage());
        }
    }
    return MySqlConnection.connection;
}
```

Creamos una conexión solo si esta no se ha establecido anteriormente

Podría ser interesante que nuestra clase tenga un método para cerrar la conexión y, además, que implemente la interfaz Closeable para poder usarla en un try-with-resources



## 4. EJECUCIÓN DE SENTENCIAS. La clase `Statement`

---

- La clase `java.sql.Statement` representa una sentencia que queremos ejecutar con la base de datos. Podemos diferenciar:
  - Sentencias de **consulta** (`SELECT`)
  - Sentencias DML: de **manipulación de datos** (`INSERT`, `UPDATE`, `DELETE`)
  - Sentencias DDL: de **manipulación de la estructura** (`CREATE TABLE`, `DROP TABLE`, `ALTER TABLE...`)

## 4. EJECUCIÓN DE SENTENCIAS. La clase Statement

- Los objetos `Statement` los crea el objeto de tipo `Connection` una vez establecida la conexión.
- Hay que tener en cuenta que el objeto `Statement` crea una apertura de recursos que hay que cerrar (haremos uso de *try-with-resources*)

```
//Establecemos la conexión
Connection connection = MySQLConnection.getConnection();

try (//Creamos el objeto statement que nos permite ejecutar una sentencia
    Statement statement = connection.createStatement();
    ) {
    ...
} catch (SQLException e) {
    return e.getMessage();
}
```



## 4. EJECUCIÓN DE SENTENCIAS. La clase `Statement`

---

- La clase `Statement` dispone de los métodos:
  - `ResultSet executeQuery(String sql)`: Permite la ejecución de sentencias de consulta.
    - Devuelve un `ResultSet` con la **información del resultado** de la **consulta** que deberemos procesar.
  - `int executeUpdate(String sql)`: Permite la ejecución de sentencias de manipulación (DML y DDL).
    - Devuelve el **número de registros afectados** en la consulta.

## 4.2 EJECUCIÓN DE CONSULTAS. La clase `ResultSet`

- `ResultSet executeQuery(String sql)`: permite la ejecución de sentencias de consulta. El objeto `ResultSet` que devuelve también debe cerrarse una vez finalizado su uso.

```
//Establecemos la conexión
Connection connection = MySqlConnection.getConnection();

try (
    // Creamos un objeto que representa la sentencia a ejecutar
    Statement statement = connection.createStatement();
    // Ejecutamos la sentencia y obtenemos el resultado
    ResultSet rs = statement.executeQuery("SELECT * FROM tareas");
)
{
    . . .
} catch (SQLException e) {
    e.printStackTrace();
}
```

## 4.2 EJECUCIÓN DE CONSULTAS. La clase `ResultSet`

- La clase `ResultSet` representa un conjunto de resultados (**tuplas**) obtenidas tras la ejecución de una consulta.
  - Para poder acceder a todas las filas deberemos de **iterar** sobre el **conjunto de tuplas / filas resultantes**. `ResultSet` ofrece un conjunto de métodos para recorrer sus tuplas. El más usado es `next()`.

```
try (  
    // Creamos un objeto que representa la sentencia a ejecutar  
    Statement statement = connection.createStatement();  
    // Ejecutamos la sentencia y obtenemos el resultado  
    ResultSet rs = statement.executeQuery("SELECT * FROM tareas");  
) {  
    // Accedemos a las diferentes filas  
    while (rs.next()) {  
        String descripcion = rs.getString("descripcion");  
        System.out.println("descripción = " + descripcion);  
        String descripcion2 = rs.getString(1);  
        System.out.println("descripción = " + descripcion2);  
    }  
} catch (SQLException e) {  
    return e.getMessage();  
}
```

Acceso mediante nombre al campo  
"descripcion" del registro

Accedemos mediante índice: primer  
campo.

## 4.2 EJECUCIÓN DE CONSULTAS. La clase `ResultSet`

- En función del tipo de datos de la columna, llamaremos a un método u otro del `ResultSet` para recuperar su valor.

Tipo estándar SQL	Método
CHAR	<code>getString("nombre   indice_Columna")</code>
VARCHAR	<code>getString("nombre   indice_Columna")</code>
SMALL	<code>getShort("nombre   indice_Columna")</code>
INTEGER	<code>getInt("nombre   indice_Columna")</code>
FLOAT	<code>getFloat ("nombre   indice_Columna")</code> <code>getDouble ("nombre/indice_Columna")</code>
DOUBLE	<code>getDouble("nombre   indice_Columna")</code>
DECIMAL	<code>getBigDecimal("nombre   indice_Columna")</code>
DATE	<code>getDate("nombre   indice_Columna").toLocalDate()</code>
TIMESTAMP	<code>getTimeStamp("nombre   indice_Columna").toLocalDateTime()</code>

## 4.2 EJECUCIÓN DE CONSULTAS. La clase `ResultSet`

- Para conseguir ejecutar una **sentencia SQL** con **valores dinámicos**, **debemos** concatenar las **partes fijas** de la sentencia con las **partes cambiantes** (variables).

