Módulo 1. DAO y JDBC. Introducción JPA e Hibernate básico

La persistencia aborda en detalle este concepto fundamental en el desarrollo de *software*, especialmente en el contexto de las aplicaciones orientadas a objetos y su interacción con los sistemas de gestión de bases de datos (SGBD). En este sentido, introducimos los siguientes conceptos:

DAO (*Data Access Object*) es un patrón de diseño usado para abstraer el acceso a datos, mientras que **JDBC (*Java Database Connectivity*)** es una API de bajo nivel para ejecutar SQL directamente. **JPA (*Java Persistence API*)** es una especificación de alto nivel para el mapeo objeto-relacional (ORM), e **Hibernate** es una implementación popular de JPA que convierte objetos Java en filas de base de datos, simplifica el código al evitar el SQL manual y permite trabajar directamente con clases Java. DAO encapsula la lógica, JDBC interactúa directamente con SQL, y JPA/Hibernate (ORM) mapea objetos a tablas; Hibernate ofrece características avanzadas sobre la especificación JPA.

En forma resumida y para ambientar lo que se explica aquí, se describen cada uno de los conceptos que se ampliarán más adelante:

1. **DAO**

- **Propósito**: separar la lógica de acceso a datos (CRUD: crear, leer, actualizar, eliminar) del resto de la aplicación, creando una capa de abstracción.
- **Beneficios**: código más modular, fácil mantenimiento, reutilización y pruebas unitarias simplificadas.

2. **JDBC**

- **Nivel**: bajo nivel.
- **Funcionalidad**: permite ejecutar comandos SQL directamente y mapear resultados a clases Java manualmente.
- **Cuándo usarlo**: para control total, rendimiento máximo, o si se prefieren consultas SQL explícitas y se tiene un buen conocimiento de la base de datos.

3. **JPA**

- **Nivel**: alto nivel, especificación estándar.
- **Propósito**: definir una forma estándar de mapear clases Java a tablas de bases de datos (ORM).
- **Beneficios**: abstrae el SQL, permitiendo interactuar con la base de datos usando objetos Java.

4. **Hibernate**

- **Relación con JPA**: es una implementación concreta y popular de la especificación JPA.
- **Funcionalidad**: implementa las funcionalidades de JPA y añade características extras (caché, carga perezosa).
- **Cómo funciona**: mapea clases Java a tablas y tipos de datos SQL, manejando las operaciones de persistencia (CRUD).

1. Persistencia

La persistencia en Java es la capacidad de **almacenar datos de forma duradera** para que no se pierdan al cerrar una aplicación, sobreviviendo a la ejecución del programa, a diferencia de las variables en memoria RAM que son volátiles; esto se logra guardando datos en sistemas no volátiles **como archivos o bases de datos.** Es fundamental en aplicaciones empresariales, donde **JPA** es el estándar moderno para mapear objetos Java a bases de datos relacionales (ORM).

¿Cómo funciona?

- **Objetos vs. datos**: los objetos Java viven en la memoria (RAM) y desaparecen al terminar el programa. La persistencia los "salva" guardándolos en un lugar permanente.
- **Almacenamiento secundario**: los datos se guardan en el disco duro, memoria *flash* o bases de datos.
- **Serialización**: Java permite convertir objetos en un flujo de bytes (serializar) para guardarlos en archivos y reconstruirlos después (deserializar).
- **ORM (mapeo objeto-relacional**): tecnologías como JPA usan anotaciones para conectar clases Java con tablas de bases de datos, simplificando así la gestión de datos.

Tecnologías clave:

- **JDBC**: para interactuar directamente con bases de datos.
- **JPA**: el estándar para persistencia en Java; unifica API como Hibernate, EclipseLink, etc.
- **Serialización de objetos**: para persistencia "ligera" en archivos.

E/S de archivos (*file* I/O): guardar y leer datos directamente de archivos.

¿Para qué sirve?

- **Aplicaciones empresariales**: sistemas ERP, aplicaciones web que necesitan guardar información de usuarios, productos, transacciones, etc.
- **Guardar configuraciones**: mantener ajustes de una aplicación entre sesiones.
- **Almacenar el estado de la aplicación**: para poder reanudarla exactamente donde se dejó.

Objetivos de una capa de persistencia

Basado en los requerimientos de Scott W. Ambler, una capa de persistencia robusta debe cumplir con diversas características:

Diversificación de mecanismos de persistencia: capacidad para trabajar con múltiples sistemas de almacenamiento (bases de datos, sistemas de ficheros, etc.).

Encapsulación completa del mecanismo de persistencia: la capa se encarga de las operaciones de almacenamiento, eliminación y recepción de objetos, lo

que libera a la aplicación de estos detalles.

- **Acciones sobre múltiples objetos**: permite consultar o eliminar varios objetos a la vez.
- **Transacciones**: soporte para acciones combinadas y parametrización de operaciones SQL.
- **Identificadores de objetos**: atributos numéricos únicos para identificar objetos.
- **Cursores y *proxies***: mecanismos para gestionar resultados y representaciones de objetos.
- **Registros**: para la generación de informes.
- **Soporte para múltiples arquitecturas**: adaptabilidad a arquitecturas cliente/servidor de dos o múltiples capas.
- **Compatibilidad con diferentes bases de datos de distintos fabricantes**: facilidad de adaptación sin modificar la aplicación.
- **Múltiples conexiones**: capacidad para trabajar con SGBD descentralizados.
- **Controladores nativos**: uso de sistemas de acceso a bases de datos comunes o proporcionados por fabricantes.

- **Consultas SQL**: permite realizar consultas SQL, aunque preferiblemente como excepción.

Un objetivo clave es la transparencia, que permite que la aplicación y la base de datos se comuniquen como si fuera un sistema único, incluso si se cambia la base de datos. La transparencia asegura la independencia de ambas partes mientras permite que trabajen de forma conjunta.

¿Qué es una capa de persistencia?

La persistencia se define como la capacidad de guardar permanentemente la información generada por una aplicación en un sistema de almacenamiento para su uso futuro.

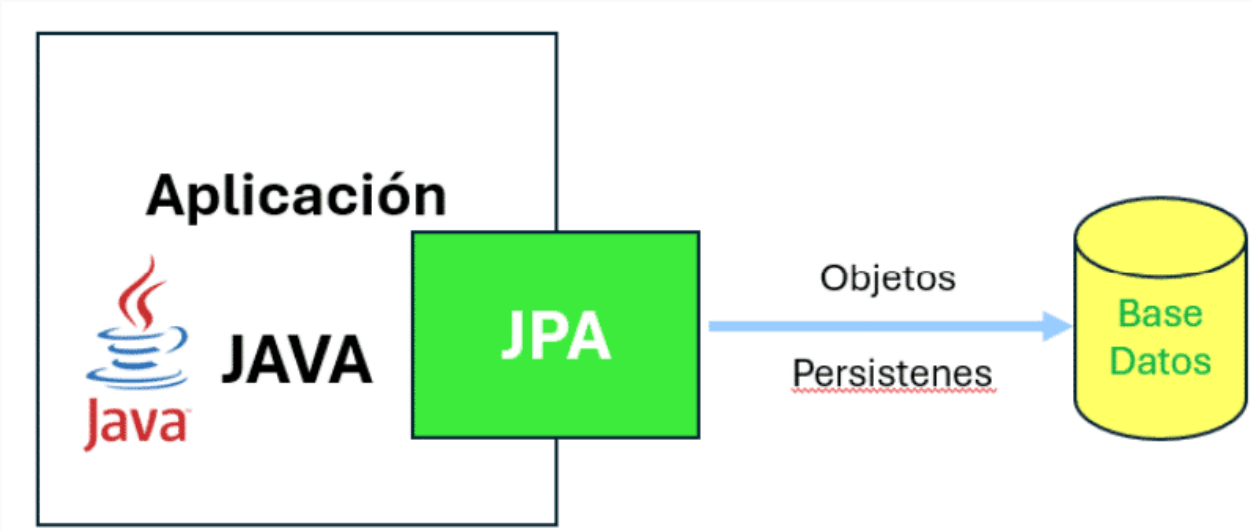
Para el programador, implica que los datos de un objeto deban sobrevivir a la ejecución del proceso que los creó y ser reutilizables en otros procesos, de forma transparente y sin necesidad de traducción explícita. El objetivo es que el programador no se preocupe por el mecanismo interno de persistencia.

Un *framework* de persistencia se encarga de gestionar de forma completa y transparente la lógica de acceso a los datos en un SGBD, ocultando los detalles de la estructura de la base de datos.

