

UNIDAD DIDÁCTICA 8

GESTIÓN DE BASES DE DATOS

**MÓDULO PROFESIONAL:
PROGRAMACIÓN**



CESUR
Tu Centro Oficial de FP

ÍNDICE

RESUMEN INTRODUCTORIO	2
INTRODUCCIÓN	2
CASO INTRODUCTORIO	3
1. BASES DE DATOS: CONCEPTOS, UTILIDADES Y TIPOS	4
1.1 Concepto y utilidad de una base de datos.....	5
1.2 Bases de datos orientadas a objeto.....	6
1.3 Bases de datos relacionales	9
2. ACCESO A BASES DE DATOS RELACIONALES	11
3. CONEXIÓN CON BASES DE DATOS RELACIONALES. CARACTERÍSTICAS, TIPOS Y MÉTODOS DE ACCESO	14
3.1 Tipo 1: Driver puente JDBC-ODBC	16
3.2 Tipo 2: Driver API Nativo / Parte JAVA.....	18
3.3 Tipo 3: Driver protocolo de red / Todo JAVA.....	20
3.4 Tipo 4: Driver protocolo Nativo / Todo JAVA.....	22
4. PREPARACIÓN DEL ENTORNO DE DESARROLLO.....	24
5. ESTABLECIMIENTO DE CONEXIONES. COMPONENTES DE ACCESO A DATOS.....	31
5.1 Los URL de JDBC.....	32
5.2 Clase drivermanager	32
6. RECUPERACIÓN DE INFORMACIÓN. SELECCIÓN DE REGISTROS. USO DE PARÁMETROS	36
7. MANIPULACIÓN DE LA INFORMACIÓN. ALTAS, BAJAS Y MODIFICACIONES. EJECUCIÓN DE CONSULTAS SOBRE LA BASE DE DATOS	39
RESUMEN FINAL	57

RESUMEN INTRODUCTORIO

En esta unidad se va a tratar en primer lugar el concepto y utilidad de una base de datos, haciendo una distinción entre las bases de datos relacionales y las bases de datos orientadas a objeto. Se verá la persistencia de objetos y las tecnologías utilizadas para traducir los objetos generados con una aplicación desarrollada en un lenguaje de programación orientado a objetos, a una base de datos relacional mediante un ORM. Se introducirá el concepto de driver para realizar una conexión con bases de datos.

Trabajaremos con bases de datos relacionales por lo que aprenderemos a configurar el entorno para poder trabajar con una base de datos en local utilizando XAMPP y MySQL. Pasaremos después a ver las características, tipos y métodos de acceso a una base de datos desde Java. Posteriormente, veremos cómo establecer una conexión para pasar después a tratar cómo se recupera la información. Y, finalmente, veremos cómo manipular la información, la realización de altas, bajas y modificaciones de datos. Para cada uno de los apartados veremos ejemplos en el lenguaje de programación Java.

INTRODUCCIÓN

En el mundo de la informática hay numerosos estándares y lenguajes, la mayoría de los cuales son incapaces de comunicarse entre sí. Afortunadamente, algunos de ellos son verdaderos referentes cuyo conocimiento es vital para los programadores. El **Structured Query Language** o SQL, se ha convertido en los últimos años en el método estándar de acceso a bases de datos. Se puede decir que prácticamente cualquier Sistema de Gestión de Bases de Datos creado en los últimos años usa SQL. Esta es la principal virtud de SQL: es un lenguaje prácticamente universal dentro de las bases de datos.

Tener conocimientos de SQL es una necesidad para cualquier profesional de las tecnologías de la información (TI). A medida que el desarrollo de sitios web se hace más común entre personas sin conocimientos de programación, el tener una cierta noción de SQL se convierte en requisito indispensable para integrar datos en páginas HTML. No obstante, éste no es el único ámbito de SQL, puesto que la mayor parte de las aplicaciones de gestión que se desarrollan hoy en día acceden, en algún momento, a alguna base de datos utilizando sentencias SQL.

CASO INTRODUCTORIO

La aplicación que estás desarrollando para un importante cliente necesita en esta fase de desarrollo integrar el acceso a bases de datos para poder almacenar y recuperar los datos. Nuestra aplicación está siendo desarrollada en Java y decidimos usar la API de JDBC, ya que nuestro cliente no tiene claro aún si va a usar MySQL o PostgreSQL como sistema gestor de bases de datos. De esta forma, podremos acceder a cualquiera de estos sistemas de bases de datos sin necesidad de tener que modificar nuestra aplicación.

Al finalizar el estudio de la unidad, serás capaz de realizar la conexión con distintos tipos de bases de datos gracias a la API JDBC; conocerás los tipos de drivers JDBC existentes; sabrás configurar el entorno de desarrollo para usar bases de datos relacionales con un servidor local y sabrás realizar consultas y modificaciones sobre bases de datos relacionales utilizando la API JDBC.

1. BASES DE DATOS: CONCEPTOS, UTILIDADES Y TIPOS

A medida que avanzas en el desarrollo de tu aplicación en Java para este importante cliente, llega el momento de integrar el acceso a bases de datos. Habías tomado la decisión de utilizar la API de JDBC para mantener la flexibilidad, ya que tu cliente aún no ha decidido el sistema gestor de bases de datos. Esta elección te permite adaptarte sin problemas a cualquier opción. Pero en esta fase, es fundamental comprender la importancia de la persistencia de datos. La persistencia garantiza que la información que almacenas se conserve de forma sólida a lo largo del tiempo y esté disponible cuando sea necesaria. Es el pilar de la integridad y la continuidad de los datos en tu aplicación y te permitirá acceder y manipular datos de manera constante en la base de datos.

La **persistencia** de datos se refiere a la capacidad de almacenar datos de forma permanente, de manera que los datos no se pierdan cuando se apague un sistema o una aplicación. La persistencia es fundamental en el desarrollo de software y sistemas de información, ya que permite que los datos se conserven a lo largo del tiempo y estén disponibles para futuras consultas y manipulaciones.

Sin embargo, puede haber una discrepancia entre el modelo de datos de un sistema de gestión de bases de datos relacional (RDBMS) y el modelo de datos de una aplicación orientada a objetos, que se conoce como desfase objeto-relacional. Este desfase surgirá, cuando se trabaja con objetos en un lenguaje orientado a objetos como Java y necesitamos interactuar con una base de datos relacional, como MySQL, PostgreSQL o SQL Server.

En una base de datos relacional, los datos se almacenan en tablas con filas y columnas, y se utilizan consultas SQL para acceder a ellos. Sin embargo, en la programación orientada a objetos, los datos se modelan como objetos con propiedades y métodos.

Esta diferencia en los modelos de datos puede dar lugar a varios desafíos, tales como:

- **Mapeo objeto-relacional:** Para superar el desfase, se puede utilizar una técnica llamada "mapeo objeto-relacional" (ORM), que consiste en mapear objetos de la aplicación a las tablas de la base de datos y viceversa, usando bibliotecas ORM, como Hibernate para Java o Entity Framework para .NET que proporcionan una capa de abstracción para el desarrollador, lo que simplifica el mapeo de objetos a tablas y facilita la interacción con la base de datos relacional.

- **Dificultades en la consulta:** Las consultas SQL difieren de las consultas orientadas a objetos. Las consultas SQL suelen ser más orientadas a conjuntos y tablas, mientras que en la programación orientada a objetos se trabaja con objetos individuales y relaciones entre ellos. El mapeo objeto-relacional (ORM) resuelve este problema actuando como un intermediario que convierte las operaciones orientadas a objetos en consultas SQL comprensibles para la base de datos sin que el desarrollador tenga que escribirlas explícitamente.
- **Diferencias de tipos de datos:** Los RDBMS suelen tener tipos de datos específicos, como enteros, cadenas y fechas, mientras que los lenguajes orientados a objetos pueden tener tipos más complejos y personalizados. El ORM debe manejar la conversión entre estos tipos de datos.
- **Transacciones y concurrencia:** Las bases de datos relacionales gestionan las transacciones y la concurrencia de manera diferente a la programación orientada a objetos. El ORM debe administrar estos aspectos para garantizar la consistencia de los datos.

Por tanto, el **desfase objeto-relacional** es un desafío común en el desarrollo de aplicaciones que requieren el uso de bases de datos relacionales y programación orientada a objetos. Las herramientas ORM proporcionan una solución efectiva para abordar este desfase y permiten a los desarrolladores trabajar con objetos en su código mientras interactúan con bases de datos relacionales en segundo plano.

1.1 Concepto y utilidad de una base de datos

Una base de datos es una colección estructurada y organizada de datos que se almacenan de manera persistente en un sistema informático. Estos datos pueden representar información sobre cualquier dominio, desde registros de clientes y transacciones financieras hasta información científica y médica. Una base de datos está diseñada para permitir la entrada, almacenamiento, recuperación, actualización y gestión de datos de manera eficiente.

Las bases de datos se utilizan para almacenar, organizar y gestionar grandes cantidades de datos, lo que permite el acceso rápido y eficiente a la información y garantiza la seguridad y privacidad de los datos almacenados.

