Módulo 3. Integrando SQL. Arquitectura JDBC

CÓRDOBA CREA FUTURO

Introducción

Introducción

SQL (Structured Query Language) es un lenguaje estandarizado para gestionar y manipular datos en bases de datos relacionales, permitiendo realizar tareas como consultar, insertar, actualizar y eliminar información, siendo fundamental para el funcionamiento de sistemas que manejan grandes volúmenes de datos estructurados en tablas, como los de bancos, salud o *retail* [minoristas]. También permite definir la estructura de la base de datos (DDL) y controlar el acceso y transacciones (DCL/TCL), siendo importante en áreas como finanzas, *retail* y desarrollo web para el análisis de datos y soporte de aplicaciones *backend* gracias a su potencia y flexibilidad.

La historia de SQL

En la década de 1970, los científicos de IBM Donald Chamberlin y Raymond Boyce desarrollaron e introdujeron el

SQL. Se originó a partir del concepto de modelos relacionales y se denominó inicialmente lenguaje de consulta estructurado en inglés (SEQUEL) antes de acortarse a SQL. Se puso a la venta en 1979 y, desde entonces, se ha convertido en el estándar global de los sistemas de gestión de bases de datos relacionales.

El Instituto Nacional Estadounidense de Estándares (ANSI) y la Organización Internacional de Normalización (ISO) estandarizaron SQL en 1986 y 1987, respectivamente. A pesar de ser un estándar, SQL tiene varios dialectos, como T-SQL para Microsoft SQL Server y PL/SQL para Oracle Database. Estos dialectos de SQL satisfacen necesidades específicas del sistema sin dejar de cumplir los comandos estándar ANSI básicos, como SELECT, UPDATE, DELETE, INSERT y WHERE.

Funciones principales

- **Consultar datos (SELECT):** recuperar información específica según criterios.
- **Insertar datos (INSERT):** añadir nuevos registros a las tablas.

- **Actualizar datos (UPDATE):** modificar datos existentes.

- **Eliminar datos (DELETE):** borrar registros de la base de datos.

- **Definir y controlar bases de datos (DDL/DCL):** crear tablas, definir estructuras y controlar accesos.

¿Cómo funciona?

SQL se comunica con un Sistema Gestor de Bases de Datos (SGBD) como MySQL, PostgreSQL u Oracle, enviándole comandos para interactuar con los datos almacenados en tablas (filas y columnas). El resultado de una consulta suele ser una tabla temporal con la información solicitada, mientras que otras operaciones pueden generar un mensaje de confirmación.

¿Para qué se usa?

Es esencial en casi cualquier sector que maneje datos estructurados, desde aplicaciones bancarias y sistemas de inventario hasta análisis de *big data* [macrodatos], siendo una herramienta clave para la gestión de la información en aplicaciones web y empresariales.

CONTINUAR

1. Fundamentos de SQL

1. Fundamentos de SQL

¿Por qué SQL es fundamental?

SQL no es solo un lenguaje de consultas, es el **puente** entre la lógica de negocio (Java) y la persistencia de datos. Pensémoslo así:

1. Aplicación Java.
2. Capa de persistencia.
3. **SQL.**
4. Base de datos.

El siguiente diagrama representa la arquitectura de una aplicación Java donde la **aplicación Java** interactúa con una **capa de persistencia** (como JPA: Java Persistence API/Hibernate) para gestionar datos, que a su vez se comunican con una **base de datos** mediante **SQL** (a través

de conectores como JDBC), permitiendo persistir y consultar objetos Java de forma transparente. JPA traduce las operaciones de objetos (CRUD: **Create** [Crear], **Read** [Leer], **Update** [Actualizar] y **Delete** [Eliminar]) a SQL y viceversa, abstrayendo gran parte del código JDBC manual y facilitando el mapeo objeto-relacional (ORM: permite a los desarrolladores mapear clases Java a tablas de bases de datos relacionales (mapeo objeto-relacional u ORM). Esto simplifica la manipulación de datos sin escribir grandes cantidades de SQL, y trabajando directamente con objetos Java que se sincronizan con la base de datos mediante anotaciones o XML para definir la relación. Es un intermediario que facilita la comunicación entre la aplicación Java y la base de datos, usando herramientas como EntityManager para operaciones CRUD (Crear, Leer, Actualizar, Borrar).

Figura 1. Visión conceptual

Fuente: elaboración propia.

En un estilo UML (con clases y relaciones) y bajo un diagrama en estilo UML, simplificado, tenemos:

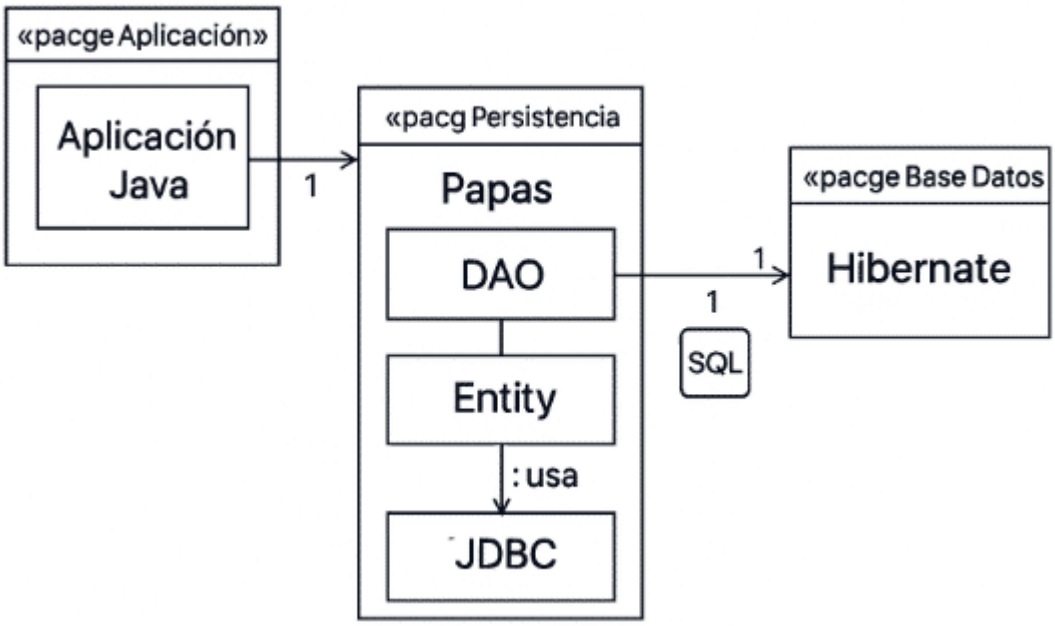
- aplicación Java → capa de persistencia → base de datos.
- Relaciones con multiplicidad (1 a 1).
- Indicador de uso de SQL en la interacción con la base de datos.