Se explica la composición de un objeto (atributos, métodos, relaciones) y cómo las bases de datos relacionales trabajan con registros y tipos primitivos.

El capítulo destaca el problema de ***impedance mismatch***, o falta de correspondencia, entre el modelo de objetos de un lenguaje de programación y el modelo de datos de las bases de datos relacionales. Las capas de persistencia surgen como una solución a este problema, estableciendo un modelo de correspondencia entre ambos.

Figura 1: Esquema general de persistencia



Fuente: elaboración propia.

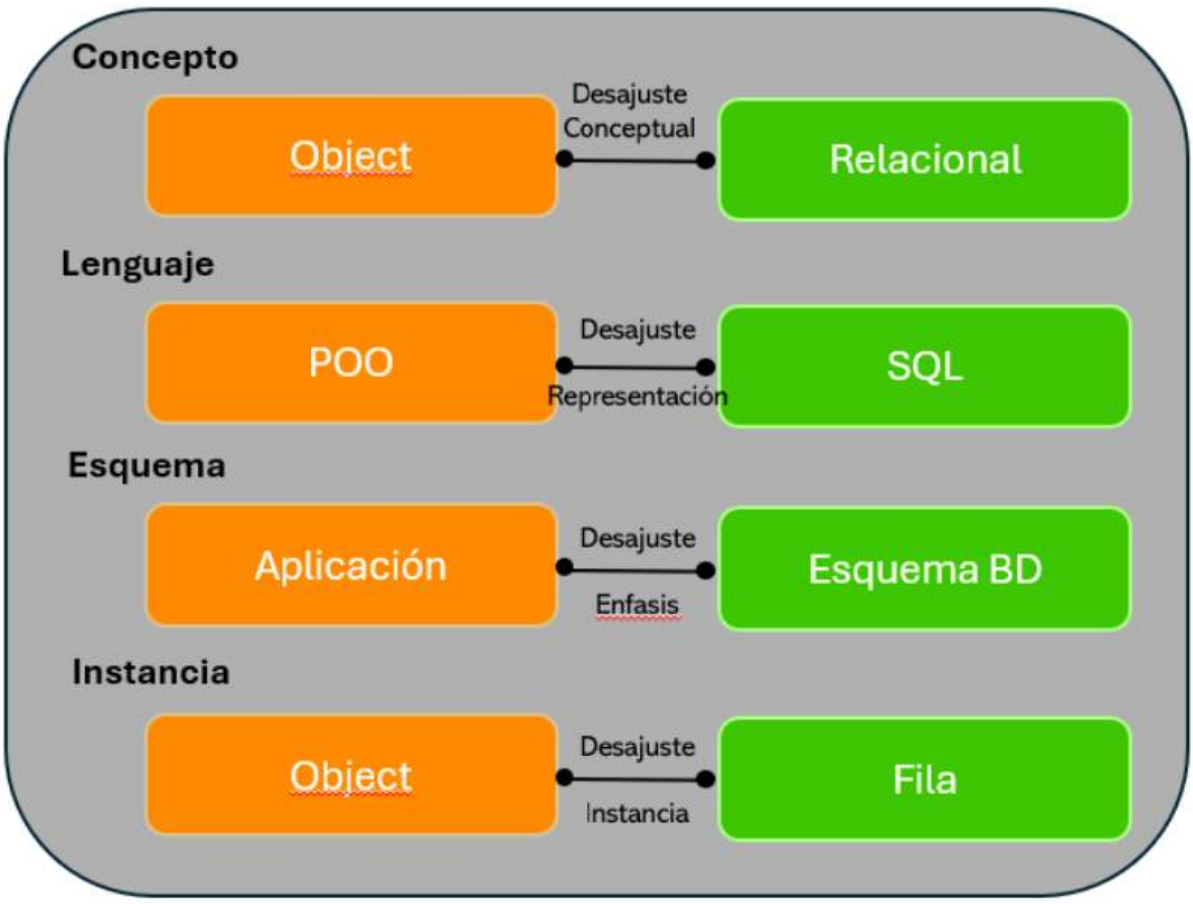
¿Qué es el problema de *impedance mismatch*?

En informática, el *impedance mismatch*, o desajuste de impedancia, se refiere a la incompatibilidad entre diferentes sistemas, especialmente entre lenguajes de programación orientados a objetos y bases de datos relacionales, o entre diferentes protocolos de comunicación. Este desajuste puede causar

problemas en la transferencia o integración de datos, ineficiencias en el rendimiento y complejidad en el desarrollo.

En una explicación más detallada, el término "desajuste de impedancia" proviene de la ingeniería eléctrica, donde se lo usa para referirse a la diferencia en la impedancia de diferentes componentes de un circuito, lo que puede causar reflexiones de señal y pérdida de potencia. En el contexto de la informática, se aplica de forma análoga a la incompatibilidad entre diferentes sistemas o modelos de datos.

Figura 2: Modelo de sistema de persistencia



Fuente: elaboración propia.

Ejemplos comunes de desajuste de impedancia

Desajuste objeto-relacional:

este es el caso más común. Ocurre cuando se intenta mapear objetos de un lenguaje de programación orientado a objetos a tablas de una base de datos relacional. La forma en que se representan y manipulan los datos en estos dos sistemas es diferente, lo que requiere la aplicación de conversiones y transformaciones, que pueden ser complejas y generar problemas. Por ejemplo, una jerarquía de objetos puede necesitar ser aplanada en una tabla, o varias tablas relacionadas pueden necesitar ser unidas para representar un objeto.

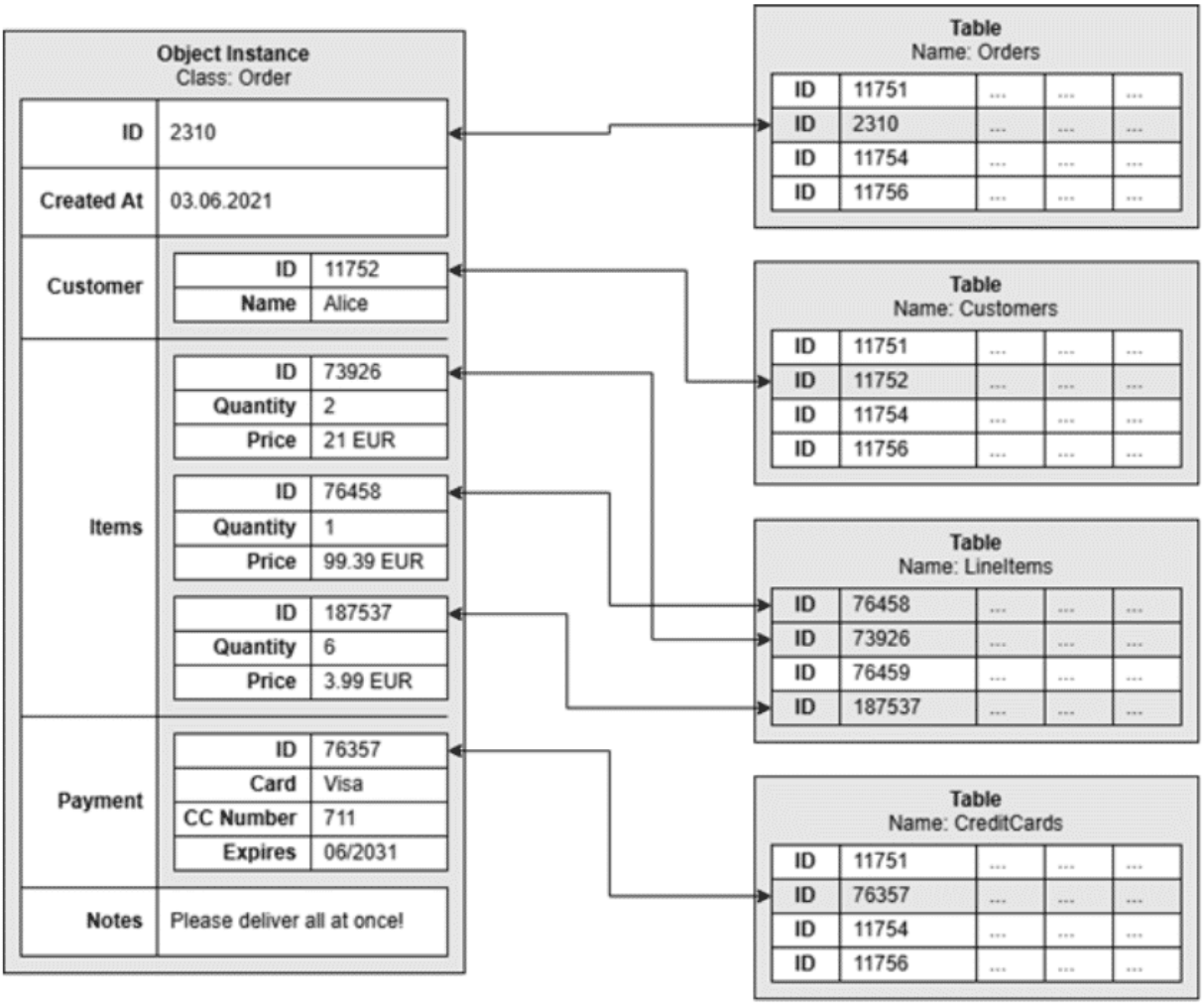
Desajuste entre protocolos de comunicación:

otro ejemplo es cuando se intenta comunicar sistemas que utilizan diferentes protocolos de comunicación, como SOAP y REST. La incompatibilidad en los formatos de datos y las estructuras de los mensajes puede causar problemas en la comunicación.