Si bien hay varios tipos de bases de datos como son, relacionales, orientadas a objetos, NoSQL, etc. Pero existen algunas características estándar que se aplican en general:

- **Modelo de Datos:** Cada base de datos sigue un modelo de datos específico que define cómo se organizan y se relacionan los datos. Como, por ejemplo, el modelo relacional, el modelo jerárquico, el modelo de grafo, etc.
- **Lenguaje de Consulta:** Las bases de datos tienen un lenguaje de consulta que permite realizar operaciones como buscar, filtrar, actualizar y eliminar datos. En bases de datos orientadas a objeto tenemos OQL (Object Query Language), LINQ (Language Integrated Query), etc. Y en bases de datos relacionales, el lenguaje SQL es ampliamente utilizado.
- **Sistema de Gestión de Bases de Datos (DBMS):** Un DBMS es un software que gestiona la creación, acceso y administración de la base de datos. Ejemplos incluyen MySQL, PostgreSQL, Oracle, MongoDB, entre otros.
- **Controladores/drivers de bases de datos:** Los controladores de bases de datos son programas que se utilizan para comunicarse con la base de datos desde una aplicación.
- **Integridad de Datos:** Las bases de datos mantienen la integridad de los datos mediante restricciones, claves primarias, claves foráneas y/o reglas de validación.
- **Seguridad:** Los sistemas de bases de datos proporcionan mecanismos de seguridad, como autenticación, autorización y cifrado, para proteger los datos.
- **Transacciones:** Las bases de datos suelen admitir transacciones para garantizar la consistencia y la atomicidad de las operaciones.

Estas características estándar forman la base de todas las bases de datos, aunque las implementaciones específicas pueden variar según el tipo y la tecnología utilizada.

1.2 Bases de datos orientadas a objeto

Las Bases de Datos Orientadas a Objetos (BDOO) o Bases de Objetos, se integran directa y perfectamente con aplicaciones escritas en lenguajes orientados a objetos porque admiten un modelo de objetos puro y son adecuados para almacenar y recuperar datos complejos, lo que permite a los usuarios navegar directamente (sin mapeo entre diferentes representaciones de la base de datos y la aplicación).

Las BDOO admiten funciones orientadas a objetos como agregación, encapsulación, polimorfismo y herencia. Los objetos pueden ser de cualquier complejidad para que contengan toda la información necesaria para describir el objeto. Todo esto hace que BDOO sea un ideal para todo programador.

Pero a pesar de tantas ventajas y años de experiencia tienen una serie de limitaciones. Algunas de las principales razones para que no sean ampliamente utilizadas en el mercado son las siguientes:

- **Falta de un modelo de datos común y universal:** No existen modelos de datos universalmente aceptados y la mayoría de los modelos no están respaldados por una base teórica sólida y bien establecida. Aunque se ha tratado de estandarizar OQL (Object Query Language), dicho lenguaje de consulta aún no existe en muchos BDOO. OQL no tiene el mismo soporte que SQL y cada empresa de BDOO ofrece su propia alternativa.
- **Competencia:** Las bases de datos relacionales y objeto-relacionales es una solución que se ha vuelto más poderosa debido al mapeo relacional de objetos que permite un rápido desarrollo y que es proporcionada por diferentes tecnologías, entre las que cabe destacar; Hibernate, Entity Framework, EclipseLink, Django ORM, SQLAlchemy, Doctrine o Active Record, entre otros. Esto se une a la aparición de otros modelos de bases de datos NoSQL más simples y flexibles, como, por ejemplo, las bases de datos de archivos como MongoDB o bases de datos de clave-valor como Redis. La flexibilidad y extensibilidad de estos modelos de datos han reemplazado a las bases de datos orientadas a objetos.

Algunos ejemplos de BDOO son:

- **db4o:** db4o es un sistema de gestión de bases de datos orientada a objetos de código abierto que se utiliza en aplicaciones Java y .NET. Permite a los desarrolladores almacenar objetos Java o .NET directamente en la base de datos sin necesidad de mapearlos a tablas, lo que facilita la persistencia de objetos complejos.
- **Zope Object Database (ZODB):** ZODB es una base de datos orientada a objetos utilizada en el entorno del servidor de aplicaciones web 'Zope'. Almacena objetos Python y se integra de manera natural con el entorno de desarrollo de Zope, aunque también puede utilizarse independientemente.

- **Versant Object Database:** Versant es una base de datos orientada a objetos que se ha utilizado en aplicaciones empresariales para almacenar y administrar datos complejos y estructuras de datos jerárquicas.
- **ObjectDB:** ObjectDB es una base de datos orientada a objetos diseñada para aplicaciones C, C++, Smalltalk, C#, .NET y Java. Al igual que db4o, permite a los desarrolladores almacenar objetos Java directamente en la base de datos.

En las bases de datos orientadas a objetos, se pueden almacenar objetos anidados o también documentos JSON que contengan colecciones de valores.

En cuanto a la **persistencia de objetos**, es un concepto ligado a la programación orientada a objetos y se refiere a la capacidad de mantener la información de objetos a lo largo del tiempo, más allá de la vida útil de la ejecución del programa. En esencia, implica guardar objetos en una forma que pueda recuperarse posteriormente, lo que es esencial para muchas aplicaciones, como sistemas de gestión de bases de datos, aplicaciones empresariales y cualquier software que requiera almacenar y recuperar datos de manera confiable.

En sistemas de bases de datos orientadas a objetos, la persistencia de objetos es más natural, ya que el modelo de datos de la base de datos se alinea directamente con el modelado de objetos en la aplicación.

Como se ha comentado anteriormente es bastante común en aplicaciones empresariales y sistemas de bases de datos, utilizar un mapeo objeto-relacional para traducir objetos en memoria a estructuras de datos relacionales (base de datos relacional).

Estándar para el mapeo objeto-relacional (ORM):

JPA (Java Persistence API) es una API de Java que proporciona un estándar para el mapeo objeto-relacional (ORM) en aplicaciones Java. Esta API facilita la interacción entre aplicaciones Java y bases de datos relacionales al permitir que los desarrolladores trabajen con objetos en lugar de escribir consultas SQL directamente. Permitiendo mapear objetos de Java en una base de datos relacional y viceversa. Las clases que se asignan a tablas de base de datos se denominan “entidades”, donde cada instancia de una entidad representa una fila en la tabla correspondiente.

Se introduce el concepto de **EntityManager** que es una interfaz que se encarga de administrar el ciclo de vida de las entidades, realizar operaciones de persistencia (almacenamiento y recuperación) y mantener la coherencia de los datos.

JPA utiliza **JPQL** (Java Persistence Query Language), que es un lenguaje de consultas similar a SQL, pero orientado a objetos. Con JPQL, puedes realizar consultas para recuperar objetos de la base de datos de manera más abstracta y orientada a objetos. JPA admite relaciones entre entidades, como relaciones uno a uno, uno a muchos y muchos a muchos. Esto permite modelar relaciones complejas en la base de datos mediante objetos Java.

Existen varias implementaciones de JPA, como Hibernate, EclipseLink y OpenJPA que son de las más populares. Estas implementaciones se encargan de traducir las operaciones de JPA a consultas SQL específicas del sistema de gestión de bases de datos subyacente. La elección depende de las necesidades específicas del proyecto y las preferencias del desarrollador.

1.3 Bases de datos relacionales

Las bases de datos relacionales (RDBMS, por sus siglas en inglés de Relational Database Management System) son un tipo de sistema de gestión de bases de datos que utilizan un modelo de datos relacional para organizar y gestionar la información y se basa en los siguientes componentes:

- **Modelo de datos relacional:** Organiza la información en tablas compuestas por filas y columnas. Cada tabla representa una entidad, como clientes, productos o pedidos, las filas contienen registros específicos de esas entidades, mientras que las columnas representan atributos o características de los registros. Cada registro se identifica por una clave o índice que se sirve para acelerar las búsquedas en la base de datos. Se crean a partir de uno o más campos de una tabla y se utilizan para encontrar registros específicos en la tabla más rápidamente.
- **Integridad de Datos:** Las bases de datos relacionales hacen un énfasis en mantener la integridad de los datos. Esto se logra mediante la aplicación de restricciones, como claves primarias y foráneas, para garantizar la consistencia y la precisión de los datos.
- **Lenguaje SQL:** Principalmente para interactuar con bases de datos relacionales, se utiliza el lenguaje SQL (Structured Query Language). Permite realizar una amplia variedad de operaciones, como consultas (SELECT), inserciones (INSERT), actualizaciones (UPDATE) y eliminaciones (DELETE) de datos.

- **Transacciones:** Las bases de datos relacionales admiten transacciones, lo que significa que las operaciones de escritura se realizan de manera atómica, es decir, todas o ninguna de las operaciones se realizan, y se pueden deshacer si ocurren errores. Esto garantiza la coherencia de los datos.
- **Normalización:** La normalización es un proceso que se utiliza para organizar las tablas de manera eficiente y reducir la redundancia de datos. Ayuda a evitar problemas como la actualización anómala y la pérdida de consistencia en los datos.

En los próximos apartados de esta unidad aprenderás a conectar tu aplicación Java con una base de datos relacional, pero sin hacer uso de ORM's.



PARA SABER MÁS

Aquí puedes ver cómo utilizar JPA en un pequeño proyecto, pero es recomendable que sigas con tu aprendizaje y aprendas primero a realizar el mapeo objeto-relacional sin usar un ORM.