Figura 2. Visión conceptual, pero en formato UML

Fuente: elaboración propia.

Realizando lo mismo, pero **más completo** y agregando clases concretas (DAO, Entity, Service), tenemos:

Figura 3. Visión conceptual, pero agregando DAO, Entity Service, entre otros



Fuente: elaboración propia.

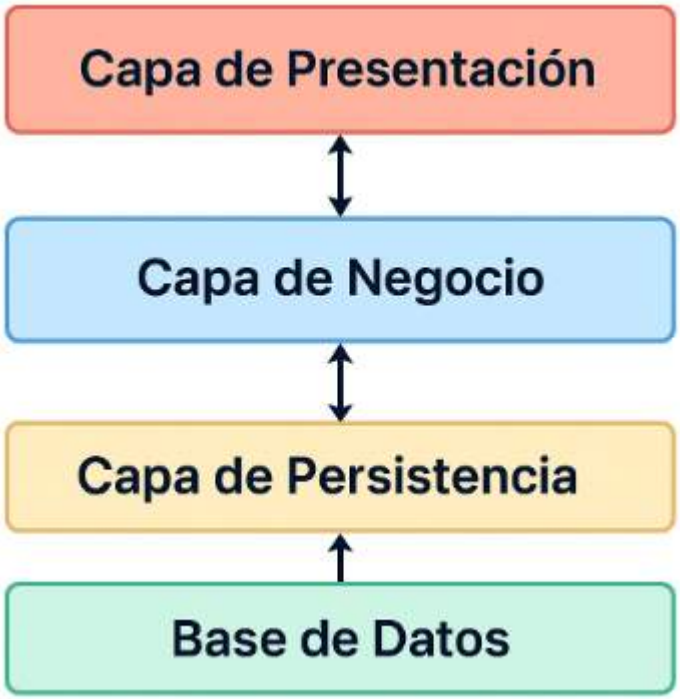
Aquí tenemos el **diagrama UML más completo** con lo siguiente.

- **Paquetes**: aplicación, persistencia, base de datos.
- **Clases concretas**:
    - DAO (Data Access Object);

    - Entity (representación de objetos persistentes),
    - JDBC (conexión directa).
- **Framework**: Hibernate.
- Indicador de uso de **SQL**.
- Relaciones con multiplicidad y estereotipos UM.

Si lo convertimos ahora en un **diagrama de arquitectura en capas** (presentación, negocio, persistencia, base de datos), tendríamos lo siguiente.

Figura 4. Diagrama de arquitectura en capas



Fuente: elaboración propia.

Diagrama de arquitectura en capas.

- **Capa de presentación** (interfaz gráfica, controladores).
  - **Capa de presentación ↔ capa de negocio**.
  - Componente UML dentro de la capa: Controller, View.
- **Capa de negocio** (servicios, lógica de negocio).
  - **Capa de negocio ↔ capa de persistencia**.
  - Componente UML dentro de la capa: Service, Business Logic.
- **Capa de persistencia** (DAO, ORM, JDBC).
  - **Capa de persistencia ↔ base de datos**.
  - Componente UML dentro de la capa: DAO, Repository, Hibernate/JDBC.