Desajuste entre sistemas:

en general, cualquier situación donde dos sistemas tienen diferentes estructuras de datos, modelos o protocolos y necesitan interactuar, puede generar un desajuste de impedancia.

Figura 3: Instancia objeto y tabla



Fuente: [imagen sin título sobre instancia objeto y tabla] (s. f.) https://tinyurl.com/22jhkfyf

Los modelos de sistema de persistencia

A lo largo del tiempo, han surgido diversos sistemas de persistencia, desde los más primitivos hasta las capas de persistencia más evolucionadas.

◦ Los sistemas de almacenamiento de datos más comunes incluyen bases de datos relacionales (SGBDR), orientadas a objetos (SGBDOO) y objeto-relacionales, así como sistemas de archivos (ASCII, XML). También se

mencionan las arquitecturas de objetos distribuidos como EJB y las bases de datos XML como de creciente importancia.

◦ Las capas de persistencia son los sistemas más evolucionados, capaces de trabajar sobre múltiples sistemas de almacenamiento.

◦ La tabla comparativa a continuación enumera y compara diferentes sistemas y capas de persistencia (serialización, JDBC-ODBC, ODBMS, EJB, JDO, ODMG, ADO) según las características de Ambler. Podemos concluir que JDO y ODMG (ejemplos de capas de persistencia) son los que cumplen la mayoría de los requerimientos.

Tabla 1: Diferentes sistemas y capas de persistencia

Requerimiento	Serialización	JDBC-ODBC	ODBMS	EJB	JDO	ODMG	ADO
Diferente tipos de mecanismos	Sistemas de archivos	SGBDR	SGBDOO	SGBDR, EAI, sistemas de archivos, etc.	Sistema de archivos SGBDR, SGBDOO, EAI, BBDD cobol, otros	Sistema de archivos SGBDR, SGBDOO, EAI, BBDD cobol, otros	SGBDR
Encapsulación completa del mecanismos de persistencia	Sí	Sí	Sí	Sí	Sí	Sí	Sí
Acciones multiobjeto	No	Sí	Sí	Sí	Sí	Sí	Sí
Transacciones	No	Sí	Sí	Sí	Sí	Sí	Sí
Extensibilidad	No	No	No	No	Sí	Sí	Solo para tipo de objeto
Identificadores de los objetos	Sí, manual	Sí, manual	Sí	Sí	Sí	Sí	Sí
Cursores	No	Sí	No	Sí	Sí	Sí	Sí
Proxies	No		Sí	Sí	Sí	Sí	Sí
Registros	Sí, manual	Sí, manual	Sí	Sí	Sí	Sí	Sí
Soporte para múltiples arquitecturas	No	No	No	No	Sí	No	No
Posibilidad de trabajar con BBDD de diferentes fabricantes	No	Sí	No	Sí	Sí	Sí	Sí
Conexiones múltiples	No	No	Sí	Sí	Sí	No	No
Uso de controladores nativos o externos	No	No nativos	No	No nativos	No nativos	No nativos	No nativos
Consultas SQL	No	Sí	Sí	Sí	Sí	No	Sí

Fuente: Vasen, 2013.

- Instancia persistente y transitoria: una instancia persistente mantiene sus datos más allá de la ejecución del proceso que la creó, mientras que una transitoria desaparece al finalizar el proceso.
- Servicio de persistencia de objetos: es un mecanismo que proporciona una interfaz única para el almacenamiento, recuperación, actualización y eliminación del estado de los objetos, ocultando las diferencias entre el modelo de objetos del lenguaje y el modelo de datos del SGBD. JDO y ODMG 3.0 son considerados servicios de persistencia de objetos en Java.
- Aproximaciones para hacer un objeto persistente:
  - **Por tipo**. El objeto es persistente si su clase tiene la capacidad de persistir.
  - **Por invocación explícita**. El programador llama a un método para hacer el objeto persistente.
  - **Por referencia**. Un objeto se hace persistente al ser referenciado por otro objeto persistente.
- Persistencia ortogonal: implica que el uso de la persistencia no afecta la manipulación de los objetos por parte de los programas, y los mismos mecanismos operan tanto en objetos persistentes como transitorios. Sus beneficios incluyen menos código, facilidad de mantenimiento y mayor productividad.
- Cierre de persistencia: un mecanismo que permite que las dependencias de un objeto también se guarden o recuperen automáticamente cuando el objeto es salvado o recuperado, lo que asegura su consistencia.
- Persistencia por alcance (o en profundidad): proceso de convertir automáticamente en persistentes todos los objetos directa o indirectamente referenciados por un objeto persistente.
- Transparencia de datos: ocurre cuando las clases persistentes y el esquema de la base de datos son uno, y los estados se manejan con el lenguaje de programación, eliminando así la *impedance mismatch*. Aunque relacionada, no debe confundirse con la persistencia ortogonal completa.
- Falta de correspondencia entre clases y datos *impedance mismatch*: se reitera este desafío que requiere el mapeo de objetos (p. ej., correspondencia objeto-registros para SGBDR) para definir cómo una clase se convierte en datos en el modelo de almacenamiento.

CONCEPTOS DE LA PERSISTENCIA DE OBJETOS

MECANISMOS DE PERSISTENCIA

Se categorizan los mecanismos de persistencia en varias áreas:

- **Acceso directo a base de datos**: implica el uso directo de la base de datos sin *middleware*. Incluye acceso a bases de datos relacionales, objeto-relacionales, orientadas a objetos y XML.
- **Mapeadores**: se basan en la traducción bidireccional entre objetos y una fuente de datos con un paradigma diferente, lidiando con el *impedance mismatch*. Los más comunes son los mapeadores objeto-relacional y objeto-XML.
- **Generadores de código**: herramientas que generan código de persistencia automáticamente a partir de metadatos, aplicando patrones de diseño.
- **Orientación a aspectos (AOP):** permite encapsular la persistencia como un "aspecto" transversal al sistema, desacoplando el código y promoviendo la modularidad y reutilización.
- **Lenguajes orientados a objetos**: utilizan funcionalidades del propio lenguaje para resolver la persistencia. Se subdividen en los siguientes:
    - **Librerías estándar**: uso de funcionalidades básicas del lenguaje (p. ej., manejo de archivos, serialización en Java).
    - **Lenguajes persistentes**: resuelven la persistencia de datos de forma transparente, adhiriéndose a los principios de persistencia ortogonal.
    - **Lenguaje de consulta integrado**: lenguajes de programación que integran lenguajes de consulta y administración de datos (p. ej., SQL, OQL, HQL).
- **Prevalencia**: propuesta alternativa que mantiene las instancias de objetos en memoria principal y realiza persistencia periódica mediante *snapshots* (serialización), y ofrece manejo de transacciones y recuperación del sistema.

CONTINUAR

2. DAO (Data Access Object)

En programación Java, DAO significa *Data Access Object* (objeto de acceso a datos), un patrón de diseño que aísla y abstrae la lógica para interactuar con una fuente de datos (como una base de datos) de la lógica de negocio de la aplicación. Permite un código más limpio, mantenible y flexible, al ocultar detalles específicos de la base de datos y facilitar operaciones CRUD (crear, leer, actualizar, borrar).

¿Para qué sirve?