```
String sql = "SELECT * FROM tareas WHERE codigo=" + idBuscado + "  
OR usuario = '" + usuarioBuscado + "'";
```

```
String sql2 = String.format("SELECT * FROM tareas WHERE codigo = %d  
OR usuario = '%s'", idBuscado, usuarioBuscado);
```

```
Statement statement = connection.createStatement();  
ResultSet rs = statement.executeQuery(sql);  
ResultSet rs2 = statement.executeQuery(sql2);
```

- Argumentos numéricos deben ir sin comillas.
- Argumentos de tipo cadena deben ir entre comillas simples `'nombre'`.

## 4.3 EJEMPLO II. EJECUCIÓN DE CONSULTA

```

@Controller
public class Example2Controller {

    @Autowired
    private MySQLConnection mySQLConnection;

    @GetMapping("/example2")
    @ResponseBody
    public String selectExample() {
        Connection connection = mySQLConnection.getConnection();

        String sql = "SELECT descripcion, realizada FROM tareas WHERE usuario = 'Juan'";

        try (
            Statement statement = connection.createStatement();
            ResultSet rs = statement.executeQuery(sql);
        ){

            StringBuilder respuesta = new StringBuilder();

            while (rs.next()) {
                respuesta.append("<p>-----</p>");
                respuesta.append("<p>Descripción: " + rs.getString("descripcion"));
                respuesta.append("<br/>Realizada: " +
                    (rs.getInt("realizada") == 0?"No":"Sí") + "</p>");
            }
            return respuesta.toString();
        } catch (SQLException e) {
            return e.getMessage();
        }
    }
}

```

Ejecutamos la  
sentencia

Accedemos a la  
información de  
los registros  
obtenidos

Consulta



# ACTIVIDAD PREVIA

---

**Actividad 2 :** Crea un controlador `Actividad2Controller` con un método `consultasActividad2` asociado a un endpoint `/actividad2` y, a partir del ejemplo de la diapositiva anterior, realiza el código necesario para obtener la siguiente información:

- Mostrar por pantalla el **código** y la **descripción** de todas las **tareas**.
- Mostrar por pantalla todos los **nombres** de las categorías
- Mostrar por pantalla la **descripción**, la **fecha de creación (en formato dd/mm/yyyy)**, si ha sido **realizada** o no (mostrando sí o no) de todas las **tareas** de **categoría 1 o 2**.

## 5. SENTENCIAS DE ACTUALIZACIÓN

- Permite **manipular la información** almacenada en la **base de datos** mediante la **ejecución de sentencias de actualización** como son: INSERT, UPDATE, DELETE
  - A diferencia de las **sentencias anteriores**, **no devuelven registros**. **Devuelve el número de filas** que han sido **afectadas por la actualización**.
  - **Utilizamos el método** `executeUpdate(String sql)` de la clase `Statement`

```
try (Statement statement = connection.createStatement()) {  
    int affectedRows = statement.executeUpdate(sql);  
    System.out.println("Filas modificadas: " + affectedRows);  
} catch (SQLException e) {  
    e.printStackTrace();  
}
```

## 5.1 EJEMPLO III. SENTENCIA DE ACTUALIZACIÓN