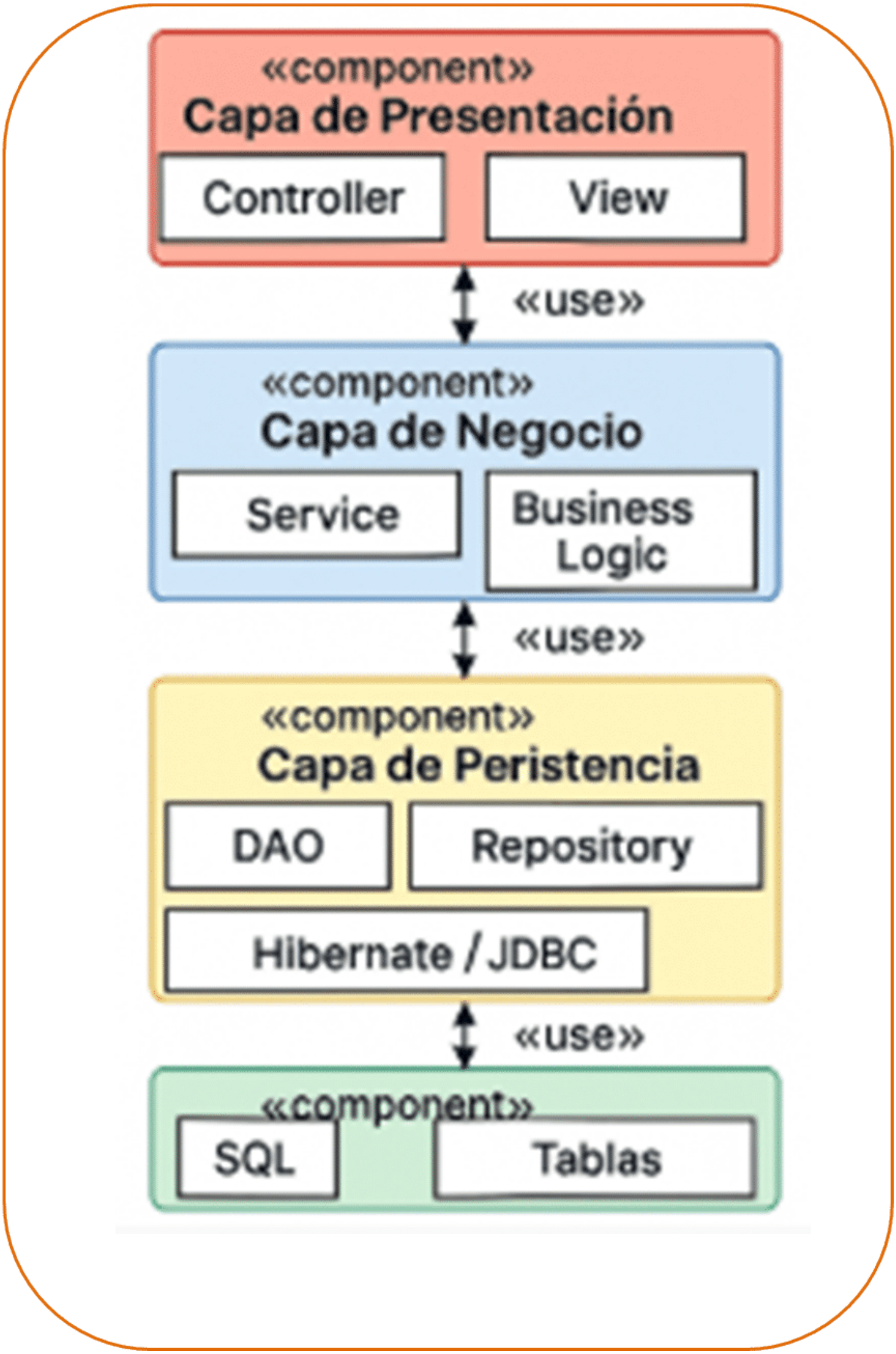
- **Base de datos** (SQL, tablas, almacenamiento)
    - Componente UML dentro de la capa: SQL, tablas.

A continuación, y dando un paso más, se muestra un **diagrama UML de componentes completo con** lo siguiente.

- **Estereotipos UML** (<<component>>, <<interface>>).
- **Dependencias** entre capas.
- **Interfaces expuestas** (por ejemplo, IController, IService, IDAO).
- **Componentes internos** en cada capa.

El diseño será así.

Figura 5. Diagrama UML


«component»
Capa de Presentación
Controller
View
«use»
«component»
Capa de Negocio
Service
Business Logic
«use»
«component»
Capa de Peristencia
DAO
Repository
Hibernate / JDBC
«use»
«component»
SQL
Tablas

Fuente: elaboración propia.

Capa de presentación

- <<component>> Controller.
- <<component>> View.
- Interfaces: IController.

Capa de negocio

- <<component>> Service.
- Interfaces: IService.

Capa de persistencia

- <<component>> DAO.
- <<component>> Repository.
- Interfaces: IDAO.

Base de datos

- <<component>> SQL Database.

Dependencias

- Controller → Service → DAO → SQL Database.

Es decir, el **diagrama UML de componentes completo** mostrado en la Figura 5 contiene lo siguiente.

- **Estereotipos UML** (<<component>>).
- **Dependencias** (<<use>>) entre capas.
- Componentes internos en cada capa.
  - **Presentación**: Controller, View.
  - **Negocio**: Service, Business Logic.
  - **Persistencia**: DAO, Repository, Hibernate/JDBC.
  - **Base de datos**: SQL, tablas.

¿Qué es y para qué sirve lo visto?

- **Especificación, no implementación:** JPA define un estándar, pero se necesita una implementación concreta (como Hibernate o EclipseLink) para usarla.

- **Mapeo objeto-relacional (ORM):** convierte objetos Java en filas de tablas de bases de datos y viceversa, resolviendo el «desajuste de impedancia» entre la programación orientada a objetos y las bases de datos relacionales.

- **Simplifica el desarrollo:** reduce drásticamente la necesidad de escribir código SQL para interactuar con la base de datos, permitiendo trabajar con objetos Java (POJOs) de forma nativa.

- **API de persistencia:** proporciona un conjunto de interfaces (paquete javax.persistence) y el lenguaje JPQL (Java Persistence Query Language) para consultar datos de forma orientada a objetos.

Componentes clave

- **Entidades (Entities):** clases Java que representan filas en una tabla de la base de datos.

- **EntityManager:** una API para gestionar el ciclo de vida de las entidades y realizar operaciones como persistir, buscar, actualizar y eliminar objetos de la base de datos.

- **Anotaciones/XML:** se usan para definir cómo se mapean las clases y sus campos a las tablas y columnas de la base de datos (ej. @Entity, @Table, @Column).

En resumen, JPA es una tecnología fundamental en Java para que los desarrolladores se centren en la lógica de negocio de sus aplicaciones, dejando la tediosa tarea de la gestión de la base de datos a un «motor» ORM que implementa la especificación JPA.

Tipos de datos y creación de tablas

En SQL, los tipos de datos definen qué información guarda una columna (números, texto, fechas, etc.), clasificándose en numéricos (INT, DECIMAL), texto (VARCHAR, CHAR, TEXT), fecha/hora (DATE, TIME, DATETIME) y otros especiales

(BOOLEAN, BINARY), mientras que la creación de tablas se hace con CREATE TABLE especificando el nombre de la tabla, las columnas, su tipo de dato y restricciones (como NOT NULL o PRIMARY KEY).

Tipos de datos comunes

- **Numéricos**: INT (enteros), DECIMAL(p,s) (exactos, para dinero), FLOAT (aproximados).
- **Texto**: VARCHAR(n) (longitud variable), CHAR(n) (longitud fija), TEXT (texto largo):
- **Fecha y hora:** DATE (fecha), TIME (hora), DATETIME (fecha y hora).
- **Lógicos**: BOOLEAN (TRUE/FALSE).
- **Binarios**: BINARY, VARBINARY (para imágenes, archivos).