Separación de responsabilidades:

separa la capa de acceso a datos de la lógica de negocio, siguiendo el principio de inversión de dependencias, para que los cambios en la base de datos no afecten el resto del programa.

Abstracción: —

proporciona una interfaz única para acceder a los datos, sin importar si provienen de una base de datos relacional (SQL), un archivo o un servicio externo.

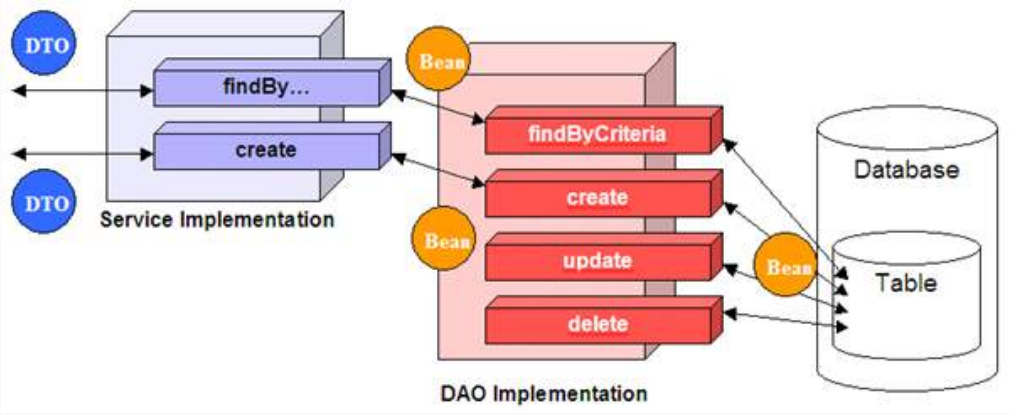
Mantenibilidad y flexibilidad: —

si cambias de base de datos, solo necesitas modificar la implementación del DAO, no la lógica de negocio que lo utiliza.

Operaciones CRUD: —

define métodos estándar (como guardar, buscar, actualizar, eliminar) para interactuar con los datos.

Figura 4: Implementación DAO



Fuente: [imagen sin título sobre implementación DAO] (s. f.).
https://intellect.icu/th/25/blogs/id9502/fd4c6855d3dc6bcf074127a1910d7d66.png

¿Cómo funciona?

- **Interfaz DAO**: define los métodos abstractos para las operaciones de datos (p. ej., UsuarioDAO con getUsuario(id)).
- **Implementación DAO**: una clase concreta (p. ej., UsuarioDAOImpl) implementa esa interfaz, y contiene el código específico para la base de datos (JDBC, JPA, etc.).
- **Lógica de negocio**: utiliza la interfaz DAO (no la implementación) para realizar operaciones, sin preocuparse por los detalles de la base de datos.

Ventajas:

- Ofrece una mayor modularidad y mantenibilidad del código.
- Facilita el cambio de la base de datos.
- Mejora la reutilización del código.
- Separa la lógica de negocio de la lógica de acceso a datos.

JDBC (*Java Database Connectivity*)

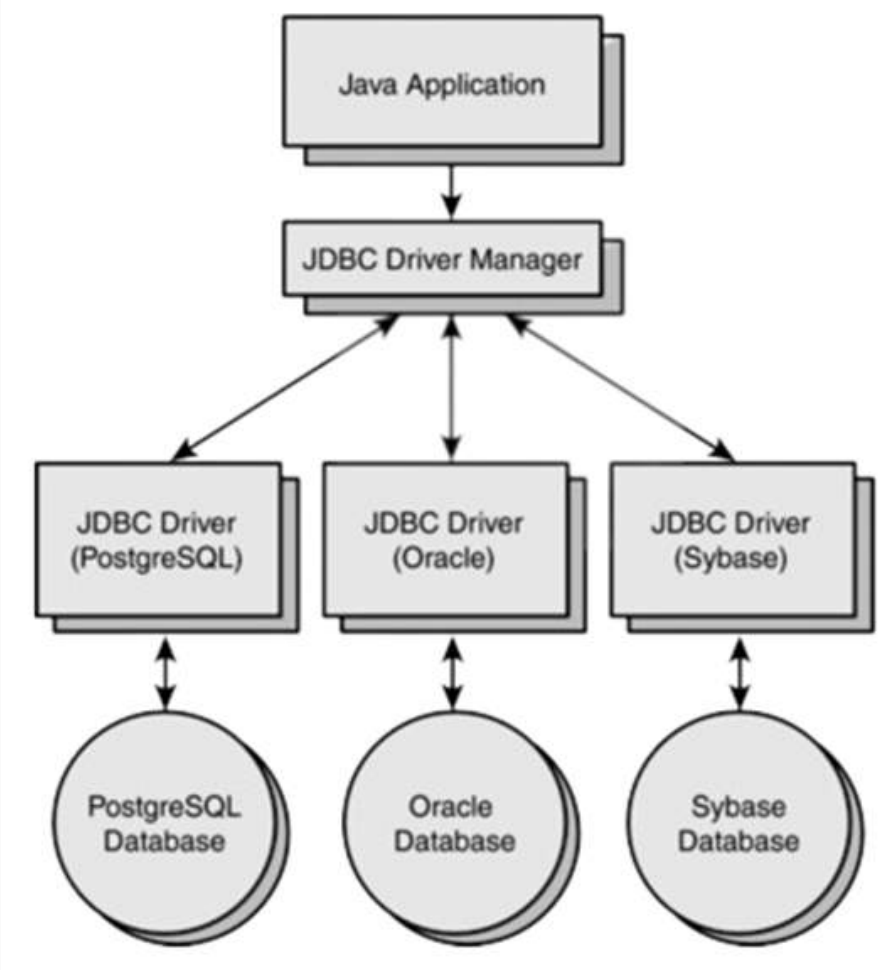
¿Qué es?

Es una API estándar de Java que proporciona un conjunto de clases e interfaces para interactuar con bases de datos.

¿Para qué sirve?

Permite a las aplicaciones Java establecer conexiones con bases de datos, ejecutar sentencias SQL y procesar los resultados.

Figura 5: JDBC



Fuente: [imagen sin título sobre JDBC] (s. f.). https://user-images.githubusercontent.com/59379254/79895890-fdf62700-83dd-11ea-8db0-2ee6ec27af6d.png

¿Cómo funciona?

- Se obtiene una conexión a la base de datos mediante un *driver* JDBC.
- Se crea una sentencia SQL.
- Se ejecuta la sentencia y se procesan los resultados.

Ventajas:

- Ofrece acceso a diversas bases de datos mediante un mismo conjunto de API.
- Tiene independencia del proveedor de la base de datos (usando *drivers* JDBC).
- Permite ejecutar consultas SQL directamente.

Relación entre DAO y JDBC

DAO utiliza JDBC para implementar las operaciones de acceso a datos. Por ejemplo, una clase DAO podría tener un método getUsuario(int id) que utiliza JDBC para realizar una consulta a la base de datos y devolver un objeto Usuario. La implementación de este método utilizaría las clases de la API JDBC para conectarse a la base de datos, ejecutar la consulta y procesar el resultado.

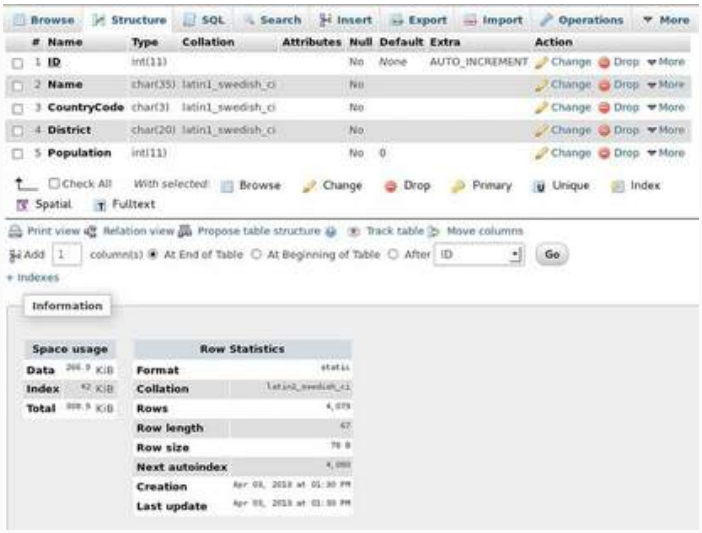
En resumen, DAO proporciona una capa de abstracción para el acceso a datos, y JDBC es la herramienta que utiliza Java para comunicarse con la base de datos.