2. ACCESO A BASES DE DATOS RELACIONALES

El siguiente paso en el desarrollo de tu aplicación en Java es crucial y consistirá en integrar el acceso a bases de datos de manera eficiente y adaptable. Te enfrentas al desafío de elegir y utilizar los drivers JDBC adecuados que son piezas clave que facilitarán la comunicación entre tu aplicación Java y la base de datos. El acceso a la base de datos con drivers es un componente esencial que garantizará que tu aplicación pueda almacenar y recuperar datos de forma confiable y eficaz.

Los controladores de bases de datos son programas que se utilizan para comunicarse con la base de datos desde una aplicación. Los controladores de bases de datos proporcionan una interfaz o medio para que la aplicación pueda enviar y recibir datos desde la base de datos.

Existen tipos diferentes de drivers o controladores dependiendo del lenguaje de programación y la base de datos utilizada. Para bases de datos **relacionales** tenemos los siguientes:

- **JDBC(Java Database Connectivity):** API específica para Java que permite a las aplicaciones Java interactuar con bases de datos relacionales. Proporciona un conjunto de clases e interfaces que permiten conectarse a la base de datos, enviar consultas SQL, procesar resultados y administrar transacciones. A través de JDBC, puedes acceder a bases de datos como MySQL, PostgreSQL, Oracle y SQL Server. El driver JDBC es comúnmente utilizado para conectarse a bases de datos MySQL desde aplicaciones Java.
- **Conectores específicos utilizados con bases de datos MySQL:** Existen diferentes conectores o drivers específicos según el lenguaje utilizado en el desarrollo de una aplicación que permiten conectarse a bases de datos MySQL, como son los siguientes:
 - Conector .NET: utilizado para conectarse a bases de datos MySQL desde aplicaciones desarrolladas en C# o Visual Basic .NET.
 - Conector para Python: utilizado para conectarse a bases de datos MySQL desde aplicaciones Python.
 - Conector para PHP: utilizado para conectarse a bases de datos MySQL desde aplicaciones PHP.
 - Conector para Ruby: utilizado para conectarse a bases de datos MySQL desde aplicaciones Ruby.

Cada uno de estos conectores está diseñado específicamente para el lenguaje de programación que se está utilizando, y proporciona una API que permite a los desarrolladores interactuar con una base de datos específica desde la aplicación. Sobre los datos contenidos en la base de datos realizaremos operaciones de alta, baja, edición o consulta de registros.

DRIVER JDBC

La API JDBC viene distribuida en dos paquetes principales dentro de Java SE y son:

- **Paquete `java.sql`:** Que contiene las clases e interfaces principales de la API JDBC, diseñado para facilitar el acceso y procesamiento de los datos que provienen de diversas fuentes, por lo general bases de datos relacionales. Aquí encontraremos clases clave, como 'Connection', 'Statement', 'ResultSet', y otras interfaces y clases relacionadas con la interacción con bases de datos.
- **Paquete `javax.sql`:** Este paquete, complementa a las clases incluidas en `java.sql`. Aunque su propósito principal es facilitar el acceso a fuentes de datos en el entorno de servidor, también incluye algunas clases e interfaces que son útiles en otros contextos. A partir de Java SE 1.4, muchas de las clases de `javax.sql` se incluyeron en la distribución estándar de Java (Java SE) lo cual facilitó el uso, eliminando la necesidad de agregar algunas bibliotecas o extensiones adicionales.

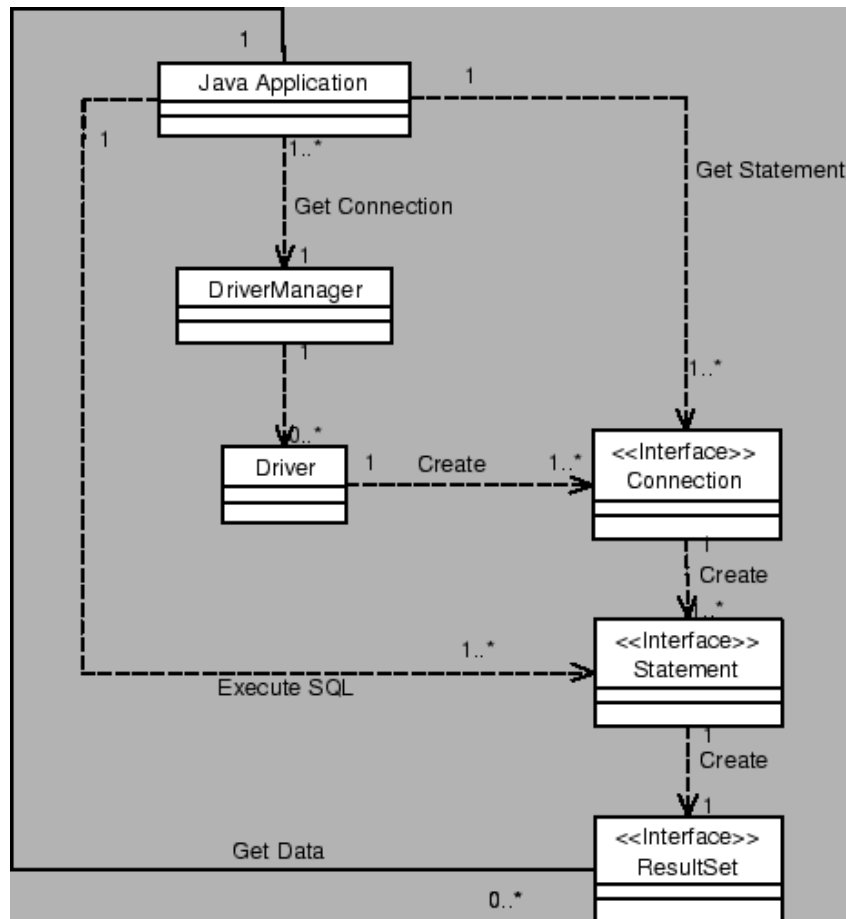
Es importante señalar que, aunque `javax.sql` se incluye en Java SE, no todos los métodos y clases en este paquete son necesariamente relevantes para todas las aplicaciones. Su utilidad dependerá de los requisitos específicos de la aplicación y del contexto en el que se esté trabajando.

Por tanto, aunque JDBC contiene un buen conjunto de clases, no es suficiente para conectarse y utilizar bases de datos relacionales. Para conectarse a una base de datos específica, JDBC requiere conectores o controladores adicionales.

Los fabricantes proporcionan APIs que permiten acceder a sus bases de datos. Un hecho que podría dar problemas a los programadores de Java si no existiera JDBC que es otra API que permite la ejecución de operaciones sobre bases de datos desde el lenguaje de programación Java, independientemente del sistema operativo donde se ejecute o de la base de datos a la cual se accede, utilizando el dialecto **SQL** (Structured Query Language) que es utilizado para gestionar y manipular bases de datos relacionales como MySQL, PostgreSQL, Oracle, SQL Server y muchas más.

Sin embargo, para que esto sea posible es necesario que el fabricante ofrezca un driver que cumpla la especificación JDBC.

Un **driver JDBC** es una capa de software intermedia que traduce las llamadas JDBC a las APIs específicas del vendedor.



Arquitectura general de una aplicación JDBC



ENLACE DE INTERÉS

En la web de Oracle puede encontrar documentación oficial sobre la API JDBC:





VÍDEO DE INTERÉS

Conoce más sobre el JDBC:



3. CONEXIÓN CON BASES DE DATOS RELACIONALES. CARACTERÍSTICAS, TIPOS Y MÉTODOS DE ACCESO

Siguiendo tu estrategia de adaptabilidad, debes analizar y conocer los tipos de drivers JDBC disponibles. Existen varios tipos, pero cada uno con sus propias características, ventajas y desventajas. La elección del driver adecuado es una decisión estratégica que impactará en la eficiencia de tu aplicación y su capacidad de adaptarse a las cambiantes necesidades de tu cliente.

La información contenida en un servidor de bases de datos es normalmente el bien máspreciado dentro de una empresa. La **API JDBC** ofrece a los desarrolladores Java un modo de conectar con dichas bases de datos. Utilizando la API JDBC, los desarrolladores pueden crear un cliente que pueda conectar con una base de datos, ejecutar instrucciones SQL y procesar el resultado de esas instrucciones.

La API proporciona **conectividad y acceso a datos** en toda la extensión de las bases de datos relacionales. JDBC **generaliza** las funciones de acceso a bases de datos más comunes abstrayendo los detalles específicos de una determinada base de datos. El resultado es un conjunto de clases e interfaces, localizadas en el paquete `java.sql`, que pueden ser utilizadas con cualquier base de datos que disponga del driver JDBC apropiado. La utilización de este driver significa que, siempre y cuando una aplicación utilice las características más comunes de acceso a bases de datos, dicha aplicación podrá utilizarse con una base de datos diferente cambiando simplemente a un driver JDBC diferente.

Los fabricantes de bases de datos más populares como Oracle, Microsoft, PostgreSQL... ofrecen APIs de su propiedad para el acceso del cliente.

Las aplicaciones cliente escritas en lenguajes nativos pueden utilizar estos APIs para obtener acceso directo a los datos, pero no ofrecen una interfaz común de acceso a diferentes bases de datos. La API JDBC ofrece una alternativa al uso de estas APIs, permitiendo acceder a diferentes servidores de bases de datos, únicamente cambiando el driver JDBC por el que ofrezca el fabricante del servidor al que se desea acceder.