```
@Controller
public class Example3Controller {

    @Autowired
    private MySqlConnection mySqlConnection;

    @GetMapping("/example3")
    @ResponseBody
    public String exampleInsertNewCategory() {
        String sql = "INSERT INTO categorias (id, nombre, icono) "
            + "VALUES (6, 'Trabajo', 'work.png)";

        Connection connection = mySqlConnection.getConnection();
        try (Statement statement = connection.createStatement())
        {
            int affectedRows = statement.executeUpdate(sql);
            return affectedRows + " nueva tarea insertada.";
        } catch (SQLException e) {
            return e.getMessage();
        }
    }
}
```

La ejecución de la sentencia devuelve el número de filas afectadas. En este caso: 1 fila afectada.

# ACTIVIDAD PREVIA

**Actividad 3:** Crea un controlador `Actividad3Controller` con un método `insertaTareaPorDefecto` asociado a un endpoint `/tarea-default-add` que reciba, a través de una petición GET **la descripción y el usuario** de una nueva tarea a insertar en la base de datos.

El resto de los datos de la tarea que no se proporcionan se establecen a los siguientes valores por defecto: **no tendrá categoría asociada, no estará realizada, prioridad baja, la fecha de creación será la fecha del sistema y la de vencimiento será dos días después.**

## NOTAS DE AYUDA

- Para transformar un objeto `LocalDateTime` a formato `Timestamp` (formato que acepta la base de datos) sólo hay que usar el método `toString()` del objeto.

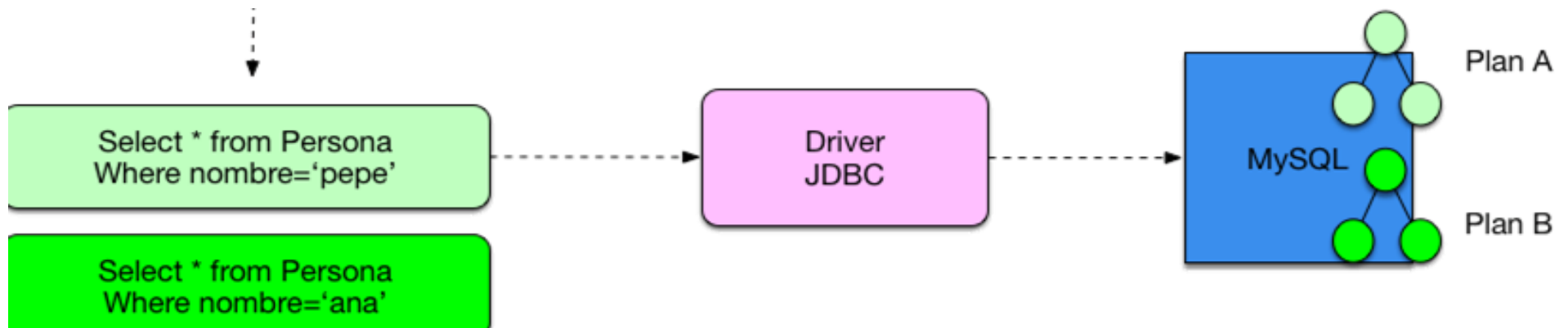
Por ejemplo:

`LocalDateTime.now().toString()` nos devuelve el String `2024-05-20 20:34:27`

- El navegador acepta que en los parámetros se inserten espacios en blanco (automáticamente los sustituye por un `%20`)

## 4. SENTENCIAS PREPARADAS: PreparedStatement

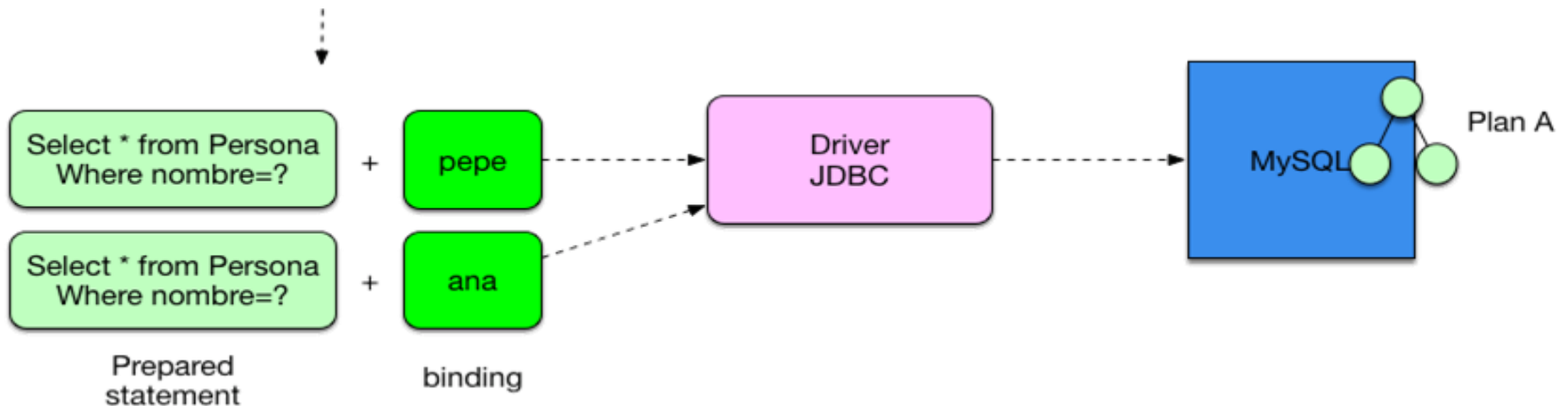
- Alternativa al uso de la clase `Statement` para llevar a cabo consultas al SGBD **optimizadas**.
  - En muchas ocasiones necesitamos ejecutar la **misma sentencia** con **diferentes parámetros**.
  - Para cada **sentencia** que ejecutamos **sobre la base de datos** se construye **un plan de ejecución** en el SGBD.



## 4. SENTENCIAS PREPARADAS: PreparedStatement

- Las **sentencias preparadas** permiten definir una **sentencia SQL parametrizada** a partir de **placeholders** (las interrogaciones), de forma que puedan **compartir** el mismo **plan de ejecución**.