Sintaxis para crear una tabla

Se usa la instrucción CREATE TABLE seguida del nombre de la tabla y, entre paréntesis, las definiciones de columna separadas por comas:

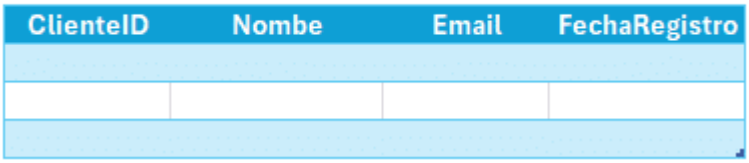
```
CREATE TABLE nombre_tabla (
  nombre_columna1 tipo_dato [restricciones],
  nombre_columna2 tipo_dato [restricciones],
  ...
);
```

Ejemplo

```
CREATE TABLE Clientes (
  ClienteID INT PRIMARY KEY,
  Nombre VARCHAR(100) NOT NULL,
  Email VARCHAR(255) UNIQUE,
  FechaRegistro DATE
);
```

Tabla 1. Tabla clientes

Tabla Clientes



ClienteID	Nombe	Email	FechaRegistro

Fuente: elaboración propia.

- **PRIMARY KEY**: identificador único para cada fila (registro).
- **NOT NULL**: asegura que la columna no puede estar vacía.
- **UNIQUE**: garantiza que todos los valores sean diferentes.

Ejemplo de tipos de datos

```
-- EJEMPLO PRÁCTICO: Tabla de usuarios

CREATE TABLE usuarios (

  -- IDENTIFICADORES

  id BIGINT PRIMARY KEY AUTO_INCREMENT,        -- 8 bytes, autoincremental
```

```
    uuid CHAR(36) UNIQUE,                    -- Para APIs externas

    -- DATOS PERSONALES
    nombre VARCHAR(100) NOT NULL,            -- VARCHAR vs CHAR: espacio dinámico
    email VARCHAR(255) UNIQUE NOT NULL,      -- Índice automático por UNIQUE

    -- DATOS NUMÉRICOS
    edad TINYINT UNSIGNED,                   -- 0-255, optimización de espacio
    salario DECIMAL(10, 2),                  -- Precisión exacta para dinero

    -- DATOS TEMPORALES
```

```
    fecha_registro TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    ultimo_login DATETIME,                -- Sin zona horaria

    -- DATOS BINARIOS
    avatar BLOB,                          -- Para imágenes pequeñas
    firma_digital VARBINARY(1024),        -- Datos binarios

    -- DATOS ESPECIALES
    preferencias JSON,                    -- PostgreSQL/MySQL 5.7+
    geolocalización POINT,                -- Coordenadas espaciales

    -- METADATOS
```

```
  activo BOOLEAN DEFAULT TRUE,

  version INT DEFAULT 0                    -- Para control de concurrencia

);
```

Nota: ¿por qué usar DECIMAL para dinero y no FLOAT?

- **Respuesta**: FLOAT tiene errores de redondeo en operaciones decimales.
- **Ejemplo**: 0.1 + 0.2 ≠ 0.3 en punto flotante.

Sentencias SQL

Se mencionan algunas sentencias.

- **SELECT**: extrae datos de una base de datos.
- **UPDATE**: actualiza datos de una base de datos.
- **DELETE**: elimina datos de una base de datos.
- **INSERT INTO**: inserta datos nuevos en una base de datos.

- **CREATE DATABASE**: crea una base de datos.
- **ALTER DATABASE**: modifica una base de datos.
- **CREATE TABLE**: crea una tabla.
- **ALTER TABLE**: modifica una tabla.
- **DROP TABLE**: elimina una tabla.
- **CREATE INDEX**: crea un índice (clave de búsqueda).
- **DROP INDEX**: elimina un índice.

Consultas SELECT avanzadas

Las consultas SELECT en SQL se usan para recuperar datos de una base de datos, especificando qué columnas y filas quieres ver de una o varias tablas, utilizando palabras clave como FROM para la tabla, WHERE para filtrar registros y ORDER BY para ordenar los resultados, siendo la sintaxis básica **SELECT columnas FROM tabla WHERE condiciones**. Es la instrucción fundamental para obtener información precisa de las tablas de la base de datos, permitiendo desde

obtener todos los datos (con SELECT *) hasta filtrarlos por criterios complejos.

Sintaxis básica

- **SELECT**: indica qué columnas quieres ver.
- ***:** selecciona todas las columnas de la tabla.
- **FROM**: especifica la tabla de donde se obtendrán los datos.
- **WHERE**: define las condiciones para filtrar las filas·
- **ORDER BY**: ordena los resultados (ascendente ASC o descendente DESC).

Ejemplos

```
-- 1. Seleccionar todas las columnas y filas de una tabla

SELECT * FROM Clientes;
```