ARTÍCULO DE INTERÉS

Existen otros estándares para la conexión con bases de datos como, por ejemplo, ODBC. Conoce más sobre ella en este enlace:



Tipos de Driver JDBC

Una de las decisiones importantes en el diseño, cuando se está proyectando una aplicación de bases de datos Java, es decidir el driver JDBC que permitirá que las clases JDBC se comuniquen con la base de datos. Los drivers JDBC se clasifican en cuatro tipos o niveles:

- Tipo 1: Puente JDBC-ODBC
- Tipo 2: Driver API nativo/parte Java
- Tipo 3: Driver protocolo de red/todo Java
- Tipo 4: Driver protocolo nativo/todo Java

Entender cómo se construyen los drivers y cuáles son sus limitaciones, nos ayudará a decidir qué driver es el más apropiado para cada aplicación.



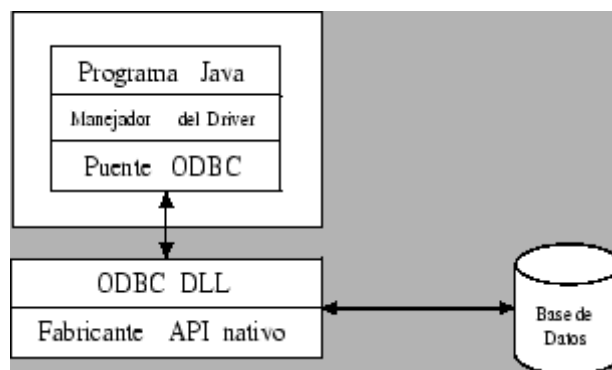
ENLACE DE INTERÉS

Aunque en esta unidad se habla de cuatro tipos de drivers JDBC, en algunas fuentes se introduce un tipo más. En este enlace puedes obtener información sobre este quinto tipo:



3.1 Tipo 1: Driver puente JDBC-ODBC

El puente JDBC-ODBC es un driver JDBC del tipo 1 que traduce operaciones JDBC en llamadas a la API ODBC. Estas llamadas son entonces cursadas a la base de datos mediante el driver ODBC apropiado. Esta arquitectura se muestra en esta figura:



Driver JDBC Tipo 1

El puente se implementa como el paquete `sun.jdbc.odbc` y contiene una biblioteca nativa utilizada para acceder a ODBC.



ENLACE DE INTERÉS

Conoce más información sobre conectividad JDBC-ODBC:



Ventajas

A menudo, el primer contacto con un driver JDBC es un puente JDBC-ODBC, simplemente porque es el driver que se distribuye como parte de Java, como el paquete `sun.jdbc.odbc.JdbcOdbcDriver`.

Además, tiene la ventaja de poder trabajar con una gran cantidad de drivers ODBC. Los desarrolladores suelen utilizar ODBC para conectar con bases de datos en un entorno distinto de Java. Por tanto, los drivers de tipo 1 pueden ser útiles para aquellas compañías que ya tienen un driver ODBC instalado en las máquinas clientes. Se utilizará normalmente en máquinas basadas en Windows que ejecutan aplicaciones de gestión. Por supuesto, puede ser el único modo de acceder a algunas bases de datos de escritorio, como MS Access, dBase y Paradox.

En este sentido, la ausencia de complejidad en la instalación y el hecho de que nos permita acceder virtualmente a cualquier base de datos, le convierte en una buena elección. Sin embargo, hay muchas razones por las que se desecha su utilización.

Desventajas

Básicamente sólo se recomienda su uso cuando se están realizando esfuerzos dirigidos a prototipos y en casos en los que no exista otro driver basado en JDBC para esa tecnología. Si es posible, se debe utilizar un driver JDBC en vez de un puente y un driver ODBC. Esto elimina totalmente la configuración cliente necesaria en ODBC.

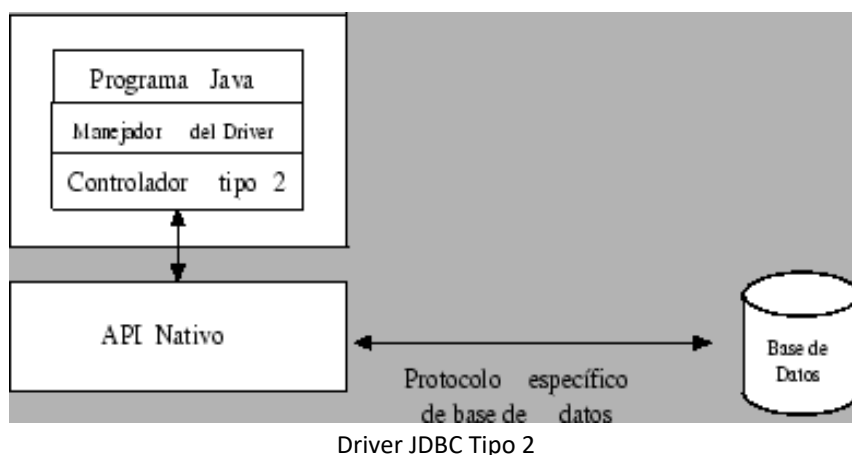
Los siguientes puntos resumen algunos de los inconvenientes de utilizar el driver puente:

- **Rendimiento.** Como se puede imaginar por el número de capas y traducciones que tienen lugar, utilizar el puente está lejos de ser la opción más eficaz en términos de rendimiento.

- Utilizando el puente JDBC-ODBC, el usuario está **limitado por la funcionalidad** del driver elegido. Es más, dicha funcionalidad se limita a proporcionar acceso a características comunes a todas las bases de datos, no pudiendo hacer uso de las mejoras que cada fabricante introduce en sus productos, especialmente en lo que afecta a rendimiento y escalabilidad.
- El driver puente **no funciona adecuadamente con applets**. El driver ODBC y la interfaz de conexión nativa deben estar ya instalados en la máquina cliente. Por eso, cualquier ventaja de la utilización de applets en un entorno de Intranet se pierde, debido a los problemas de despliegue que conllevan las aplicaciones tradicionales.
- La mayoría de los navegadores **no tienen soporte nativo del puente**. Como el puente es un componente opcional del Java SDK Standard Edition, no se ofrece con el navegador. Incluso si fuese ofrecido, sólo los applets de confianza (aquellos que permiten escribir en archivos) serán capaces de utilizar el puente. Esto es necesario para preservar la seguridad de los applet. Para terminar, incluso si el applet es de confianza, ODBC debe ser configurado en cada máquina cliente.

3.2 Tipo 2: Driver API Nativo / Parte JAVA

Los drivers de tipo 2, del que es un ejemplo el driver JDBC/OCI de Oracle, utilizan la interfaz de métodos nativos de Java para convertir las solicitudes de API JDBC en llamadas específicas a bases de datos para RDBMS como SQL Server, Informix, Oracle o PostgreSQL, como se puede ver en esta figura:



Aunque los drivers de tipo 2 habitualmente ofrecen mejor rendimiento que el puente JDBC-ODBC, siguen teniendo los mismos problemas de despliegue en los que la interfaz de conectividad nativa debe estar ya instalada en la máquina cliente. El driver JDBC necesita una biblioteca suministrada por el fabricante para traducir las funciones JDBC en lenguaje de consulta específico para ese servidor. Estos drivers están normalmente escritos en alguna combinación de Java y C/C++, ya que el driver debe utilizar una capa de C para realizar llamadas a la biblioteca que está escrita en C.



ENLACE DE INTERÉS

Amplía información sobre los cuatro tipos de drivers JDBC existentes en este enlace:



Ventajas

- El driver de tipo 2 ofrece un rendimiento significativamente mayor que el puente JDBC-ODBC, ya que las llamadas JDBC no se convierten en llamadas ODBC, sino que son directamente nativas.

Desventajas

- La biblioteca de la base de datos del fabricante necesita iniciarse en cada máquina cliente. En consecuencia, los drivers de tipo 2 no se pueden utilizar en Internet. Los drivers de tipo 2 muestran menor rendimiento que los de tipo 3 y 4.
- Un driver de tipo 2 también utiliza la interfaz nativa de Java, que no está implementada de forma consistente entre los distintos fabricantes de JVM por lo que habitualmente no es muy portable entre plataformas.



VÍDEO DE INTERÉS

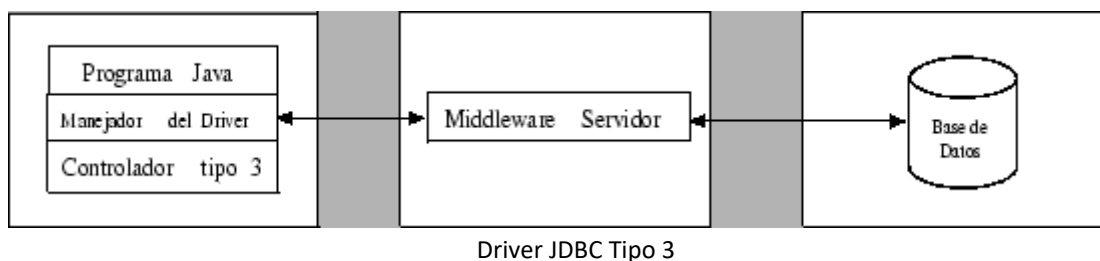
Con este vídeo tendrás una visión general del acceso a BBDD:



3.3 Tipo 3: Driver protocolo de red / Todo JAVA

Los drivers JDBC de tipo 3 están implementados en una aproximación de tres capas por lo que las solicitudes de la base de datos JDBC están traducidas en un protocolo de red independiente de la base de datos y dirigidas al servidor de capa intermedia. El servidor de la capa intermedia recibe las solicitudes y las envía a la base de datos utilizando para ello un driver JDBC del tipo 1 o del tipo 2 (lo que significa que se trata de una arquitectura muy flexible).