Tabla y conjunto de tablas

Una tabla es una estructura que organiza los datos en filas y columnas, similar a una hoja de cálculo. Cada fila representa un registro, y cada columna representa un campo dentro de ese registro. Las tablas son fundamentales para almacenar y recuperar información de manera estructurada en un sistema de gestión de bases de datos.

- **Estructura**: las tablas se componen de filas (registros) y columnas (campos).
- **Filas (registros)**: cada fila contiene los datos específicos de un elemento o entidad. Por ejemplo, en una tabla de clientes, cada fila podría representar un cliente diferente con sus datos asociados.
- **Columnas (campos)**: cada columna representa un atributo o característica de la entidad. Por ejemplo, en una tabla de clientes, las columnas podrían ser "Nombre", "Apellido", "Dirección", "Teléfono", etc.
- **Almacenamiento**: las tablas son la forma más común de almacenar datos en una base de datos relacional.
- **Recuperación**: la información almacenada en las tablas se puede recuperar y manipular mediante consultas.
- **Importancia**: las tablas son esenciales para organizar, gestionar y analizar datos de manera eficiente dentro de una base de datos.

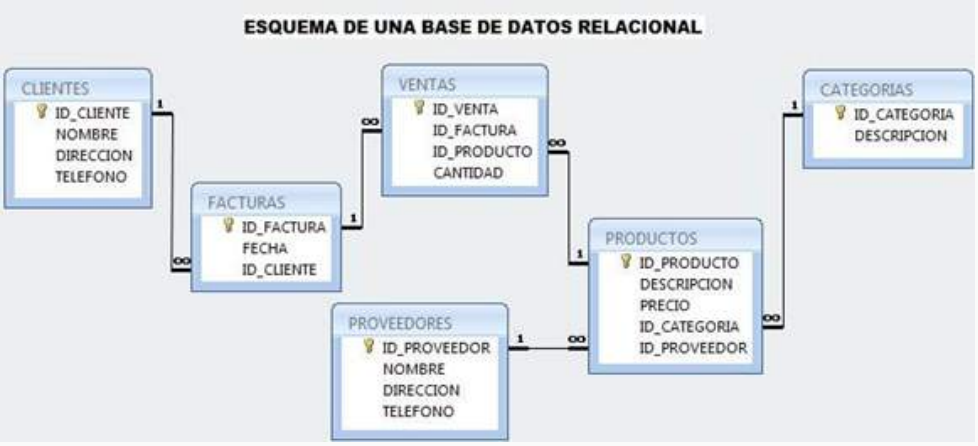
Figura 6: Tabla desarrollada con phpMyAdmin



Fuente: captura de pantalla de phpMyAdmin.

Aquí aparece, por lo general, otro concepto asociado, que es el modelo relacional de base de datos. Este, para el modelado y la gestión de bases de datos, es un modelo de datos basado en la lógica de predicados y en la teoría de conjuntos.

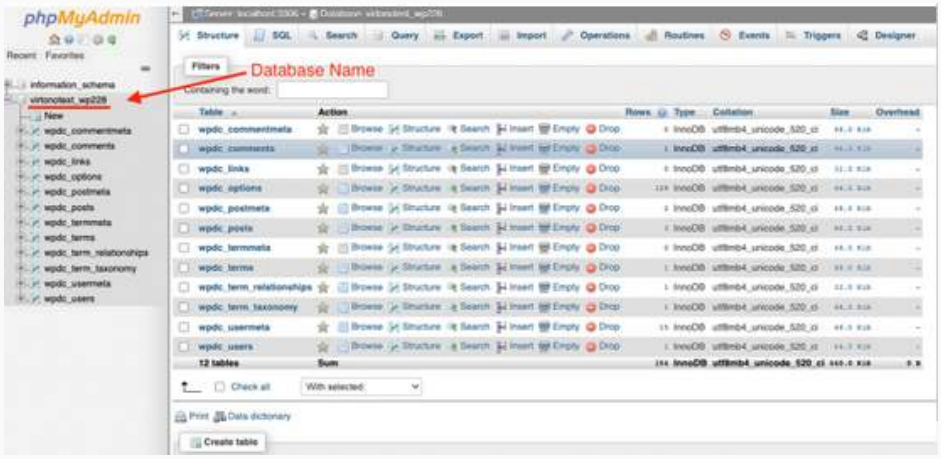
Figura 7: Esquema de una base de datos relacional



Fuente: captura de pantalla de phpMyAdmin.

Una base de datos relacional es un conjunto de una o más tablas estructuradas en registros (líneas) y campos (columnas), que se vinculan entre sí por un campo en común. En ambos casos posee las mismas características, como el nombre de campo, tipo y longitud. A este campo generalmente se lo denomina “ID”, “Identificador” o “Clave”; y a esta manera de construir bases de datos se la denomina “modelo relacional”.

Figura 7: Bases de datos y tablas en phpMyAdmin



Fuente: [imagen sin título sobre XXX] (s. f.). LINK

Estrictamente hablando, el término se refiere a una colección específica de datos. Sin embargo, a menudo se la usa de forma errónea como sinónimo del software usado para gestionar esa colección de datos. Ese software se conoce como “sistema gestor de base de datos relacional” (SGBD).

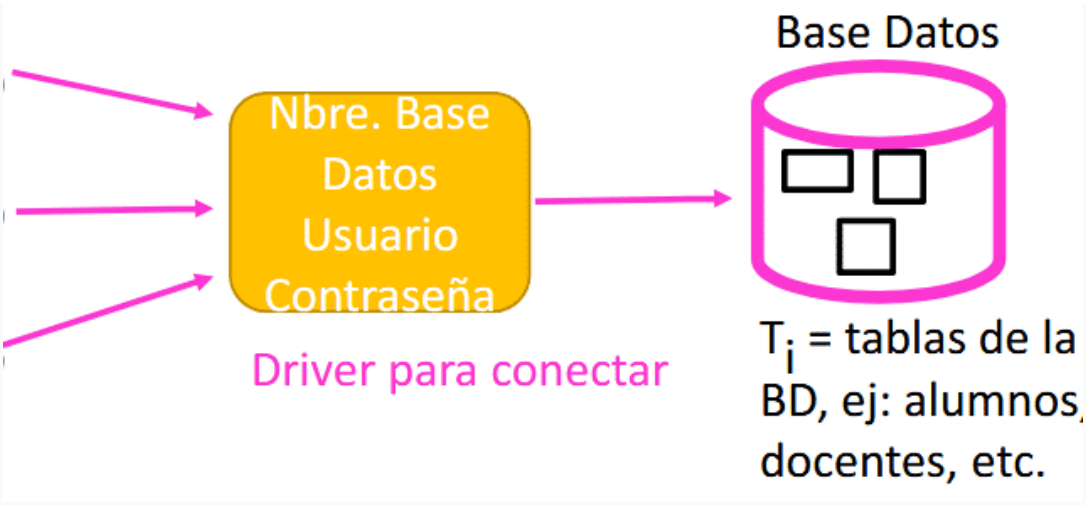
Las bases de datos relacionales pasan por un proceso conocido como normalización de una base de datos, el cual es entendido como el proceso necesario para que una base de datos sea utilizada de manera óptima.

El conjunto de tablas se puede asociar en un único componente, llamado base de datos.

Base de datos

Una base de datos es una colección organizada y estructurada de datos diseñada para almacenar, gestionar y recuperar información de manera eficiente. Piensa en ella como un sistema digital que permite guardar datos de forma organizada, como si fuera una gran hoja de cálculo con filas y columnas.

Figura 8: Base de datos

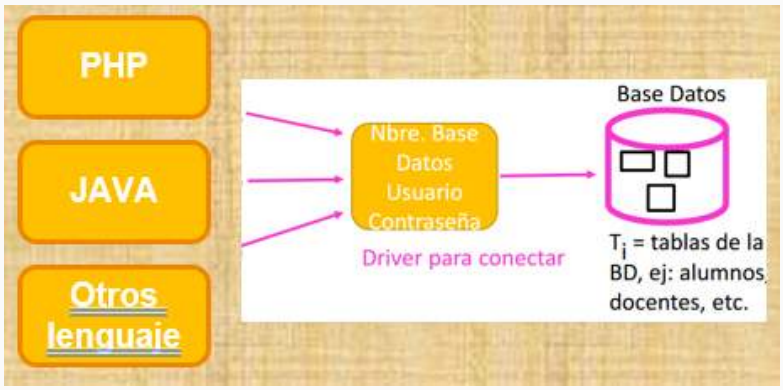


Fuente: elaboración propia.