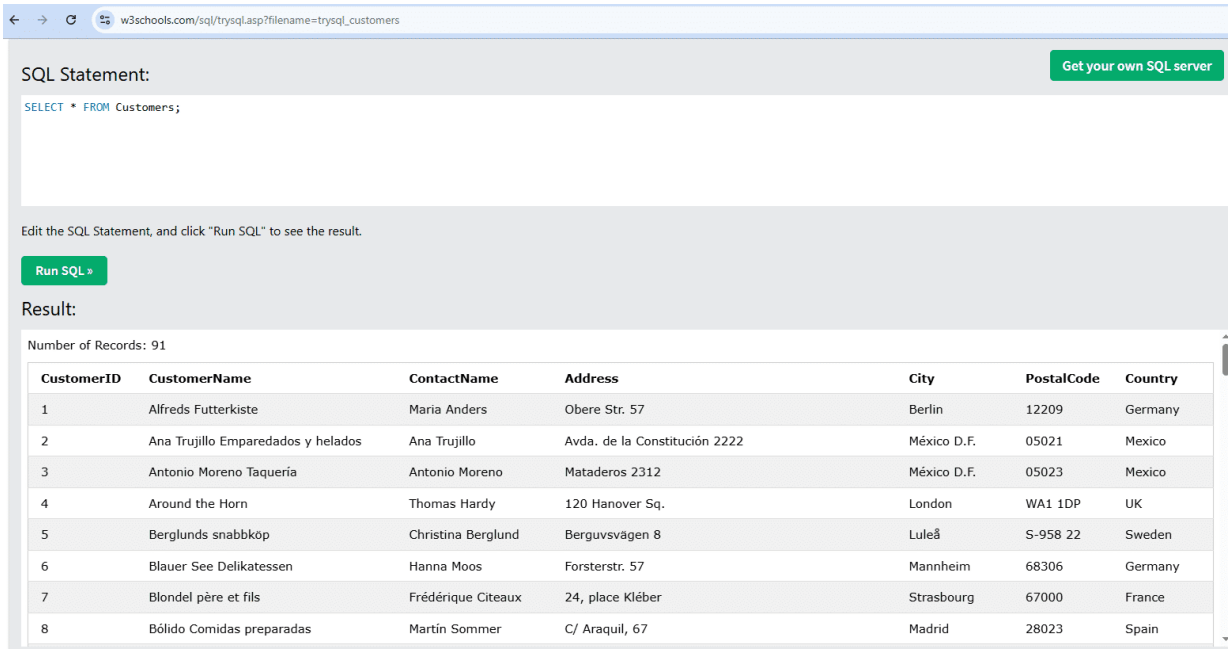
```
-- 2. Seleccionar columnas específicas

SELECT nombre, correo FROM Clientes;



-- 3. Seleccionar y filtrar por una condición

SELECT nombre, telefono FROM Clientes WHERE ciudad = 'Madrid';



-- 4. Seleccionar y ordenar resultados

SELECT * FROM Productos ORDER BY precio DESC;



-- 5. Seleccionar múltiples columnas con alias (renombrar columnas)

SELECT nombre AS 'Nombre del Cliente', email AS 'Correo Electrónico' FROM Clientes;
```

Ejemplos *online*

Acceder al siguiente *link*: https://www.w3schools.com/sql/trysql.asp?filename=trysql_customers

Allí podrás ver lo que se muestra a continuación.

Figura 6. SQL Statement



CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
5	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-958 22	Sweden
6	Blauer See Delikatessen	Hanna Moos	Forsterstr. 57	Mannheim	68306	Germany
7	Blondel père et fils	Frédérique Citeaux	24, place Kléber	Strasbourg	67000	France
8	Bólido Comidas preparadas	Martín Sommer	C/ Araquil, 67	Madrid	28023	Spain

Fuente: captura de pantalla de W3Schools (https://shorturl.at/CPi8L).

Lo que se observa en la Figura 6 es el contenido total de la tabla **Customers.**

Usando la caja de arriba de la Figura 6, es decir, la que dice **SELECT address FROM Customers**, reemplazar esas últimas sentencias por las que listan seguidamente y ejecutarlas usando el botón Run SQL.

De la tabla **Customers**.

a) Lista solo la columna **address**:

- **SELECT address FROM** Customers;.

b) Lista todas las columnas, cuyo campo Country sea igual a Mexico:

- **SELECT** * **FROM** Customers **WHERE** Country='Mexico';.

c) Lista las columnas CustomerName, Address:

- **SELECT** CustomerName, Address **FROM** Customers **WHERE** Country='Mexico';.

d) Lista los distintos países de la columna Country:

- **SELECT DISTINCT** Country **FROM** Customers;.

e) Lista todas las columnas, pero ordenadas por Country:

- **SELECT** * **FROM** Customers **ORDER BY** Country;.

f) Lista todas las columnas que son de Brazil y de la ciudad de Río de Janeiro cuyos ID son mayor que el valor 50:

- **SELECT** * **FROM** Customers **WHERE** Country **=** 'Brazil' **AND** City **=** 'Rio de Janeiro' **AND** CustomerID **>** 50;.

g) Lista las columnas cuya columna Country es Germany o Spain, es decir, con que se cumpla una de las dos:

- **SELECT** * **FROM** Customers **WHERE** Country **=** 'Germany' **OR** Country **=** 'Spain';.

Se recomienda leer y ejecutar todos los comandos del tutorial del siguiente enalce: https://www.w3schools.com/sql/default.asp

Podrás darte cuenta de que podrás realizar todo tipo de consultas, pero de que no podrás tener acceso a modificar y borrar algún dato, tabla o base de datos que no sean los ejemplos que se muestran en el tutorial.

Cláusulas GROUP BY, HAVING

Sentencia SQL GROUP BY

La sentencia GROUP BY agrupa las filas con los mismos valores en filas de resumen, como «Buscar el número de clientes en cada país».

La sentencia GROUP BY se utiliza a menudo con funciones de agregación (COUNT(), MAX(), MIN(), SUM(), AVG()) para agrupar el conjunto de resultados por una o más columnas,JOINS (INNER, LEFT, RIGHT, FULL).

Práctico: para entender esta sentencia, se recomienda ejecutar, en el siguiente enlace, la sentencia detallada abajo: https://www.w3schools.com/sql/trysql.asp?filename=trysql_select_groupby

SELECT COUNT(CustomerID), Country

FROM Customers

GROUP BY Country;.

Hacerlo usando la tabla Customers de la base de datos (aquí se muestra una parte del contenido).

Tabla 2. Tabla Customers

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
5	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-958 22	Sweden

Fuente: captura de pantalla de W3Schools (https://shorturl.at/8M9lN).

Cláusula HAVING de SQL

La cláusula HAVING se agregó a SQL porque la palabra clave WHERE no se puede usar con funciones de agregación.