La arquitectura en conjunto consiste en tres capas: la capa cliente JDBC y driver, la capa intermedia y la base o las bases de datos a las que se accede.



El driver JDBC se ejecuta en el cliente e implementa la lógica necesaria para enviar a través de la red comandos SQL al servidor JDBC, recibir las respuestas y manejar la conexión.

El componente servidor intermedio puede implementarse como un componente nativo, o alternativamente escrito en Java. Las implementaciones nativas conectan con la base de datos utilizando bien una biblioteca cliente del fabricante o bien ODBC. El servidor tiene que configurarse para la base o bases de datos a las que se va a acceder.

Esto puede implicar asignación de números de puerto, configuración de variables de entorno, o de cualquier otro parámetro que pueda necesitar el servidor. Si el servidor intermedio está escrito en Java, puede utilizar cualquier driver en conformidad con JDBC para comunicarse con el servidor de bases de datos mediante el protocolo propietario del fabricante. El servidor JDBC maneja varias conexiones con la base de datos, así como excepciones y eventos de estado que resultan de la ejecución de SQL. Además, organiza los datos para su transmisión por la red a los clientes JDBC.

Ventajas

- El driver protocolo de red/todo Java tiene un componente en el servidor intermedio, por lo que no necesita ninguna biblioteca cliente del fabricante para presentarse en las máquinas clientes.
- Los drivers de tipo 3 son los que mejor funcionan en redes basadas en Internet o Intranet, aplicaciones intensivas de datos, en las que un gran número de operaciones concurrentes como consultas, búsquedas, etc., son previsibles y escalables y su rendimiento es su principal factor. Hay muchas oportunidades de optimizar la portabilidad, el rendimiento y la escalabilidad.
- El protocolo de red puede estar diseñado para hacer el driver JDBC cliente muy pequeño y rápido de iniciar, lo que es perfecto para el despliegue de aplicaciones de Internet.
- Además, un driver tipo 3, normalmente, ofrece soporte para características como almacenamiento en memoria caché (conexiones, resultados de consultas, etc.), equilibrio de carga, y administración avanzada de sistemas como el registro.
- La mayor parte de aplicaciones web de bases de datos basadas en 3 capas implican seguridad, firewalls y proxies y los drivers del tipo 3 ofrecen normalmente estas características.

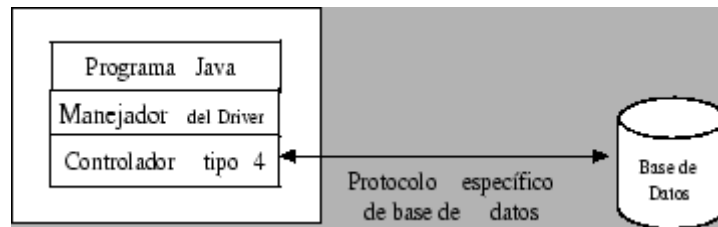
Inconvenientes

- Los drivers de tipo 3 requieren código específico de bases de datos para realizarse en la capa intermedia.
- Además, atravesar el conjunto de registros puede llevar mucho tiempo, ya que los datos vienen a través del servidor de datos.

3.4 Tipo 4: Driver protocolo Nativo / Todo JAVA

Este tipo de driver comunica directamente con el servidor de bases de datos utilizando el protocolo nativo del servidor. Estos drivers pueden escribirse totalmente en Java, son independientes de la plataforma y eliminan todos los aspectos relacionados con la configuración en el cliente. Sin embargo, este driver es específico de un fabricante determinado de base de datos. Cuando la base de datos necesita ser cambiada a un producto de otro fabricante, no se puede utilizar el mismo driver. Por el contrario, hay que reemplazarlo y también el programa cliente, o su asignación, para ser capaces de utilizar una cadena de conexión distinta para iniciar el driver.

Estos drivers traducen JDBC directamente a protocolo nativo sin utilizar ODBC o la API nativa, por lo que pueden proporcionar un alto rendimiento de acceso a bases de datos.



Driver JDBC Tipo 4

Ventajas

- Como los drivers JDBC de tipo 4 no tienen que traducir las solicitudes de ODBC o de una interfaz de conectividad nativa, o pasar la solicitud a otro servidor, el rendimiento es bastante bueno. Además, el driver protocolo nativo/todo Java da lugar a un mejor rendimiento que los de tipo 1 y 2.
- Además, no hay necesidad de instalar ningún software especial en el cliente o en el servidor. Además, estos drivers pueden bajarse de la forma habitual.

Desventajas

- Con los drivers de tipo 4, el usuario necesita un driver distinto para cada base de datos.



ARTÍCULO DE INTERÉS

Consulta esta información muy útil para conectar Java con MySQL en NetBeans:



ENLACE DE INTERÉS

Visualiza un ejemplo para conectar Java con MySQL en Eclipse:



PARA SABER MÁS

Conoce los 7 pasos a seguir para el manejo de MySQL con Java.



4. PREPARACIÓN DEL ENTORNO DE DESARROLLO

El equipo de desarrollo ya tiene claro el tipo de driver JDBC, además el cliente ha dejado a nuestra elección el sistema gestor de bases de datos, por lo que hemos escogido MySQL que es ampliamente utilizado en aplicaciones de todo tipo, por rápido, confiable, escalable y compatible con una amplia variedad de lenguajes de programación y plataformas. Por otro lado, usaremos un servidor local que permitirá al equipo crear y probar la aplicación en un entorno controlado antes de implementarla en un servidor de producción en línea.

Para poder conectar nuestro código Java con una base de datos de tipo MySQL necesitaremos instalar una serie de herramientas software en nuestro ordenador que nos ayudarán con esta tarea. Las principales serán las siguientes:

- Conector JDBC para conectar Java a MySQL.
- Servidor MySQL (lo incorpora XAMPP).
- Aplicación MySQL Workbench (incorpora servidor MySQL y conector JDBC).

Proceso para configurar nuestro equipo

A continuación, se muestra una forma sencilla de configurar un equipo con sistema operativo Windows.

1. Descargar driver JDBC

Hay varios sitios y formas de obtener el driver JDBC, en esta unidad te mostramos la siguiente manera.

En el sitio web oficial de MySQL de Oracle puedes encontrar los diferentes controladores basados en estándares para JDBC, ODBC y .Net, etc. Esta dirección (<https://www.mysql.com/products/connector/>) se utiliza cuando solo quieres descargar el controlador. En nuestro caso, queremos el controlador JDBC de MySQL en un archivo .zip (Punto rojo en imagen, hacer clic en Download), ya que es un componente separado del servidor de MySQL y se puede descargar y utilizar de forma independiente.

The screenshot shows the MySQL website's 'MySQL Connectors' page. The header includes the MySQL logo and navigation links. The sidebar on the left lists various MySQL products and documentation. The main content area is titled 'MySQL Connectors' and provides an overview of the drivers available. It lists drivers developed by MySQL and the community, including ADO.NET, ODBC, JDBC, Node.js, Python, and C++ drivers, each with a 'Download' link.

Conectores MySQL
Fuente: Elaboración propia

En la siguiente ventana, selecciona en “Operating Systems” la opción “Platform independent” y escoge el archivo .zip

Connector/J 8.1.0

Select Operating System:

Platform Independent

Platform Independent (Architecture Independent), Compressed TAR Archive

(mysql-connector-j-8.1.0.tar.gz)

8.1.0

4.0M

[Download](#)

MD5: 4a95d62b0cfbad68b92ffc62ae7ee266 | [Signature](#)

Platform Independent (Architecture Independent), ZIP Archive

(mysql-connector-j-8.1.0.zip)

8.1.0

4.8M

[Download](#)

MD5: d745362823ec4d37fa0607746d40a1b9 | [Signature](#)

Conectores MySQL-Selección plataforma independiente

Fuente: Elaboración propia

Clic en Download y lo descargamos. Y en la próxima ventana seleccionamos “No thanks, just start my download”.

MySQL Community Downloads

Login Now or Sign Up for a free account.

An Oracle Web Account provides you with the following advantages:

- Fast access to MySQL software downloads
- Download technical White Papers and Presentations
- Post messages in the MySQL Discussion Forums
- Report and track bugs in the MySQL bug system

Login »
using my Oracle Web account

Sign Up »
for an Oracle Web account

MySQL.com is using Oracle SSO for authentication. If you already have an Oracle Web account, click the Login link. Otherwise, you can signup for a free account by clicking the Sign Up link and following the instructions.