```
String sql = "SELECT * FROM tareas WHERE codigo = ?";
```



## 4. EJEMPLO 4. SENTENCIAS PREPARADAS

```
@GetMapping("/example4")
@ResponseBody
public String example4InsertWithPreparedStatement() {

    StringBuilder respuesta = new StringBuilder();

    Connection connection = mySQLConnection.getConnection();

    String sql = "INSERT INTO categorias (id,nombre,icono) VALUES (?,?,?)";

    try (PreparedStatement preparedStatement = connection.prepareStatement(sql)) {
        preparedStatement.setInt(1, 7);
        preparedStatement.setString(2, "Juegos");
        preparedStatement.setString(3, "play.png");
        int affectedRows = preparedStatement.executeUpdate();
        respuesta.append(affectedRows + " nueva categoría insertada");

        preparedStatement.setInt(1, 8);
        preparedStatement.setString(2, "Otros");
        preparedStatement.setString(3, "other.png");
        affectedRows = preparedStatement.executeUpdate();
        respuesta.append(affectedRows + " nueva categoría insertada");

        return respuesta.toString();
    } catch (SQLException e) {
        return e.getMessage();
    }
}
```

**¡¡Ojo!!**

Número de  
campo: empieza  
desde 1

← INSERCIÓN 1

← INSERCIÓN 2

## 4. SENTENCIAS PREPARADAS: `PreparedStatement`

**Algunos métodos** para dar valores a cada uno de los **placeholders** son:

- ❖ `setString()`
- ❖ `setBoolean()`
- ❖ `setInt()`
- ❖ `setDate()`
- ❖ `setTime()`
- ❖ `setTimestamp()`
- ❖ `setNull(XXX, Types.Integer)`
- ❖ ...

Como siempre, puedes **consultar** todos los **métodos** de la **API** en la [documentación oficial](#)



## 4. SENTENCIAS PREPARADAS: PreparedStatement

A continuación, se presentan **alguno de los ejemplos** de utilización de los métodos para dar valor a los diferentes **placeholders** en función del tipo de datos:

```
ps.setNull(X, Types.INTEGER); //Valor nulo
ps.setInt(X, tarea.getCodigo()); //Valor entero
ps.setString(X, tarea.getUsuario()); //Valor cadena
ps.setBoolean(X, tarea.isRealizada()); //Valor booleano
ps.setTimestamp(X, Timestamp.valueOf(tarea.getCreatedOn())); //Valor LocalDateTime
ps.setDate(X, Date.valueOf(user.getBirthday())); //Valor LocalDate
ps.setTime(X, Time.valueOf(reloj.getTime)); //Valor LocalTime
```

Tipo de dato del valor que espera el método

Paquete java.sql

## 4.1 SENTENCIAS PREPARADAS. Seguridad

- **También nos protegen** frente a posibles **inyecciones SQL**. Se trata de un ataque por el cual nos permite **insertar código SQL** en una **sentencia**.
  - Imaginemos que queremos ejecutar esta sentencia donde nombre es un parámetro que **inserta el usuario** desde **la consola** o interfaz gráfica:

```
String sql = "SELECT * FROM tareas WHERE categoria = " + categoria;
```

- Si el **usuario inserta** para el valor de **nombre** "1;drop table tareas"  
Estaría ejecutando la siguiente sentencia:

```
SELECT * FROM tareas WHERE categoria = 1; drop table tareas
```

Las **sentencias preparadas protegen** frente a este tipo de **ataques de inyección**. Aunque esta ejecución no **funcionaría, sirva este ejemplo para entender el problema**

# ACTIVIDADES PREVIAS

---

**Actividad 4:** Crea un nuevo controlador `Actividad4Controller` que, a partir del creado en la actividad 3, ahora haga uso de sentencias preparadas

**Actividad 5:** Realiza un nuevo controlador `/tarea-description-update` que actualice una tarea, haciendo uso de sentencias preparadas, a partir de su código y de una nueva descripción.

## 4.2 SENTENCIAS PREPARADAS: Auto-incremental

- En algunas ocasiones necesitamos **recuperar el valor auto-numérico** asignado por la base de datos después de una inserción.

```

@GetMapping("/examples")
@ResponseBody
public String example5InsertWithPreparedStatementAndNewKeyInfo() {

    Connection connection = mySQLConnection.getConnection();

    String sql = "INSERT INTO categorias (nombre,icono, descripcion) VALUES (?,?,?)";

    try (PreparedStatement preparedStatement = connection.prepareStatement(
        sql, PreparedStatement.RETURN_GENERATED_KEYS) {

        preparedStatement.setString (1, "Administrativas");
        preparedStatement.setString (2, "admin.png");
        preparedStatement.setString (3, "Tareas como renovar el dni, presentar la renta.");

        int affectedRows = preparedStatement.executeUpdate();

        try (ResultSet rs = preparedStatement.getGeneratedKeys()) {
            if (rs.next()) {
                int autoIncremental = rs.getInt(1);
                StringBuilder respuesta = new StringBuilder();

                respuesta.append("Se ha creado " + affectedRows
                    + " nueva tarea con id " + autoIncremental);

                return respuesta.toString();
            }
        }

        return "No se ha creado una nueva categoría";

    } catch (SQLException e) {
        return e.getMessage();
    }
}

```

**NO**  
INDICAMOS el campo  
ID porque queremos que  
se lo asigne el SGBD

Indicamos  
que  
queremos  
recuperar el  
id asignado.

Obtenemos el id (auto-  
incremental) asignado, que  
se recupera de un  
ResultSet que produce el  
método getGeneratedKeys

# ACTIVIDAD PREVIA

---

**Actividad 6:** Crea un nuevo controlador `Actividad6Controller` copia del creado en la actividad 4, que muestre por pantalla una vez insertada la tarea, el mensaje:

Nueva tarea con id *X* insertada correctamente

- Esto es todo, de momento... :-)