Para entender el concepto en forma práctica, ejecutar las sentencias que siguen en este enlace: https://www.w3schools.com/sql/trysql.asp?filename=trysql_select_having

```
SELECT COUNT(CustomerID), Country

FROM Customers

GROUP BY Country
```

HAVING COUNT(CustomerID) > 5;.

También ejecuta, en el mismo enlace, lo siguiente:

SELECT COUNT(CustomerID), Country

FROM Customers

GROUP BY Country

HAVING COUNT(CustomerID) > 5

ORDER BYCOUNT(CustomerID) DESC;.

SQL para Aplicaciones

SQL es fundamental para aplicaciones, ya que permite a los desarrolladores comunicarse con bases de datos relacionales para crear, leer, actualizar y eliminar datos, siendo la columna vertebral de sistemas web, móviles y empresariales para gestionar información, generar informes y automatizar procesos, integrándose con lenguajes como Python, Java y PHP para construir aplicaciones robustas y escalables.

Usos principales de SQL en aplicaciones

- **Gestión de datos:,**CRUD (Crear, Leer, Actualizar, Eliminar) registros en tablas, desde la creación de la estructura (tablas, columnas, relaciones) hasta la manipulación de datos específicos.

- **Desarrollo web y móvil:** alimenta el *backend* de sitios y *apps*, recuperando información para mostrarla al usuario (ej. productos en una tienda *online*, perfiles de redes sociales).

- **Análisis de datos:** extrae, filtra y agrupa grandes volúmenes de datos para identificar tendencias, generar informes y apoyar la toma de decisiones (Business Intelligence).

- **Automatización:** usa *triggers* [disparadores] y procedimientos almacenados para ejecutar tareas automáticamente en la base de datos, lo que mejora la eficiencia.

- **Seguridad y autenticación:** gestiona usuarios y permisos dentro de la aplicación para controlar el acceso a la información.

- **Integración con otros lenguajes:** se incrusta en lenguajes como Python, Java, PHP para construir aplicaciones de alto rendimiento.

Componentes clave

- **JDBC API**: un conjunto de interfaces y clases en Java para la comunicación con bases de datos.
- *Drivers* **[controladores] JDBC**: archivos .jar específicos para cada base de datos (ej. MySQL Connector/J, SQL Server JDBC Driver) que implementan la API JDBC para ese motor.
- **Paquete** java.sql: contiene las clases principales como Connection, Statement, PreparedStatement, ResultSet y DriverManager.

El siguiente **diagrama de arquitectura JDBC** tiene los siguientes elementos.

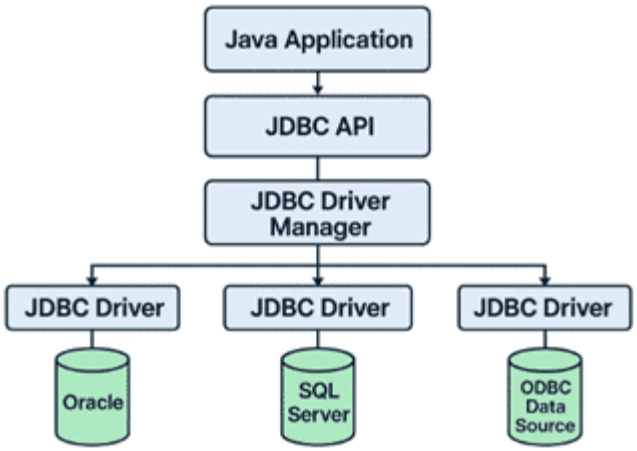
- Java Application en la parte superior.
- Conexión a través de JDBC API.
- JDBC Driver Manager como intermediario.

Tres ramas que representan diferentes JDBC Drivers:

- Uno conectado a Oracle (base de datos).
- Otro conectado a SQL Server.
- Otro conectado a una fuente de datos ODBC Data Source.

Este diagrama ilustra cómo una aplicación Java interactúa con diferentes bases de datos mediante JDBC y sus controladores específicos.

Figura 7. Flujo del diagrama



Fuente: elaboración propia.

Java Application → JDBC API → JDBC Driver Manager → *drivers* → bases de datos.

JDBC es una API de Java para acceder a manejadores de bases de datos. Esta API consiste en un conjunto de clases e interfaces que permiten a cualquier programa Java acceder a sistemas de bases de datos de forma homogénea.

En otras palabras, con el API JDBC no es necesario escribir un programa para acceder a Sybase, otro programa para acceder a Oracle y otro programa para acceder a MySQL. Con esta API, se puede crear un solo programa que sea capaz de enviar sentencias SQL a la base de datos apropiada.

Para poder realizar una conexión a una base de datos a través de un programa de Java, necesitamos el manejador o *driver* correspondiente al gestor de bases de datos a usar. En nuestro caso, al utilizar MySQL, utilizaremos el driver mysql-connector-java-5.1.6-bin, disponible en http://dev.mysql.com/downloads/connector/j/. Este *driver* hay que almacenarlo en el directorio de bibliotecas de la aplicación.

Todo programa Java que tiene que acceder a una base de datos MySQL debe tener las siguientes dos líneas de código.

En la primera, cargamos el conector JDBC para MySQL: **Class.forName(«com.mysql.jdbc.Driver»);**.

En la segunda, se crea la conexión a la base de datos. Entre paréntesis, especificamos la ruta y nombre de la base de datos,//localhost/database,, un nombre de usuario,*user*, y contraseña,*password*,que permita la conexión.

Connection con = DriverManager.getConnection(«jdbc:mysql://localhost/database»,»user»,»password»);.

Si todo sale bien, ya estaremos conectados a la base de datos. A partir de aquí, podremos ejecutar sentencias SQL. Pero, ¿cómo?

Lo primero que hay que hacer es crear un objeto de tipo Statement. Dicho objeto suministra métodos que permiten enviar sentencias SQL a la base de datos. Existen dos métodos importantes, cada uno con una función distinta.

- **executeUpdate(sentencia_sql);**: este método sirve para ejecutar sentencias de creación en bases de datos, de tablas de inserción de registros en las tablas. Es decir, permite enviar sentencias CREATE, DROP, INSERT y UPDATE. Simplemente se escribe entre comillas dobles la sentencia SQL a enviar.

