

Guía instalación de Docker, imagen y contenedor de MySQL, más comandos básicos

Un poco de teoría.

1- ¿Qué es Docker?

Docker es una plataforma que permite crear, empaquetar y ejecutar aplicaciones en un entorno aislado llamado contenedor. Los contenedores son como "cajas" ligeras y portables que contienen todo lo necesario para ejecutar una aplicación: código, dependencias, bibliotecas, y configuraciones.

¿Para qué sirve?

- Portabilidad: Puedes ejecutar el mismo contenedor en tu máquina local, un servidor o en la nube sin necesidad de configuraciones adicionales.
- Aislamiento: Cada aplicación corre en su propio contenedor, lo que evita conflictos entre dependencias de diferentes aplicaciones.
- Escalabilidad: Facilita el despliegue y escalado de aplicaciones en ambientes de producción.

Ejemplo de uso:

- Empaquetar una aplicación web con su servidor, base de datos y todas las dependencias en contenedores separados.
- Probar una aplicación en diferentes versiones de Java o Node.js sin afectar tu sistema.

2- Conceptos clave: Imagen y Contenedor

Relación entre imagen y contenedor

Una imagen es el plano o molde, mientras que un contenedor es una instancia de ese plano.

Piensa en una imagen como la receta de un pastel.

El contenedor es el pastel hecho a partir de esa receta.

Una imagen no puede "usarse" directamente; es solo un archivo estático que contiene el software y sus dependencias. Para "usar" la imagen, debes convertirla en un contenedor.

Un contenedor no existe sin una imagen. Siempre se crea a partir de una imagen.

Ejemplo práctico

Supongamos que estás trabajando con una imagen oficial de MySQL (mysql:8):

La imagen:

Contiene todo lo necesario para ejecutar MySQL: sistema base, binarios de MySQL, configuraciones predeterminadas, etc.

Es un archivo inmutable almacenado en tu máquina o descargado desde Docker Hub.

No puedes interactuar con la imagen directamente.

El contenedor:

Cuando usas la imagen mysql:8 para crear un contenedor, este arranca un servidor MySQL en tu máquina.

Puedes interactuar con el contenedor (por ejemplo, acceder a la base de datos, cambiar configuraciones, añadir datos).

El contenedor es "activo" y puede guardar estado mientras esté corriendo.

3- Instalar Docker en Ubuntu (moderna con Docker Compose)

Paso 1: Actualizar el sistema y preparar las dependencias

Actualiza los paquetes disponibles:

```
sudo apt-get update
```

Instala los paquetes necesarios para Docker:

```
sudo apt-get install -y ca-certificates curl gnupg
```

Crea un directorio para almacenar claves de repositorios:

```
sudo install -m 0755 -d /etc/apt/keyrings
```

Descarga y configura la clave GPG oficial de Docker:

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg
```

Asegúrate de que la clave sea legible:

```
sudo chmod a+r /etc/apt/keyrings/docker.gpg
```

Paso 2: Configurar el repositorio de Docker

Añade el repositorio de Docker:

```
echo \n"deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg] \nhttps://download.docker.com/linux/ubuntu \ $(lsb_release -cs) stable" | sudo tee \n/etc/apt/sources.list.d/docker.list > /dev/null
```

Actualiza los paquetes para incluir Docker:

```
sudo apt-get update
```

Paso 3: Instalar Docker y plugins modernos

Instala Docker junto con los plugins de buildx y docker-compose:

```
sudo apt-get install -y docker-ce docker-ce-cli containerd.io docker-buildx-plugin  
docker-compose-plugin
```

Verifica que Docker se haya instalado correctamente:

```
docker --version
```

Verifica que Docker Compose esté disponible:

```
docker compose version
```

Paso 4: Ejecutar Docker sin usar sudo (opcional)

Añade tu usuario al grupo de Docker:

```
sudo usermod -aG docker $USER
```

Cierra sesión y vuelve a iniciarla para aplicar los cambios.

Comprobar que todo funciona

Ejecuta el contenedor de prueba hello-world para validar la instalación:

```
docker run hello-world
```

Si ves un mensaje de éxito, Docker está funcionando correctamente.

4- Descargar e iniciar un contenedor de MySQL

Descargar la imagen oficial de MySQL:

```
docker pull mysql:8
```

Esto descargará la última versión de MySQL 8.

Crear y ejecutar un contenedor de MySQL: Usa el comando docker run para crear y ejecutar un contenedor basado en la imagen de MySQL:

```
docker run -d --name mi_mysql -e MYSQL_ROOT_PASSWORD=tu_contraseña -p  
3306:3306 mysql:8
```

-d: Ejecuta el contenedor en segundo plano (modo "detached").

--name mi_mysql: Asigna un nombre al contenedor.

-e MYSQL_ROOT_PASSWORD=tu_contraseña: Establece la contraseña del usuario root de MySQL.

-p 3306:3306: Expone el puerto 3306 del contenedor en el puerto 3306 del host.

Verificar que el contenedor esté en ejecución:

```
docker ps
```

Deberías ver el contenedor `mi_mysql` en la lista.

[Conectarte a MySQL dentro del contenedor](#)

Abrir una consola dentro del contenedor:

```
docker exec -it mi_mysql mysql -u root -p
```

Introduce la contraseña configurada en el paso anterior (`tu_contraseña`).

Usar MySQL normalmente: Una vez dentro, puedes crear bases de datos, tablas y ejecutar consultas como lo harías en una instalación local.

[Gestión del contenedor](#)

Detener el contenedor:

```
docker stop mi_mysql
```

Reiniciar el contenedor:

```
docker start mi_mysql
```

Eliminar el contenedor (si ya no lo necesitas):

```
docker rm -f mi_mysql
```

Ver todas las imágenes disponibles:

```
docker images
```

5- Guía completa de comandos básicos MYSQL

[Conectarte a MySQL](#)

Abrir MySQL desde la terminal del contenedor:

```
mysql -u root -p
```

Introduce la contraseña de root.

Conectarte a una base de datos existente:

```
USE nombre_base_datos;
```

[Bases de datos](#)

Crear una base de datos:

```
CREATE DATABASE nombre_base_datos;
```

Ver todas las bases de datos:

```
SHOW DATABASES;
```

Eliminar una base de datos:

```
DROP DATABASE nombre_base_datos;
```

Usuarios y permisos

Crear un nuevo usuario:

```
CREATE USER 'nombre_usuario'@'localhost' IDENTIFIED BY 'contraseña';
```

Dar todos los privilegios al usuario:

```
GRANT ALL PRIVILEGES ON *.* TO 'nombre_usuario'@'localhost';
```

```
FLUSH PRIVILEGES;
```

Revocar privilegios de un usuario:

```
REVOKE ALL PRIVILEGES ON *.* FROM 'nombre_usuario'@'localhost';
```

Eliminar un usuario:

```
DROP USER 'nombre_usuario'@'localhost';
```

Significado de *.*

- El primer asterisco (*) representa todas las bases de datos en el servidor MySQL.
- El segundo asterisco (*) representa todas las tablas dentro de esas bases de datos.

En conjunto, *.* significa todas las tablas en todas las bases de datos del servidor MySQL.

Ejemplo práctico:

- Si usas **GRANT ALL PRIVILEGES ON *.***, estás otorgando permisos sobre todo el servidor.
- Si usas algo como **GRANT SELECT ON mi_base.***, solo otorgas permisos a todas las tablas dentro de la base de datos mi_base.

Tablas

Crear una tabla básica:

```
CREATE TABLE empleados (  
    id INT AUTO_INCREMENT PRIMARY KEY,  
    nombre VARCHAR(50),  
    salario DECIMAL(10,2),  
    fecha_contratacion DATE  
);
```

Ver las tablas de la base de datos:

```
SHOW TABLES;
```

Eliminar una tabla:

```
DROP TABLE nombre_tabla;
```

Ver la estructura de una tabla:

```
DESCRIBE nombre_tabla;
```

Eliminar todos los registros de una tabla:

```
TRUNCATE TABLE empleados;
```

Claves foráneas

Crear una tabla con clave foránea:

```
CREATE TABLE departamentos (  
    id INT AUTO_INCREMENT PRIMARY KEY,  
    nombre VARCHAR(50)  
);
```

```
CREATE TABLE empleados (  
    id INT AUTO_INCREMENT PRIMARY KEY,  
    nombre VARCHAR(50),  
    departamento_id INT,  
    FOREIGN KEY (departamento_id) REFERENCES departamentos(id)  
);
```

Consultas básicas

Insertar datos:

```
INSERT INTO empleados (nombre, salario, fecha_contratacion)  
VALUES ('Juan Perez', 50000.00, '2024-12-12');
```

Seleccionar datos:

```
SELECT * FROM empleados;
```

Actualizar datos:

```
UPDATE empleados
```

```
SET salario = 55000.00
```

```
WHERE nombre = 'Juan Perez';
```

Eliminar datos:

```
DELETE FROM empleados
```

```
WHERE nombre = 'Juan Perez';
```

Filtrar datos con condiciones:

```
SELECT * FROM empleados
```

```
WHERE salario > 30000;
```

Ordenar resultados:

```
SELECT * FROM empleados
```

```
ORDER BY salario DESC;
```

Agrupar resultados:

```
SELECT departamento_id, AVG(salario) AS salario_promedio
```

```
FROM empleados
```

```
GROUP BY departamento_id;
```

Constraints (Restricciones)

Agregar una restricción NOT NULL:

```
ALTER TABLE empleados MODIFY nombre VARCHAR (50) NOT NULL;
```

Agregar una clave única:

```
ALTER TABLE empleados ADD CONSTRAINT unico_nombre UNIQUE (nombre);
```

Restricción de valor:

```
CREATE TABLE ejemplo (  
    id INT PRIMARY KEY,  
    edad INT CHECK (edad >= 18)  
);
```

Tipos de datos comunes

Numéricos:

- **INT** (Números enteros).
- **DECIMAL (M, D)** (Números decimales con precisión).
- **FLOAT o DOUBLE** (Números en coma flotante).

Cadenas:

- **VARCHAR(M)** (Texto de longitud variable).
- **CHAR(M)** (Texto de longitud fija).

Fechas:

- **DATE** (Solo fecha: AAAA-MM-DD).
- **DATETIME** (Fecha y hora: AAAA-MM-DD HH:MM: SS).

Consultas avanzadas

Join (Unir tablas):

```
SELECT empleados.nombre, departamentos.nombre AS departamento
FROM empleados
JOIN departamentos ON empleados.departamento_id = departamentos.id;
```

Subconsultas:

```
SELECT nombre
FROM empleados
WHERE salario > (SELECT AVG (salario) FROM empleados);
```

Comparación de tipos de JOINS:

Operación en Oracle (+)	Equivalente en MySQL	Descripción
= columna1 = columna2(+)	LEFT JOIN	Todas las filas de la tabla izquierda más las coincidentes de la derecha.
= columna1(+)= columna2	RIGHT JOIN	Todas las filas de la tabla derecha más las coincidentes de la izquierda.
-	FULL OUTER JOIN	Todas las filas de ambas tablas, con o sin coincidencias (requiere emulación en MySQL).
Sin (+)	INNER JOIN	Solo las filas que tienen coincidencias en ambas tablas.

6- Guía conectores para Bases de Datos Relacional

Funcionamiento JDBC



Clase DriverManager

DriverManager es una clase que gestiona los controladores JDBC (drivers) para conectar aplicaciones Java a bases de datos específicas. Es responsable de encontrar y establecer la conexión adecuada con una base de datos utilizando un controlador compatible.

Métodos más usados de DriverManager

Método:

[getConnection\(String url\)](#)

Uso: Establece una conexión con la base de datos utilizando solo la URL de la base de datos.

Ejemplo:

```
Connection connection =  
DriverManager.getConnection("jdbc:mysql://localhost:3306/mi_base");
```

Método:

getConnection(String url, String user, String password)

Uso: Establece una conexión con la base de datos especificando también el usuario y la contraseña.

Ejemplo:

```
Connection connection = DriverManager.getConnection(
    "jdbc:mysql://localhost:3306/mi_base", "mi_usuario", "mi_contraseña"
);
```

Método:

getConnection(String url, Properties info)

Uso: Permite establecer la conexión utilizando un objeto Properties para pasar configuraciones adicionales, como usuario, contraseña, y otros parámetros.

Ejemplo:

```
Properties props = new Properties();
props.put("user", "mi_usuario");
props.put("password", "mi_contraseña");

Connection connection =
    DriverManager.getConnection("jdbc:mysql://localhost:3306/mi_base", props);
```

Clase Connection

Connection representa una conexión activa con una base de datos. A través de esta clase, puedes ejecutar consultas, gestionar transacciones, y cerrar la conexión.

Métodos más usados de Connection:

Método:

createStatement()

Uso: Crea un objeto Statement para ejecutar sentencias SQL.

Ejemplo:

```
Statement stmt = connection.createStatement();

ResultSet rs = stmt.executeQuery("SELECT * FROM clientes");
```

Método:

prepareStatement(String sql)

Uso: Crea un objeto PreparedStatement para consultas precompiladas con parámetros.

Ejemplo:

```
PreparedStatement ps = connection.prepareStatement("SELECT * FROM clientes  
WHERE id = ?");  
  
ps.setInt(1, 1);  
  
ResultSet rs = ps.executeQuery();
```

Método:

close()

Uso: Cierra la conexión con la base de datos. Siempre debe llamarse al finalizar el uso de la conexión.

Ejemplo:

```
connection.close();
```

Método:

setAutoCommit(boolean autoCommit)

Uso: Habilita o deshabilita el modo de autocommit para transacciones. Si está en false, debes confirmar manualmente las transacciones.

Ejemplo:

```
connection.setAutoCommit(false);
```

Método:

commit()

Uso: Confirma todas las transacciones pendientes realizadas en la conexión.

Ejemplo:

```
connection.commit();
```

Método:

rollback()

Uso: Deshace todas las transacciones pendientes realizadas en la conexión desde el último commit.

Ejemplo:

```
connection.rollback();
```

Método:

getMetaData()

Uso: Obtiene información sobre la base de datos, como las tablas, columnas, y versiones.

Ejemplo:

```
DatabaseMetaData metaData = connection.getMetaData();  
System.out.println("Base de datos: " + metaData.getDatabaseProductName());
```

Método:

isClosed()

Uso: Verifica si la conexión está cerrada.

Ejemplo:

```
if (connection.isClosed()) {  
    System.out.println("La conexión está cerrada.");  
}
```

Clase Statement

Qué es: Es una interfaz de JDBC que permite ejecutar sentencias SQL estáticas en la base de datos.

Uso: Se usa cuando la consulta SQL no necesita parámetros dinámicos.

Ejemplo:

```
Statement stmt = conexion.createStatement();  
ResultSet rs = stmt.executeQuery("SELECT * FROM empleados");
```

Limitaciones:

- No es eficiente para consultas repetitivas.
- Vulnerable a inyección SQL porque no soporta parámetros.

Clase PreparedStatement

Qué es: Es una subclase de Statement que permite precompilar consultas SQL con parámetros dinámicos.

Uso: Se usa cuando se necesitan consultas con parámetros o cuando se ejecutan las mismas consultas repetidamente.

Ejemplo:

```
PreparedStatement pstmt = conexion.prepareStatement("SELECT * FROM  
empleados WHERE id = ?");  
  
pstmt.setInt(1, 1);  
  
ResultSet rs = pstmt.executeQuery();
```

Ventajas:

Más seguro: Protege contra inyección SQL.

Mejor rendimiento: Las consultas se precompilan.

Soporte para parámetros dinámicos.

Clase ResultSet

Qué es: Es una interfaz que representa los resultados de una consulta SQL.

Uso: Se utiliza para leer los datos devueltos por una consulta, fila por fila.

Ejemplo:

```
while (rs.next()) {  
    System.out.println(rs.getString("nombre"));  
}
```

Tipos de navegación:

Por defecto, es unidireccional y de solo lectura.

Se puede hacer bidireccional y actualizable configurándolo en Statement.

Método executeQuery

Qué es: Un método de Statement o PreparedStatement para ejecutar consultas SQL que devuelven resultados (SELECT).

Retorno: Devuelve un objeto ResultSet.

Ejemplo:

```
ResultSet rs = stmt.executeQuery("SELECT * FROM empleados");
```

Método executeUpdate

Qué es: Un método de Statement o PreparedStatement para ejecutar consultas SQL que no devuelven resultados (INSERT, UPDATE, DELETE, CREATE, etc.).

Retorno: Devuelve el número de filas afectadas.

Ejemplo:

```
int filasAfectadas = stmt.executeUpdate("UPDATE empleados SET salario = 5000  
WHERE id = 1");
```

Método DatabaseMetaData

Qué es: Una interfaz que proporciona información sobre la base de datos, como tablas, usuarios, claves primarias, y relaciones.

Uso: Se obtiene a través del objeto Connection.

Ejemplo:

```
DatabaseMetaData metaData = conexion.getMetaData();  
System.out.println("Nombre del DBMS: " + metaData.getDatabaseProductName());
```

Información útil:

Listar tablas.

Soporte de funciones y tipos de datos.

Relación entre tablas (claves foráneas).

Método ResultSetMetaData

Qué es: Una interfaz que proporciona información sobre las columnas de un ResultSet.

Uso: Se utiliza para conocer el nombre, tipo y propiedades de las columnas devueltas por una consulta.

Ejemplo:

```
ResultSetMetaData rsMeta = rs.getMetaData();  
int columnas = rsMeta.getColumnCount();  
for (int i = 1; i <= columnas; i++) {  
    System.out.println("Columna " + i + ": " + rsMeta.getColumnName(i));  
}
```

Método *getImportedKeys*

Qué es: Un método de DatabaseMetaData que obtiene las claves foráneas que otras tablas apuntan a una tabla específica.

Retorno: Un ResultSet con la información de las claves importadas.

Ejemplo:

```
ResultSet rs = metaData.getImportedKeys(null, null, "mi_tabla");

while (rs.next()) {

    System.out.println("Columna: " + rs.getString("FKCOLUMN_NAME"));

    System.out.println("Referencia: " + rs.getString("PKTABLE_NAME"));

}
```

Método *getExportedKeys*

Qué es: Un método de DatabaseMetaData que obtiene las claves foráneas que apuntan a una tabla específica desde otras tablas.

Retorno: Un ResultSet con la información de las claves exportadas.

Ejemplo:

```
ResultSet rs = metaData.getExportedKeys(null, null, "mi_tabla");

while (rs.next()) {

    System.out.println("Columna: " + rs.getString("PKCOLUMN_NAME"));

    System.out.println("Tabla relacionada: " + rs.getString("FKTABLE_NAME"));

}
```

Ejemplo de tablas y el uso de los métodos *getImportedKeys* y *getExportedKeys*

Tabla clientes

```
CREATE TABLE clientes (

    cliente_id INT PRIMARY KEY,

    nombre VARCHAR(100)

);
```

Tabla pedidos

```
CREATE TABLE pedidos (  
    pedido_id INT PRIMARY KEY,  
    cliente_ref_id INT,  
    fecha DATE,  
    FOREIGN KEY (cliente_ref_id) REFERENCES clientes(cliente_id)  
);
```

Aquí:

En clientes, la clave primaria se llama cliente_id.

En pedidos, la clave foránea que referencia a clientes se llama cliente_ref_id.

Usando getImportedKeys

Al consultar las claves importadas de la tabla pedidos, ahora se ve más claro que esta tabla importa cliente_id desde la tabla clientes como cliente_ref_id.