Características principales

- **Almacenamiento estructurado**: los datos se organizan en tablas, con filas (registros) y columnas (campos).
- **Acceso y gestión**: un sistema de gestión de bases de datos (DBMS) facilita la interacción con la base de datos para realizar operaciones como consultar, modificar, actualizar o eliminar datos.
- **Eficiencia**: permiten buscar y recuperar información específica de forma rápida y precisa.
- **Escalabilidad**: pueden adaptarse a grandes volúmenes de datos y a crecimientos futuros.
- **Diversidad**: pueden contener diferentes tipos de datos, como texto, números, imágenes, videos, etc.

Figura 9: Lenguajes por medio de *drivers* que trabajan con BD



Fuente: elaboración propia.

Tipos de bases de datos

- **Relacionales**: los datos se organizan en tablas con relaciones predefinidas entre ellas.
- **NoSQL**: ofrecen mayor flexibilidad en la estructura de los datos y se adaptan mejor a datos no estructurados o semiestructurados.
- **Basadas en la nube**: se alojan y gestionan en la nube, lo que facilita la escalabilidad y la gestión.

En resumen, las bases de datos son herramientas fundamentales en la era digital para organizar y administrar información, tanto para pequeñas tareas como para grandes sistemas de gestión empresarial.

CONTINUAR

3. DAO y JDBC

Relación con UML

En UML, las clases y los objetos son conceptos fundamentales en el modelado orientado a objetos. Una clase es una plantilla o modelo para crear objetos, mientras que un objeto es una instancia específica de una clase. Por un lado, los diagramas de clases muestran la estructura estática de un sistema, incluyendo las clases, sus atributos, métodos y relaciones. Los diagramas de objetos, por otro lado, muestran instancias concretas de clases en un momento determinado, con valores específicos para sus atributos.

Clases: una clase es un plano o plantilla que define la estructura y el comportamiento de un conjunto de objetos.

- **Componentes**
  - **Atributos**: representan las características o propiedades de los objetos de la clase.
  - **Métodos**: representan las operaciones o acciones que los objetos de la clase pueden realizar.

- **Visualización**

Se representan en diagramas de clase como rectángulos divididos en tres secciones: nombre de la clase, atributos y métodos.

Objetos: un objeto es una instancia específica de una clase.

- **Componentes**
  - Atributos con valores: los objetos tienen valores específicos para sus atributos, a diferencia de las clases, que solo definen el tipo de atributos.

- **Visualización**

Se representan en diagramas de objetos como rectángulos con el nombre del objeto, seguido de dos puntos y el nombre de la clase, y subrayado. También se muestran los valores de sus atributos.

Relaciones entre clases y objetos

- **Diagramas de clases:** muestran las relaciones entre las clases, como herencia, asociación, agregación y composición.
- **Diagramas de objetos**: muestran las relaciones entre objetos específicos, como los que se crean a partir de las clases del diagrama de clases

Ahora, una referencia a un modelo UML para bases de datos, utilizando diagramas de clases, permite visualizar la estructura lógica de la base de datos, incluyendo tablas, atributos y relaciones. Este enfoque facilita la comprensión, el diseño y la comunicación del esquema de la base de datos, especialmente en proyectos complejos.

Conceptos clave

Diagramas de clases.

La herramienta principal para modelar la estructura de la base de datos en UML. Representan las tablas como clases, con sus atributos (columnas) y relaciones.

Clases.

Representan las tablas de la base de datos, con sus atributos (columnas) y métodos (opcionalmente).

Atributos.

Representan las columnas de las tablas, con sus tipos de datos y restricciones.

Relaciones.

Representan las conexiones entre tablas, como uno a uno, uno a muchos, o muchos a muchos, utilizando diferentes tipos de líneas y símbolos en el diagrama.

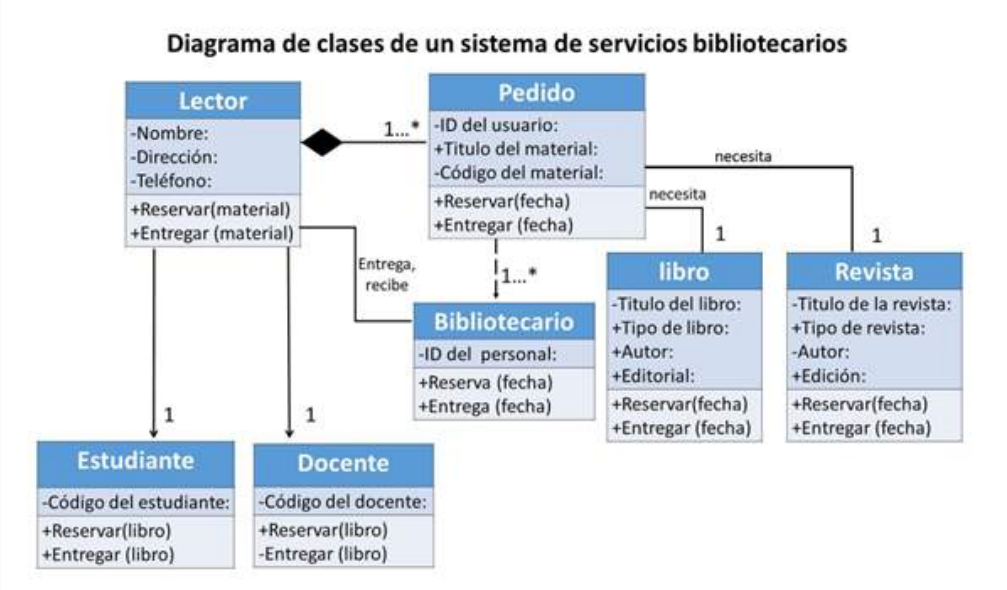
Beneficios de utilizar UML para modelado de bases de datos

- **Visualización**: permite una representación gráfica clara y concisa de la estructura de la base de datos.
- **Comunicación**: facilita la comunicación entre desarrolladores, analistas y usuarios finales sobre el diseño de la base de datos.

- **Diseño**: ayuda a diseñar la base de datos de manera efectiva, identificando posibles problemas o inconsistencias antes de la implementación.

- **Documentación**: sirve como documentación visual del diseño de la base de datos.

Figura 10: UML para bases de datos, utilizando diagramas de clases



Fuente: [imagen sin título sobre UML para bases de datos] (s. f.). https://miro.medium.com/v2/resize:fit:720/format:webp/1*oNhL8sgqrtS6_zwUZ9_dyg.png

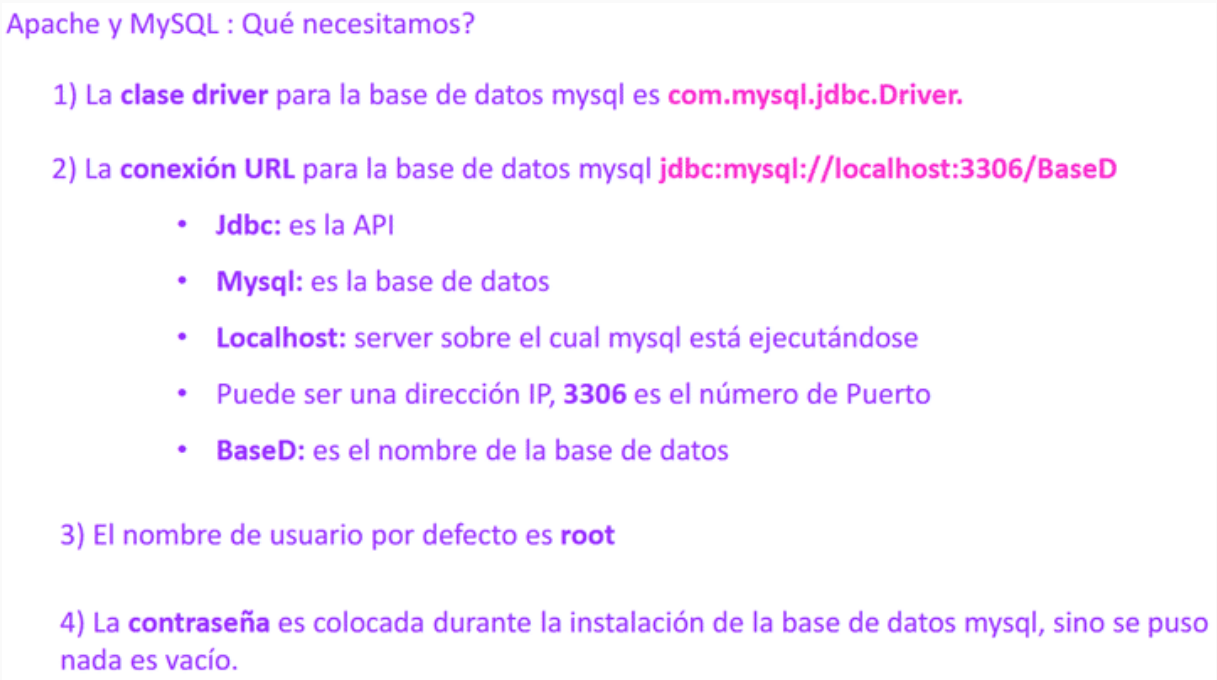
JDBC: conexión a BD

Para conectarse a una base de datos, es necesario realizar varios pasos, a saber:

1. Registrar el *driver*.

2. Obtener la conexión.
3. Crear declaración.
4. Ejecutar *query*.
5. Cerrar conexión.

Figura 11. Conexión a BD



Fuente: elaboración propia.