- **executeQuery(sentencia_sql);**: este método sirve para ejecutar sentencias de consulta sobre tablas de una base de datos. Es decir, permite enviar sentencias SELECT. El resultado que devuelve se tiene que almacenar en un objeto de tipo **ResultSet**. Una vez obtenido dicho objeto, podemos acceder a sus datos con el método getObject(nombre_columna). Para ir avanzando por los diferentes registros almacenados en el objeto ResultSet, se emplea el método next().

Los objetos Statement y ResultSet que hayamos creado previamente deben cerrarse con el método close() cuando hayamos terminado de usarlos, de una manera parecida a cuando se cierran los flujos de lectura y de escritura en los accesos a ficheros.

Ejemplo

JDBC (Java Database Connectivity)

Es el enfoque más común y fundamental.

- **Conexión:** se usa DriverManager.getConnection() para establecer la conexión con la BD, especificando el driver y la URL.

- **Ejecución de consultas**

  - Statement: para sentencias SQL simples.
  - PreparedStatement: para sentencias con parámetros (más seguro y eficiente).

- **Procesamiento de resultados:** ResultSet se usa para recorrer las filas devueltas por una consulta SELECT**.**

- **Manejo de datos:** las clases de java.sql mapean tipos de datos SQL a tipos de datos Java (ej. INTEGER a int, DOUBLE a double).

Si agregas el siguiente código en tu aplicación Java, reemplazando el nombre de la base de datos, usuario, contraseña, IP o localhost, podrías conectarte sin problemas.

// Ejemplo básico de JDBC

```
import java.sql.*;



try (Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/db", "user", "pass");

    Statement stmt = conn.createStatement();

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



   while (rs.next()) {

                        System.out.println(rs.getInt("id") + ": " + rs.getString("nombre"));

   }

} catch (SQLException e) {

   e.printStackTrace();
```

```
}
```

CONTINUAR

2. Transacciones (ACID, COMMIT, ROLLBACK)

2. Transacciones (ACID, COMMIT, ROLLBACK)

En SQL, las Transacciones ACID son un conjunto de propiedades fundamentales (Atomicidad, Consistencia, Aislamiento, Durabilidad) que garantizan la integridad y fiabilidad de las operaciones en una base de datos, asegurando que una transacción se complete totalmente o no se realice en absoluto, manteniendo los datos válidos y seguros ante fallos y gestionando correctamente las operaciones concurrentes.

¿Qué es una transacción SQL?

Es una secuencia de una o más operaciones (sentencias SQL como INSERT, UPDATE, DELETE) que se tratan como una **única unidad lógica de trabajo** para modificar la base de datos, garantizando que todas se aplican o ninguna lo hace, preservando la consistencia.

Propiedades ACID

- **Atomicidad (Atomicity):** «Todo o nada». La transacción se ejecuta completamente o no se ejecuta en absoluto. Si un paso falla, todos los cambios anteriores se revierten (*rollback*).

    - Ejemplo: una transferencia bancaria. Si el débito funciona, pero el crédito falla, se cancela todo para que el dinero no desaparezca.

- **Consistencia (Consistency):** la transacción lleva la base de datos de un estado válido a otro estado válido. Mantiene las reglas y restricciones de integridad definidas (como *constraints*, *triggers*).

    - Ejemplo: si una cuenta no puede tener saldo negativo, una transacción que lo causaría sería rechazada para mantener la consistencia

- **Aislamiento (Isolation):** las transacciones concurrentes se ejecutan de forma independiente. El estado intermedio de una

transacción no es visible para otras, como si se ejecutaran una tras otra.

  - Ejemplo: en una reserva, otro usuario no debería ver una habitación como disponible si ya está siendo reservada por una transacción simultánea

- **Durabilidad (Durability):** una vez que una transacción se confirma (COMMIT), sus cambios son permanentes y persisten, incluso si hay fallos del sistema (cortes de energía, caídas).

  - Ejemplo: el registro de una compra debe guardarse de forma permanente en la base de datos para no perderse

Importancia en SQL

Estas propiedades son cruciales para sistemas donde la integridad de los datos es crítica, como bancos, comercio electrónico y sistemas de inventario, previniendo corrupción y garantizando operaciones fiables.

Ejemplo

```sql
-- Transferencia bancaria: DEBE SER ATÓMICA

START TRANSACTION;



-- 1. Restar del origen

UPDATE cuentas

SET saldo = saldo - 1000

WHERE id = 1 AND saldo >= 1000;



-- 2. Sumar al destino

UPDATE cuentas

SET saldo = saldo + 1000

WHERE id = 2;
```

```
-- Verificar ambas operaciones exitosas

IF (ROW_COUNT() = 2) THEN

  COMMIT;  -- Todo ok, confirmar

ELSE

  ROLLBACK;  -- Algo falló, revertir

END IF;
```

ACID explicado

- **A**tomicidad: todo o nada.
- **C**onsistencia: reglas se mantienen.
- A(**i**)slamiento: transacciones no se interfieren.
- **D**urabilidad: sobrevive a fallos del sistema.

Transacciones COMMIT y ROLLBACK

COMMIT y ROLLBACK son comandos SQL que controlan las transacciones, unidades lógicas de trabajo en una base de datos; COMMIT guarda permanentemente los cambios, mientras que ROLLBACK los deshace, revirtiendo la base de datos a su estado anterior, esencial para mantener la integridad de los datos ante errores o condiciones no deseadas.