```
ResultSet rs = metaData.getImportedKeys(null, null, "pedidos");  
while (rs.next()) {  
    System.out.println("Tabla primaria: " + rs.getString("PKTABLE_NAME"));  
    System.out.println("Columna primaria: " + rs.getString("PKCOLUMN_NAME"));  
    System.out.println("Columna foránea: " + rs.getString("FKCOLUMN_NAME"));  
}
```

Salida:

Tabla primaria: clientes

Columna primaria: cliente_id

Columna foránea: cliente_ref_id

Usando getExportedKeys

Si ahora consultas las claves exportadas de la tabla clientes, queda claro que otras tablas (como pedidos) están referenciando su clave primaria cliente_id como cliente_ref_id.

```
ResultSet rs = metaData.getExportedKeys(null, null, "clientes");

while (rs.next()) {

    System.out.println("Tabla foránea: " + rs.getString("FKTABLE_NAME"));

    System.out.println("Columna primaria: " + rs.getString("PKCOLUMN_NAME"));

    System.out.println("Columna foránea: " + rs.getString("FKCOLUMN_NAME"));

}
```

Salida:

```
Tabla foránea: pedidos

Columna primaria: cliente_id

Columna foránea: cliente_ref_id
```

Métodos en Statement y PreparedStatement

executeQuery(String sql)

Uso: Ejecuta consultas SQL que devuelven un conjunto de resultados (como SELECT).

Devuelve: Un ResultSet con los datos obtenidos.

Ejemplo:

```
ResultSet rs = statement.executeQuery("SELECT * FROM clientes");
```

executeUpdate(String sql)

Uso: Ejecuta instrucciones SQL que afectan filas en la base de datos, como INSERT, UPDATE, DELETE, y algunas operaciones DDL (CREATE TABLE, DROP TABLE).

Devuelve: Un entero que indica el número de filas afectadas.

Ejemplo:

```
int rowsAffected = statement.executeUpdate("DELETE FROM clientes

WHERE id = 1");
```

execute(String sql)

Uso: Ejecuta cualquier instrucción SQL, ya sea una consulta (SELECT) o una actualización (INSERT, UPDATE, etc.).

Devuelve: Un booleano.

true si el resultado es un conjunto de resultados (ResultSet).

false si el resultado es una actualización o no devuelve nada.

Ejemplo:

```
boolean hasResultSet = statement.execute("CREATE TABLE nueva_tabla (id INT)");  
if (hasResultSet) {  
    ResultSet rs = statement.getResultSet();  
} else {  
    int updateCount = statement.getUpdateCount();  
}
```

addBatch(String sql) (en Statement)

Uso: Añade múltiples instrucciones SQL para ejecutarlas como un lote.

Ejemplo:

```
statement.addBatch("INSERT INTO clientes (id, nombre) VALUES (1, 'Juan')");  
statement.addBatch("INSERT INTO clientes (id, nombre) VALUES (2, 'María')");  
statement.executeBatch(); // Ejecuta las instrucciones en el lote
```

executeBatch() (en Statement)

Uso: Ejecuta todas las instrucciones añadidas con addBatch.

Devuelve: Un array de enteros, donde cada valor indica el número de filas afectadas por cada instrucción.

Ejemplo:

```
int[] results = statement.executeBatch();
```

Métodos adicionales en PreparedStatement

PreparedStatement hereda todos los métodos de Statement, pero también tiene métodos específicos para asignar valores a los parámetros de la consulta preparada (?).

setInt(int parameterIndex, int value)

Uso: Asigna un valor entero a un parámetro.

Ejemplo:

```
preparedStatement.setInt(1, 123);
```

setString(int parameterIndex, String value)

Uso: Asigna un valor de tipo String a un parámetro.

Ejemplo:

```
preparedStatement.setString(2, "Juan");
```

setDouble(int parameterIndex, double value)

Uso: Asigna un valor de tipo double a un parámetro.

Ejemplo:

```
preparedStatement.setDouble(3, 19.99);
```

clearParameters()

Uso: Limpia los valores de todos los parámetros asignados en el PreparedStatement.

Ejemplo:

```
preparedStatement.clearParameters();
```

ClearParameteres()

El método clearParameters() en un PreparedStatement se usa cuando necesitas reutilizar el mismo objeto PreparedStatement para ejecutar múltiples consultas con diferentes parámetros, y deseas asegurarte de que no queden valores residuales en los parámetros definidos previamente.

Esto es útil en escenarios donde el mismo PreparedStatement ejecuta varias consultas similares con parámetros diferentes y necesitas limpiar todos los valores asignados antes de reconfigurarlos.

Escenario de uso

Supongamos que tienes una consulta preparada para insertar datos en una tabla, y quieres usar el mismo PreparedStatement para insertar múltiples filas. Si no llamas a `clearParameters()`, los valores anteriores se mantienen, lo que podría llevar a errores si cambias solo algunos parámetros y no otros.

Ejemplo sin `clearParameters()`

```
String sql = "INSERT INTO clientes (id, nombre, saldo) VALUES (?, ?, ?)";

try (PreparedStatement ps = connection.prepareStatement(sql)) {
    ps.setInt(1, 1);      // Primer cliente
    ps.setString(2, "Juan");
    ps.setDouble(3, 100.50);
    ps.executeUpdate();

    ps.setInt(1, 2);      // Segundo cliente
    ps.setString(2, "María");

    // ¡Error! Olvidaste cambiar el parámetro del saldo
    ps.executeUpdate(); // Esto usará el saldo de 100.50 del cliente anterior
}
```

Problema: En este caso, saldo seguirá siendo 100.50 porque no se cambió explícitamente para el segundo cliente.

Ejemplo usando `clearParameters()`

Con `clearParameters()`, te aseguras de empezar "desde cero" al configurar los parámetros para la siguiente ejecución:

```
String sql = "INSERT INTO clientes (id, nombre, saldo) VALUES (?, ?, ?)";

try (PreparedStatement ps = connection.prepareStatement(sql)) {
    ps.setInt(1, 1);      // Primer cliente
    ps.setString(2, "Juan");
    ps.setDouble(3, 100.50);
    ps.executeUpdate();

    ps.clearParameters(); // Limpia los parámetros previos

    ps.setInt(1, 2);      // Segundo cliente
    ps.setString(2, "María");
    ps.setDouble(3, 200.50);
    ps.executeUpdate();
}
```

```
ps.setInt(1, 2);      // Segundo cliente

ps.setString(2, "María");

ps.setDouble(3, 200.75);  // Configura un nuevo saldo

ps.executeUpdate();      // Inserta correctamente

}
```

Ventaja: Con `clearParameters()`, te aseguras de que todos los parámetros queden limpios y evitas usar valores residuales de consultas anteriores.

¿Cuándo es especialmente útil?

Reutilización intensiva: Si un `PreparedStatement` se reutiliza en bucles o para operaciones repetitivas, como en un sistema de lotes (batch).

```
for (Cliente cliente : listaClientes) {

    ps.clearParameters();

    ps.setInt(1, cliente.getId());

    ps.setString(2, cliente.getNombre());

    ps.setDouble(3, cliente.getSaldo());

    ps.executeUpdate();

}
```

Prevención de errores: Si los parámetros pueden variar significativamente entre ejecuciones y quieres asegurarte de que no queden valores antiguos.