Aquí un ejemplo de conexión a una base de datos llamada BaseD, que tiene una tabla llamada emp.

Figura 12: Conexión a una base de datos

Fuente: elaboración propia.

JPA (Java Persistence API)

En el contexto de JPA, **@Entity** es una anotación que se utiliza para marcar una clase Java como una entidad, lo que significa que esta clase representa una tabla en una base de datos relacional. Cada instancia de la entidad corresponde a una fila en esa tabla, y sus atributos representan columnas.

Dicho de otra forma, @Entity indica que un objeto Java es una representación de datos persistentes en la base de datos.

Las características clave son las siguientes:

Persistencia: **@Entity** es parte de la API de JPA que facilita la persistencia de objetos Java en bases de datos. Sus instancias pueden guardarse, recuperarse o eliminarse de una base de datos.

Mapeo objeto-relacional: JPA se basa en el concepto de mapeo objeto-relacional (ORM), donde las clases Java (entidades) se mapean a tablas en la base de datos y los atributos de la clase se mapean a columnas en la tabla.

Identificador: una entidad JPA debe tener un identificador único, que se define con la anotación **@Id**. Este identificador se utiliza para distinguir entre diferentes instancias de la entidad. Dicho de otra forma, tiene un identificador (generalmente, una clave primaria).

Relaciones: las entidades pueden tener relaciones entre sí (uno a uno, uno a muchos, muchos a muchos), que también se definen con anotaciones JPA como **@OneToOne**, **@OneToMany**, **@ManyToMany**, etc.

Ejemplo: en este ejemplo, la clase "Empleado" está marcada como una entidad (@Entity) y tiene un identificador (@Id). Cada instancia de "Empleado" representará una fila en la tabla de empleados de la base de datos.

Figura 13: Ejemplo

```
@Entity
Public class Empleado {
@Id
Private Long id;
Private String nombre;
Private String apellido;

// Geters y setters
}
```

Anotación	Función
`@Entity`	Marca la clase como entidad.
`@Table(name="...")`	Especifica el nombre de la tabla en la BD (opcional si coincide).
`@Id`	Indica la clave primaria.
`@GeneratedValue`	Define cómo se genera el ID (ej: autoincremental).
`@Column(name="...")`	Mapea un atributo a una columna (opcional si nombres coinciden).

Fuente: elaboración propia.

Te invitamos a analizar el siguiente ejemplo observando cada línea y ver la nomenclatura en las asociaciones de la tabla, columnas, etc.

Figura 14: Ejemplo de código Java persistente

```java
@Table(name = "estudiantes")  // Tabla en la BD
public class Estudiante {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)  // Auto-incremental
    private Long id;

    @Column(name = "nombre_completo", length = 100, nullable = false)
    private String nombre;

    @Column(name = "edad")
    private int edad;

    // Constructor vacio (obligatorio para JPA)
    public Estudiante() {}

    // Constructor con parámetros
    public Estudiante(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }

    // Getters y Setters (obligatorios)
    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }

    public String getNombre() { return nombre; }
    public void setNombre(String nombre) { this.nombre = nombre; }

    public int getEdad() { return edad; }
    public void setEdad(int edad) { this.edad = edad; }
}
```

Fuente: elaboración propia.

Asociación entre clases (modelo de dominio)

Fundamentos de persistencia en Java: asociación, DAO y JDBC

Asociación entre clases (modelo de dominio)

Las asociaciones definen relaciones entre entidades (clases) y se reflejan en la base de datos.

Tabla 2: Tipos de asociaciones

Uno a Uno (@OneToOne)
Ejemplo: Persona ↔ Pasaporte.
 	
@Entity **public class Persona** {
 	
@Id **private Long** id;
 	
@OneToOne(mappedBy = "**persona**")
 	
private Pasaporte pasaporte;
 	
}
 	
@Entity **public class Pasaporte** {
 	
@Id **private Long** id;
	Uno a Muchos (@OneToMany)
	Ejemplo: Autor → List<Libro>.
	
	@Entity **public class Autor** {
	
	@Id **private Long** id;
	
	@OneToOne(mappedBy = "**autor**")
	
	private Pasaporte pasaporte;
	
	private PList<Libro> = new ArrayList<>();
	
	}

```
 @OneToOne

        @JoinColumn(name =
"persona_id")

  private Persona persona;

}
```

Muchos a Uno (@ManyToOne)

Ejemplo: Libro → Autor

```
@Entity

@Table(name = “libros”)

public class Libro {

  @Id

@GeneratedValue


private Long id;
```

Muchos a Muchos (@ManyToMany)

Ejemplo: Estudiante ↔ Curso.

```
@Entity public class Estudiante {

@Id private Long id;

@ManyToMany

@JoinTable(name =
"estudiante_curso",

joinColumns = @JoinColumn(name =
"estudiante_id"),

inverseJoinColumns =
@JoinColumn(name = "curso_id"))
```

```
private String titulo;

@ManyToOne

    @OneToOne(mappedBy = "autor_id")

  private Autor autor;

}
```

```
private Set<Curso> cursos = new HashSet<>();

}
```

Fuente: elaboración propia.

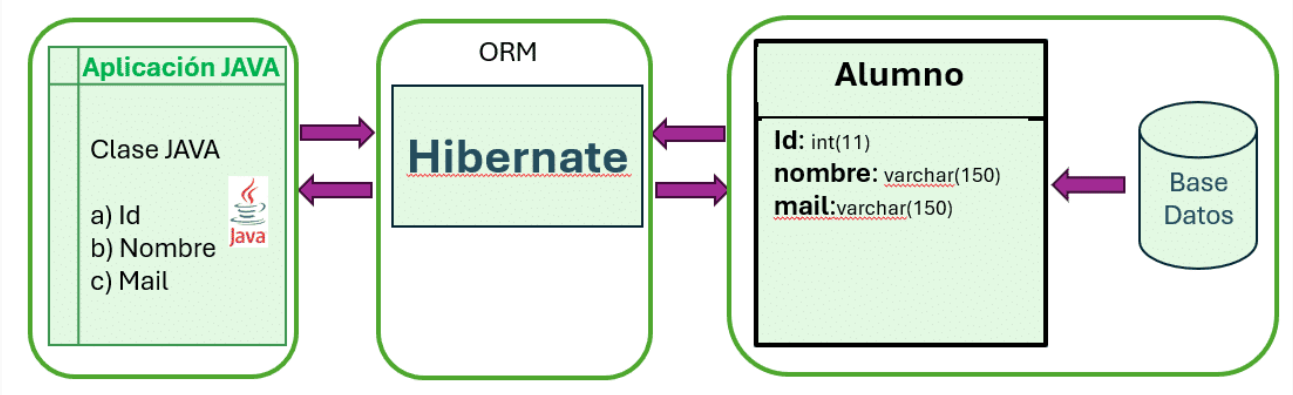
CONTINUAR

4. HIBERNATE

Hibernate es un *framework* de código abierto para Java que actúa como un mapeador objeto-relacional (ORM), traduciendo clases Java y objetos en tablas y filas de bases de datos relacionales, y viceversa, para que puedas trabajar con datos usando objetos POO en lugar de escribir SQL manual. Esto simplifica la persistencia de datos, mejora la productividad y permite portabilidad entre diferentes bases de datos.

Hibernate facilita la interacción entre aplicaciones Java y bases de datos relacionales. Simplifica el acceso a datos eliminando la necesidad de escribir código SQL repetitivo. Hibernate asigna objetos Java a tablas de base de datos y tipos de datos Java a tipos de datos SQL.