No thanks, just start my download. •

Conectores MySQL-Descargar realizada

Fuente: Elaboración propia

Descomprimos el archivo y veremos una carpeta con el nombre “mysql-connector-j-8.1.0” (la versión puede cambiar), al abrir la carpeta ya tenemos dentro nuestro conector, que es un archivo con extensión .jar

« Usuarios > Usuario > Downloads > mysql-connector-j-8.1.0 > mysql-connector-j-8.1.0				▼
Nombre	Fecha de modificación	Tipo	Tamaño	
src	26/06/2023 12:10	Carpeta de archivos		
build	26/06/2023 12:10	Archivo de origen ...	95 KB	
CHANGES	26/06/2023 12:10	Archivo	275 KB	
INFO_BIN	26/06/2023 12:10	Archivo	1 KB	
INFO_SRC	26/06/2023 12:10	Archivo	1 KB	
LICENSE	26/06/2023 12:10	Archivo	70 KB	
mysql-connector-j-8.1.0 •	26/06/2023 12:10	Executable Jar File	2.428 KB	
README	26/06/2023 12:10	Archivo	2 KB	

Conector mysql-connector-j-8.1.0

Fuente: Elaboración propia

Ahora abre Eclipse y crea un nuevo proyecto, dentro del mismo crea una carpeta o paquete, puedes llamarlo “ConectorJDBC” (o con otro nombre) y arrastra y suelta el archivo anterior dentro de este paquete. Ahora ya tienes el archivo en tu proyecto.

Pero como el conector JDBC es una biblioteca externa que no viene incluida por defecto en Eclipse, tienes que agregarlo como librería, para ello haz clic derecho sobre este archivo y selecciona “Build Path”, después “agregar build path”, de forma automática verás que en la carpeta “Referenced Libraries” se habrá incluido el archivo.

Una vez que hayas agregado el controlador JDBC a tu proyecto, puedes comenzar a utilizarlo para conectarte a la base de datos.

2. Sistema gestor de bases de datos MySQL

Ahora necesitas un gestor de bases de datos MySQL que utilizaremos para almacenar nuestra base de datos. Con este gestor también podremos, insertar, eliminar, modificar y realizar numerosas operaciones. Como desarrollador tendrás que acceder a esta base de datos desde código Java para realizar altas, bajas, modificaciones y eliminaciones de los datos que allí están almacenados. En la unidad vamos a utilizar el gestor de bases de datos MySQL que viene incluido en Xampp.

Xampp es un paquete software gratuito y de código abierto que proporciona un conjunto de herramientas para configurar y ejecutar un servidor web local completo para sistemas operativos Windows, Linux y macOS.

El nombre XAMPP es un acrónimo que representa los componentes principales del paquete e incluye phpMyAdmin:

- X: sistema operativo.
- Apache: servidor web HTTP, que responde a las solicitudes de los clientes web y envía los datos correspondientes al navegador del usuario.
- MySQL/MariaDB: sistema de gestión de bases de datos relacionales.
- PHP: lenguaje de programación.
- Perl: lenguaje de programación.

Se utiliza principalmente para desarrollar y probar aplicaciones web en un entorno local antes de implementarlas en un servidor remoto.



ENLACE DE INTERÉS

Puedes descargarte Xampp desde este enlace:



VÍDEO DE INTERÉS

Visualiza una explicación completa sobre como instalar y configurar XAMPP:



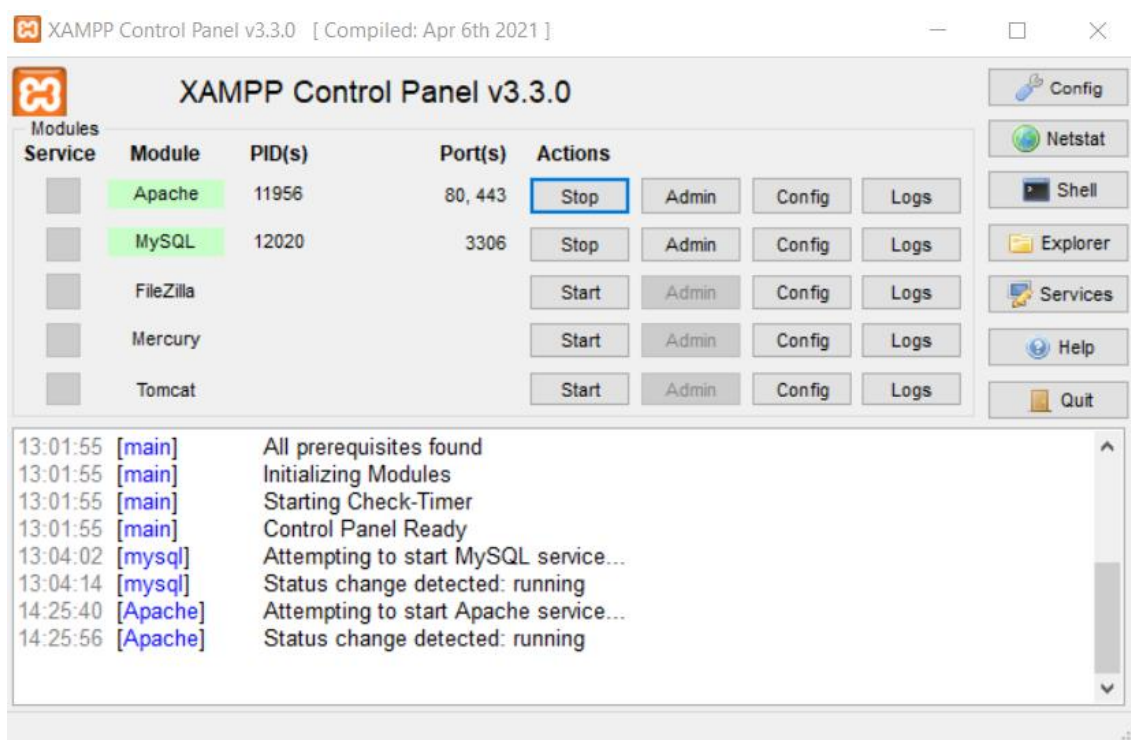
Xampp incluye una herramienta cliente web, con interfaz gráfica y de administración de bases de datos llamada **phpMyAdmin**.

Esta herramienta nos va a facilitar la tarea de consultar el estado de la base de datos, hacer modificaciones en la estructura de las tablas, o modificar los registros de una forma manual y con una interfaz más cómoda que desde la línea de comandos. (también nos servirá para comprobar que nuestro código realiza las inserciones, eliminaciones, etc., en los registros de las tablas de nuestra base de datos de forma correcta, entre otras cosas).

phpMyAdmin es aplicación web que en nuestro caso conectará con MySQL a través del servidor web Apache, que actúa de intermediario para transmitir las solicitudes entre el cliente y el servidor de bases de datos, por lo que tendremos que tener arrancados y ejecutándose ambos.

3. Iniciando servicios en XAMPP

Abrimos XAMPP y hacemos clic en “Start” de Apache y de MySQL.



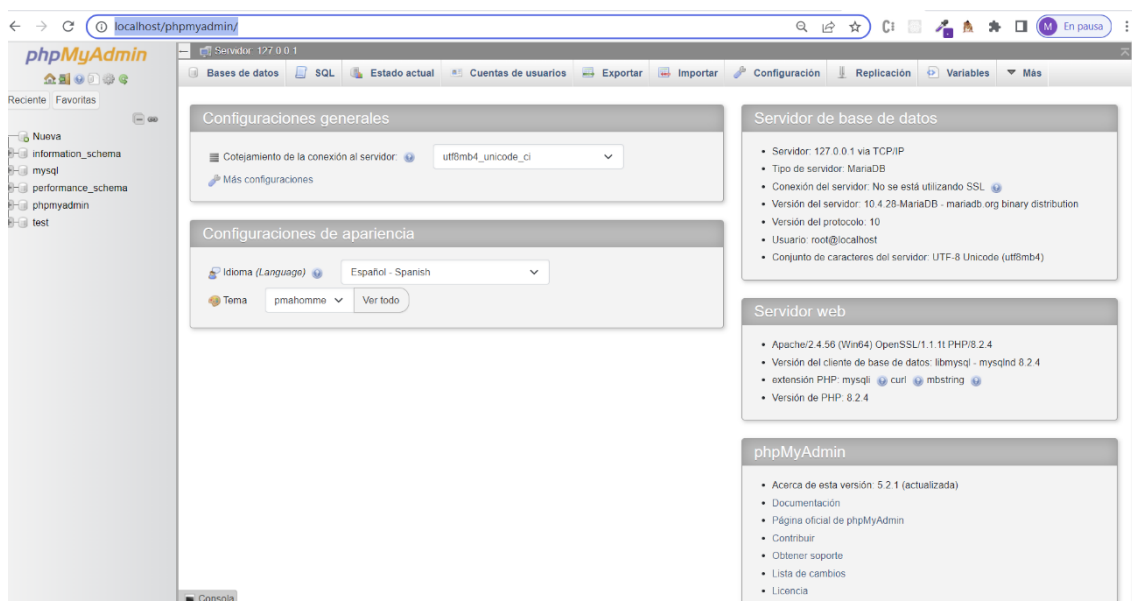
Instalación de Xampp

Fuente: Elaboración propia

En la imagen ambos servicios, tanto Apache como Xampp, están en funcionamiento. Los puertos por defecto en Apache son el 80 para HTTP y el 443 para HTTPS(SSL) y en MySQL el 3306 para las conexiones a la base de datos MySQL, por tanto, debemos comprobar que no estén siendo usados por otra aplicación, porque si es así, habría que modificarlos para que no haya conflictos.