Ejemplo

```
BEGIN TRANSACTION; -- Start the transaction

-- Perform database operations

UPDATE accounts SET balance = balance - 100 WHERE account_id = 1;

UPDATE accounts SET balance = balance + 100 WHERE account_id = 2;

COMMIT; -- Finalize the transaction
```

Si se produce un error, puedes deshacer la transacción:

```
BEGIN TRANSACTION;
```

```
UPDATE accounts SET balance = balance - 100 WHERE account_id = 1;

-- Simulate an error

ROLLBACK; -- Undo the changes
```

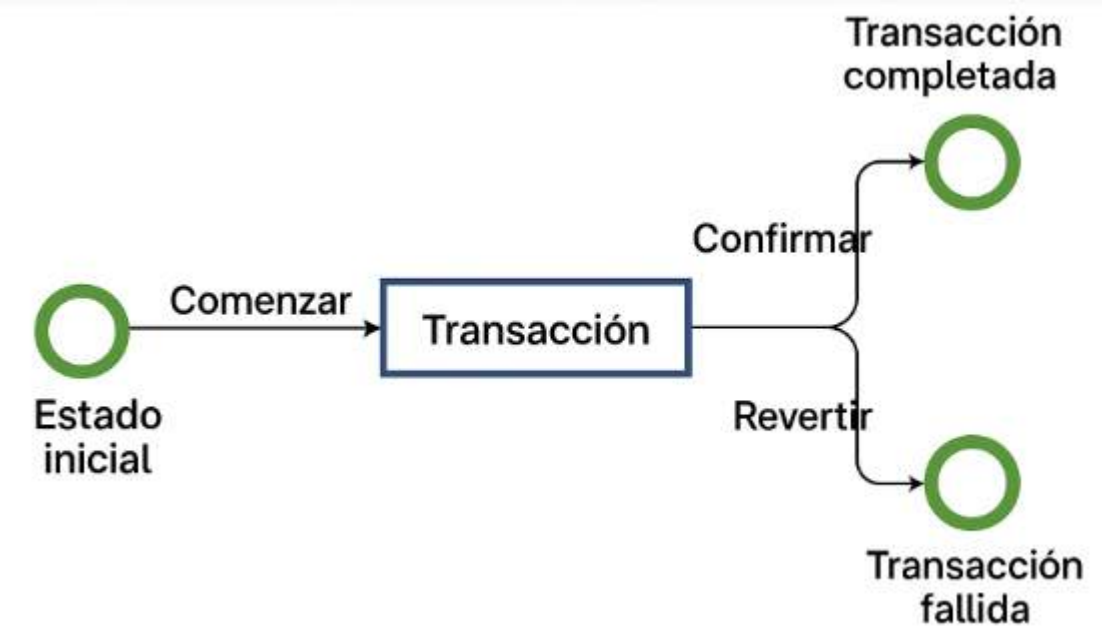
Dominar las transacciones SQL es una habilidad que merece la pena para cualquier desarrollador o administrador de bases de datos. Para empezar, primero deberías aprender los fundamentos de las propiedades ACID y luego practicar implementaciones básicas con BEGIN, COMMIT y ROLLBACK.

La siguiente figura muestra un diagrama de flujo de transacciones con los siguientes elementos.

- Estado inicial (círculo verde a la izquierda).
- Flecha hacia un rectángulo central etiquetado como transacción.
- Desde transacción salen dos ramas.
    - **Commit** → (círculo verde superior derecho) confirma los cambios (éxito).

- **Rollback** → (círculo verde inferior derecho) revierte los cambios (fallo).

Figura 8. Diagrama de flujo de transacciones



Fuente: elaboración propia.

Patrones de acceso a datos

Patrón DAO (Data Access Object)

Problema: mezclar lógica de negocio con SQL

Solución: separar responsabilidades.

Ejemplo

```
// INTERFACE - Contrato de operaciones
public interface UsuarioDAO {
  Usuario encontrarPorId(Long id);
  List<Usuario> listarTodos();
  void guardar(Usuario usuario);
  void actualizar(Usuario usuario);
  void eliminar(Long id);
}

// IMPLEMENTACIÓN CONCRETA
public class UsuarioDAOImpl implements UsuarioDAO
{
```

```
    private Connection conexion;



    @Override
    public Usuario encontrarPorId(Long id) {
        String sql = "SELECT * FROM usuarios WHERE id =
?";


                    try (PreparedStatement stmt =
conexion.prepareStatement(sql)) {
            stmt.setLong(1, id);
            ResultSet rs = stmt.executeQuery();


            if (rs.next()) {
                    return mapearUsuario(rs);  // Método de
mapeo
```

```
            }

        } catch (SQLException e) {

            throw new DataAccessException("Error al buscar
usuario", e);

        }

        return null;

    }



    private Usuario mapearUsuario(ResultSet rs) throws
SQLException {

        Usuario usuario = new Usuario();

        usuario.setId(rs.getLong("id"));

        usuario.setNombre(rs.getString("nombre"));

        usuario.setEmail(rs.getString("email"));
```

```
        // ... más campos

        return usuario;

    }

}
```

Ventajas

1. Separación de preocupaciones.
2. Fácil de examinar (*mocks*).
3. Cambio de fuente de datos sin afectar negocio.

Inyección de dependencias para DAO

Se da un ejemplo con y sin uso de DAO, donde el **beneficio** permite cambiar la implementación del DAO sin modificar el Service.

```
// SIN INYECCIÓN - ACOPLAMIENTO FUERTE

public class UsuarioService {
```

```
            private UsuarioDAO usuarioDAO = new
UsuarioDAOImpl();


  public Usuario obtenerUsuario(Long id) {

     return usuarioDAO.encontrarPorId(id);

  }

}


// CON INYECCIÓN - BAJO ACOPLAMIENTO

public class UsuarioService {

  private UsuarioDAO usuarioDAO;


  // Constructor injection

  public UsuarioService(UsuarioDAO usuarioDAO) {
```

```
        this.usuarioDAO = usuarioDAO;
    }

    public Usuario obtenerUsuario(Long id) {
        return usuarioDAO.encontrarPorId(id);
    }
}

// USO
UsuarioDAO dao = new UsuarioDAOImpl();
UsuarioService service = new UsuarioService(dao);
```

CONTINUAR

Lección 4 de 4

Descarga en PDF