Figura 15: ***Framework*** **Hibernate**



Fuente: elaboración propia.

En resumen, Hibernate es un puente entre el código Java y la base de datos que te permite trabajar directamente con objetos, en lugar de consultas SQL.

Más detalles:

- **Mapeo objeto-relacional (ORM):** gestiona la conversión entre objetos Java y registros de base de datos, y viceversa.
- **Simplificación de código**: permite realizar operaciones CRUD sin escribir código SQL complejo.
- **Independencia de la base de datos**: puede funcionar con diferentes bases de datos relacionales, lo que facilita la portabilidad de la aplicación.
- **Consultas HQL**: proporciona un lenguaje de consulta propio, HQL (*Hibernate Query Language*), que es más orientado a objetos que SQL.
- Transacciones: facilita la gestión de transacciones, lo que asegura la consistencia de los datos.
- **Caché**: ofrece mecanismos de caché para mejorar el rendimiento de las aplicaciones.

Ejemplo: imagina que tienes una clase Cliente en Java y una tabla clientes en tu base de datos. Hibernate permite que mapees el objeto Cliente a la tabla clientes de forma que, cuando guardes un objeto Cliente,

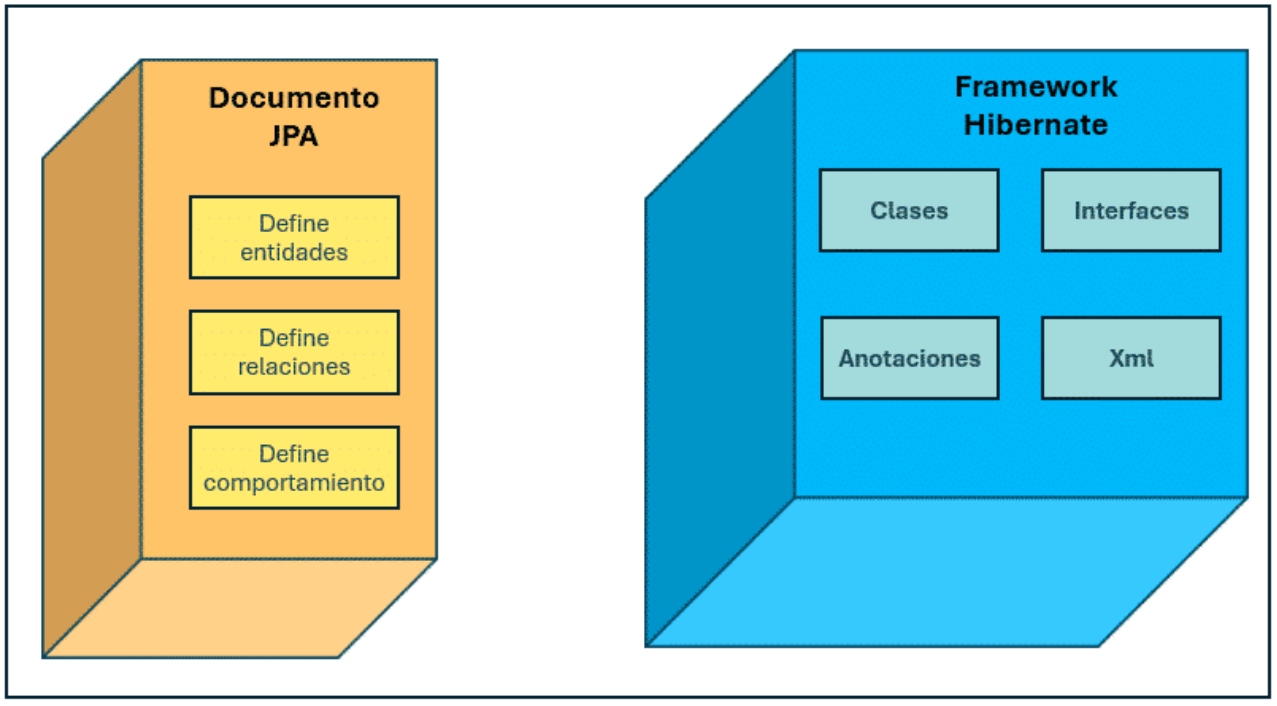
Hibernate se encargue de generar la sentencia SQL INSERT correspondiente.

JPA vs. Hibernate

Como podemos ver, Hibernate es una herramienta poderosa para simplificar el desarrollo de aplicaciones Java que interactúan con bases de datos, y permite enfocarse en la lógica de negocio, en lugar de la complejidad de las operaciones de base de datos.

En el ecosistema de persistencia de datos en aplicaciones Java, JPA e Hibernate tienen una relación estrecha pero importante de diferenciar.

Figura 16: JPA y *framework* Hibernate



Fuente: elaboración propia.

JPA es una especificación que define una serie de interfaces y anotaciones para facilitar el acceso y el mapeo entre los objetos de Java y las bases de datos relacionales. A través de JPA, se estandarizan operaciones comunes de persistencia como el mapeo de clases a tablas y la gestión del ciclo de vida de las entidades. Por ejemplo, algunas de las anotaciones más comunes que ofrece JPA son @Entity, @Id y @GeneratedValue(strategy = GenerationType.IDENTITY). Sin embargo, JPA no es una implementación en sí misma: requiere de un proveedor que la implemente, como Hibernate, EclipseLink o TopLink.

Por otro lado, Hibernate es un *framework* ORM que implementa JPA, pero va más allá. Hibernate no solo ofrece la funcionalidad estándar de JPA, sino que también agrega características avanzadas que lo hacen una opción poderosa para gestionar la persistencia en aplicaciones Java. Algunos ejemplos:

- Su propio concepto de Session, que gestiona el ciclo de vida de las entidades y permite realizar operaciones CRUD.
- Cachés de primer y segundo nivel, que optimizan el acceso a los datos y reducen las consultas innecesarias a la base de datos.
- La capacidad de definir cómo se cargan las relaciones entre entidades mediante *lazy loading* (carga diferida) o *eager fetching* (carga inmediata), lo que ayuda a controlar el rendimiento.
- Un lenguaje de consultas propio, el *Hibernate Query Language* (HQL), que ofrece más flexibilidad que JPQL.
- Soporte nativo para auditoría y versionamiento, que permite rastrear cambios en las entidades a lo largo del tiempo y gestionar concurrencia de

manera efectiva.

Es importante señalar que cuando agregamos la dependencia de Spring Data JPA en un proyecto Spring Boot, Hibernate es el proveedor predeterminado que se incluye, lo que simplifica aún más la configuración de persistencia.

Mientras que JPA establece las bases y estandariza cómo interactuar con las bases de datos, Hibernate lleva esa funcionalidad a un nivel superior, con características adicionales que hacen que el trabajo con datos en aplicaciones Java sea más eficiente y flexible.

CONTINUAR

Referencias

[Imagen sin título sobre esquema de una base de datos relacional]. (s. f.). https://www.liveworksheets.com/worksheet/es/informatica/7139142

[Imagen sin título sobre instancia objeto y tabla]. (s. f.). https://img.notionusercontent.com/s3/prod-files-secure%2F1f0b2aad-4a26-4d8f-9617-7b97baea2107%2Ff793034e-9477-4d7e-8a15-f2057defdd2e%2Fimpedance_mismatch.svg/size/?exp=1767807511&sig=HithfV1aaO1M6L3AmB6Qg9Qj9jF5Vdzg46s1kSH4Xno&id=79b99836-95fc-454f-8bb1-a2a9b2f61f12&table=block

[Imagen sin título sobre implementación DAO]. (s. f.). https://intellect.icu/th/25/blogs/id9502/fd4c6855d3dc6bcf074127a1910d7d66.png

[Imagen sin título sobre JDBC]. (s. f.). https://user-images.githubusercontent.com/59379254/79895890-fdf62700-83dd-11ea-8db0-2ee6ec27af6d.png

[Imagen sin título sobre UML para bases de datos]. (s. f.). https://miro.medium.com/v2/resize:fit:720/format:webp/1*oNhL8sgqrtS6_zwUZ9_dyg.png

Vasen, F. (2013). Las políticas científicas de las universidades nacionales argentinas en el sistema científico nacional. *Ciencia, docencia y tecnología*, (46). 9-32.

https://www.scielo.org.ar/scielo.php?script=sci_arttext&pid=S1851-17162013000100001

CONTINUAR

Descarga en PDF