Para acceder a phpMyAdmin hay varias opciones, puedes escoger hacer clic en el botón “Admin” que se encuentra en la misma fila que “MySQL” o también podemos abrir un navegador y escribimos lo siguiente: <http://localhost/phpmyadmin/>

Si todo ha sido correctamente instalado y configurado verás esta pantalla.



Ventana principal de phpMyAdmin

Fuente: Elaboración propia

En la parte izquierda de la ventana aparece un listado de las bases de datos que ya están creadas. Nuestro siguiente paso será crear una base de datos nueva en phpMyAdmin y las tablas de esa base de datos.



VÍDEO DE INTERÉS

Conoce más información sobre el programa cliente phpMyAdmin y cómo manejarlo:



5. ESTABLECIMIENTO DE CONEXIONES.

COMPONENTES DE ACCESO A DATOS

Estás muy emocionado porque por fin vas a realizar la conexión con la base de datos que has creado. El establecimiento de conexiones y la gestión de acceso a datos desempeñan un papel crítico para garantizar que la aplicación pueda interactuar de manera efectiva con la base de datos. Esto es especialmente importante porque la aplicación necesita recuperar información, almacenar registros, y realizar operaciones relacionadas con estos datos. Para este fin, Java proporciona una variedad de componentes de acceso a datos para interactuar con la base de datos.

La interfaz `java.sql.Connection` representa una conexión con una base de datos. Es una interfaz porque la implementación de una conexión depende de la red, del protocolo y del vendedor. El API JDBC ofrece dos vías diferentes para obtener conexiones. La primera utiliza la clase `java.sql.DriverManager` y es adecuada para acceder a bases de datos desde programas cliente escritos en Java. El segundo enfoque se basa en el acceso a bases de datos desde aplicaciones J2EE (Java 2 Enterprise Edition).

Se considera cómo se obtienen las conexiones utilizando la clase `java.sql.DriverManager`. En una aplicación, se pueden obtener una o más conexiones para una o más bases de datos utilizando drivers JDBC. Cada driver implementa la interfaz `java.sql.Driver`. Uno de los métodos que define esta interfaz es el método `connect()`, que permite establecer una conexión con la base de datos y obtener un objeto `Connection`.

En lugar de acceder directamente a clases que implementan la interfaz `java.sql.Driver`, el enfoque estándar para obtener conexiones es registrar cada driver con `java.sql.DriverManager` y utilizar los métodos proporcionados en esta clase para obtener conexiones. `java.sql.DriverManager` puede gestionar múltiples drivers. Antes de entrar en los detalles de este enfoque, es preciso entender cómo JDBC representa la URL de una base de datos.

5.1 Los URL de JDBC

La noción de una URL en JDBC es muy similar al modo típico de utilizar las URL. Las URL de JDBC proporcionan un modo de identificar un driver de base de datos. Un URL de JDBC representa un driver y la información adicional necesaria para localizar una base de datos y conectar a ella. Su sintaxis es la siguiente:

`jdbc:<subprotocol>:<subname>`

Existen tres partes separadas por dos puntos:

- **Protocolo:** En la sintaxis anterior, jdbc es el protocolo. Éste es el único protocolo permitido en JDBC.
- **Subprotocolo:** Utilizado para identificar el driver que utiliza la API JDBC para acceder al servidor de bases de datos. Este nombre depende de cada fabricante.
- **Subnombre:** La sintaxis del subnombre es específica del driver.

Por ejemplo, para una base de datos MySQL llamada “Bank”, el URL al que debe conectar es:

`jdbc:mysql:Bank`

Alternativamente, si se estuviera utilizando Oracle mediante el puente JDBC-ODBC, nuestro URL sería:

`jdbc:odbc:Bank`

Como puede ver, los URL de JDBC son lo suficientemente flexibles como para especificar información específica del driver en el subnombre.

5.2 Clase DriverManager

El propósito de la clase `java.sql.DriverManager` (gestor de drivers) es proporcionar una capa de acceso común encima de diferentes drivers de base de datos utilizados en una aplicación. En este enfoque, en lugar de utilizar clases de implementación Driver directamente, las aplicaciones utilizan la clase `DriverManager` para obtener conexiones. Esta clase ofrece tres métodos estáticos para obtener conexiones.

Sin embargo, `DriverManager` requiere que cada driver que necesite la aplicación sea registrado antes de su uso, de modo que el `DriverManager` sepa que está ahí.

El enfoque JDBC para el registro de un driver de base de datos puede parecer oscuro al principio. Fíjese en el siguiente fragmento de código que carga el driver de base de datos de MySQL:

```
try {  
    Class.forName("com.mysql.jdbc.Driver");  
} catch (ClassNotFoundException e) {  
    // Driver no encontrado  
}
```

En tiempo de ejecución, el `ClassLoader` localiza y carga la clase `com.mysql.jdbc.Driver` desde la ruta de clases utilizando el cargador de clase de autoarranque. Mientras carga una clase, el cargador de clase ejecuta cualquier código estático de inicialización para la clase.

En JDBC, se requiere que cada proveedor de driver registre una instancia del driver con la clase `java.sql.DriverManager` durante esta inicialización estática. Este registro tiene lugar automáticamente cuando el usuario carga la clase del driver (utilizando la llamada `Class.forName()`).

Es importante destacar que en versiones de Java más recientes (a partir de Java 6), **no siempre es necesario** utilizar `Class.forName(driver)` para cargar el controlador, ya que JDBC puede utilizar Service Provider Interfaces (SPI) para registrar automáticamente los controladores disponibles. Sin embargo, en algunas situaciones y para compatibilidad con versiones anteriores, todavía se utiliza `Class.forName(driver)` para cargar el controlador de manera explícita.

Una vez que el driver ha sido registrado con el `java.sql.DriverManager`, se puede utilizar sus métodos estáticos para obtener conexiones.



ENLACE DE INTERÉS

La documentación oficial sobre la clase `DriverManager` la encontrarás en este enlace:



El gestor de drivers tiene tres variantes del método estático `getConnection()` utilizado para establecer conexiones. El gestor de drivers delega estas llamadas en el método `connect()` de la interfaz `java.sql.Driver`.

Dependiendo del tipo de driver y del servidor de base de datos, una conexión puede conllevar una conexión de red física al servidor de base de datos o a un proxy de conexión. Las bases de datos integradas no requieren conexión física.

Exista o no una conexión física, el objeto de conexión es el único objeto que utiliza una conexión para comunicar con la base de datos. Toda comunicación debe tener lugar dentro del contexto de una o más conexiones.

Se consideran ahora los diferentes métodos para obtener una conexión:

- **java.sql.DriverManager** recupera el driver apropiado del conjunto de drivers registrados.
- **public static Connection getConnection(String url) throws SQLException** El URL de la base de datos está especificado en la forma de `jdbc:subprotocol:subname`. Para poder obtener una conexión a la base de datos es necesario que se introduzcan correctamente los parámetros de autenticación requeridos por el servidor de bases de datos.
- **public static Connection getConnection(String url, java.util.Properties info) throws SQLException**
Este método requiere una URL y un objeto `java.util.Properties`. El objeto `Properties` contiene cada parámetro requerido para la base de datos especificada. La lista de propiedades difiere entre bases de datos.

Dos propiedades comúnmente utilizadas para una base de datos son `autocommit=true` y `create=false`. Se pueden especificar estas propiedades junto con la URL como `jdbc:subprotocol:subname; autocommit=true;create=true` o se pueden establecer estas propiedades utilizando el objeto `Properties` y pasar dicho objeto como parámetro en el anterior método `getConnection()`.

```
String url = "jdbc:mysql:Bank";
Properties p = new Properties();
p.put("autocommit", "true");
p.put("create", "true");
Connection connection = DriverManager.getConnection(url, p);
```

En caso de que no se adjunten todas las propiedades requeridas para el acceso, se generará una excepción en tiempo de ejecución.

La tercera variante toma como argumentos además del URL, el nombre del usuario y la contraseña.

En este ejemplo, utiliza un driver MySQL, y requiere un nombre de usuario y una contraseña para obtener una conexión:

```
String url = "jdbc:mysql:Bank";  
String user = "root";  
String password = "root";  
Connection connection = DriverManager.getConnection(url,  
                                                    user, password);
```

Observe que todos estos métodos están sincronizados, lo que supone que sólo puede haber un hilo accediendo a los mismos en cada momento. Estos métodos lanzan una excepción `SQLException` si el driver no consigue obtener una conexión.

Interfaz Driver

Cada driver debe implementar la interfaz `java.sql.Driver`. En MySQL, la clase `com.mysql.jdbc.Driver` implementa la interfaz `java.sql.Driver`.

La clase `DriverManager` utiliza los métodos definidos en esta interfaz. En general, las aplicaciones cliente no necesitan acceder directamente a la clase `Driver` puesto que se accederá a la misma a través de la API JDBC. Esta API enviará las peticiones al `Driver`, que será, quién en último término, acceda a la base de datos.

6. RECUPERACIÓN DE INFORMACIÓN. SELECCIÓN DE REGISTROS. USO DE PARÁMETROS

La conexión con la base de datos ha sido un éxito, no has tenido ningún problema y tienes vía libre para comenzar a codificar los métodos de acceso a la misma. Utilizarás un ResultSet que es una estructura que contiene los resultados de una consulta, permitiendo a la aplicación acceder a los datos de la base de datos de manera estructurada y así realizar operaciones como la lectura, navegación y manipulación de los registros devueltos por la consulta.

El objeto Statement devuelve un objeto `java.sql.ResultSet` que encapsula los resultados de la ejecución de una sentencia SELECT. Esta interfaz es implementada por los vendedores de drivers. Dispone de métodos que permiten al usuario navegar por los diferentes registros que se obtienen como resultado de la consulta.

El siguiente método, `executeQuery`, definido en la interfaz `java.sql.Statement` le permite ejecutar las instrucciones SELECT:

```
public ResultSet executeQuery (String sql) throws SQLException
```

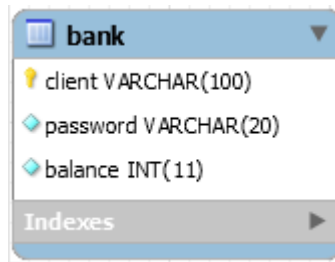
La interfaz `java.sql.ResultSet` ofrece varios métodos para recuperar los datos que se obtienen de realizar una consulta:

- `getBoolean()`
- `getInt()`
- `getShort()`
- `getByte()`
- `getDate()`
- `getDouble()`
- `getfloat()`

Todos estos métodos requieren el nombre de la columna (tipo `String`) o el índice de la columna (tipo `int`) como argumento. La sintaxis para las dos variantes del método `getString()` es la siguiente:

```
public String getString(int columnIndex) throws SQLException  
public String getString(String columnName) throws SQLException
```

Para los ejemplos que se van a ver a partir de este punto, se va a usar la tabla bank. La estructura de la tabla se puede ver en esta imagen:



En la clase CreateTableBank, se crea un nuevo método queryAll() que recupere todos los datos de la tabla bank.



EJEMPLO PRÁCTICO

Método QueryAll que recupera todos los datos de la tabla bank:

```
public void queryAll() throws SQLException {  
    String sqlString =  
        "SELECT client, password, balance" +  
        "FROM bank";  
    Statement statement = connection.createStatement();  
    ResultSet rs = statement.executeQuery(sqlString);  
    while (rs.next()) {  
        System.out.println(rs.getString("client") +  
            rs.getString("password") +  
            rs.getInt("balance"));  
    }  
}
```

Este método crea un objeto Statement, utilizado para invocar el método executeQuery() con una instrucción SQL (SELECT) como argumento.

El objeto java.sql.ResultSet contiene todas las filas de la tabla bank que coinciden con la instrucción SELECT. Utilizando el método next() del objeto ResultSet, se pueden recorrer todas las filas contenidas en el bloque de resultados. En cualquier fila, se pueden utilizar uno de los métodos getXxx() descritos anteriormente para recuperar los campos de una fila.

La interfaz `ResultSet` también permite conocer la estructura del bloque de resultados. El método `getMetaData()` ayuda a recuperar un objeto `java.sql.ResultSetMetaData` que tiene varios métodos para describir el bloque de resultados, algunos de los cuales se enumeran a continuación:

- `getTableName()`
- `getColumnCount()`
- `getColumnName()`
- `getColumnType()`

Tomando un bloque de resultados, se puede utilizar el método `getColumnCount()` para obtener el número de columnas de dicho bloque. Conocido el número de columnas, se puede obtener la información de tipo asociada a cada una de ellas.



EJEMPLO PRÁCTICO

Por ejemplo, este método imprime la estructura del bloque de resultados:

```
public void getMetaData() throws SQLException {
    String sqlString = "SELECT * FROM bank";
    Statement statement = connection.createStatement();
    ResultSet rs = statement.executeQuery(sqlString);
    ResultSetMetaData metaData = rs.getMetaData();
    int noColumns = metaData.getColumnCount();
    for (int i=1; i<noColumns+1; i++) {
        System.out.println(metaData.getColumnName(i)
                           + " " +
                           metaData.getColumnType(i));
    }
}
```

El método anterior obtiene el número de columnas del bloque de resultados e imprime el nombre y el tipo de cada columna. En este caso, los nombres de columna son `client`, `password` y `balance`. Observe que los tipos de columna son devueltos como números enteros. Por ejemplo, todas las columnas de tipo `VARCHAR` retornarán el entero 12, las del tipo `DATE`, 91. Estos tipos son constantes definidas en la interfaz `java.sql.Types`. Fíjese también en que los números de columnas empiezan desde 1 y no desde 0.

7. MANIPULACIÓN DE LA INFORMACIÓN. ALTAS, BAJAS Y MODIFICACIONES. EJECUCIÓN DE CONSULTAS SOBRE LA BASE DE DATOS

Para poder gestionar y procesar la lógica de los datos almacenados en tu base de datos está siendo fundamental dominar el lenguaje SQL (Structured Query Language). Este lenguaje te está permitiendo realizar consultas, inserciones, actualizaciones, eliminaciones y definiciones de datos, incluyendo la creación de tablas, la especificación de claves primarias y foráneas, etc. Al principio te ha costado un poco comprender y asimilar la sintaxis y el funcionamiento, pero al final has descubierto que es uno de tus lenguajes preferidos.

Antes de poder ejecutar una sentencia SQL, es necesario obtener un objeto de tipo Statement. Una vez creado dicho objeto, podrá ser utilizado para ejecutar cualquier operación contra la base de datos.

El siguiente método crea un objeto Statement, que se puede utilizar para enviar instrucciones SQL a la base de datos.

Statement createStatement() throws SQLException

La finalidad de un objeto Statement es ejecutar una instrucción SQL que puede o no devolver resultados. Para ello, la interfaz Statement dispone de los siguientes métodos:

- `executeQuery()`, para sentencias SQL que recuperen datos de un único objeto `ResultSet`.
- `executeUpdate()`, para realizar actualizaciones que no devuelvan un `ResultSet`. Por ejemplo, sentencias DML SQL (Data Manipulation Language) como `INSERT`, `UPDATE` y `DELETE`, o sentencias DDL SQL (Data Definition Language) como `CREATE TABLE`, `DROP TABLE` y `ALTER TABLE`. El valor que devuelve `executeUpdate()` es un entero (conocido como la cantidad de actualizaciones) que indica el número de filas que se vieron afectadas. Las sentencias que no operan en filas, como `CREATE TABLE` o `DROP TABLE`, devuelven el valor cero.



ENLACE DE INTERÉS

Conoce más sobre la interfaz Statement:



Crear la tabla bank

Para ilustrar el API JDBC, se considerará la clase CreateTableBank. Esta clase ofrece los métodos `initialize()` y `close()` para establecer y liberar una conexión con la base de datos.

El método `createTableBank` crea la tabla `bank`, utilizando para ello un objeto de tipo `Statement`. Sin embargo, dado que el método `executeUpdate()` ejecuta una sentencia SQL de tipo `CREATE TABLE`, ésta no actualiza ningún registro de la base de datos y por ello este método devuelve cero. En caso de ejecutar una sentencia de tipo `INSERT`, `UPDATE` o `DELETE`, el método devolvería el número de filas que resultasen afectadas por el cambio.

```
public class CreateTableBank {
    String driver      = "com.mysql.jdbc.Driver";
    String url         = "jdbc:mysql:Bank";
    String login       = "root";
    String password    = "root";
    String createTableBank = "CREATE TABLE bank (" +
                            "client VARCHAR(100) NOT NULL, " +
                            "password VARCHAR(20) NOT NULL, " +
                            "balance Integer NOT NULL, " +
                            "PRIMARY KEY(client))";

    Connection connection = null;
    Statement statement    = null;
    public void initialize() throws
        SQLException, ClassNotFoundException {
        Class.forName(driver);
        connection = DriverManager.getConnection(url,
                                                login, password);
    }
    public void createTableBank() throws SQLException {
        statement = connection.createStatement();
        statement.executeUpdate(createTableBank);
    }
}
```

```
        public void close() throws SQLException {
            try {
                connection.close();
            } catch (SQLException e) {
                throw e;
            }
        }
    }
}
```

Una vez creada la tabla bank, el siguiente paso podría ser la introducción en la misma de los datos de los clientes. Si dichos datos están disponibles en un fichero, el código para leerlos e introducirlos en la base de datos sería el siguiente:

```
public void insertData() throws SQLException, IOException {
    String client, password;
    int balance;
    BufferedReader br = new BufferedReader(
        new FileReader("clients.txt"));
    try {
        do {
            client = br.readLine();
            password = br.readLine();
            balance = Integer.parseInt(br.readLine());
            String sqlString =
                "INSERT INTO bank VALUES('"
                    + client + "','" + password + "','"
                    + balance + "')";
            statement.executeUpdate(sqlString);
            statement.close();
        } while (br.readLine() != null);
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        br.close();
    }
}
```

El formato del archivo de entrada es: nombre del cliente, clave de acceso y saldo, introducidos en líneas separadas, y seguidos de una línea separatoria como se muestra a continuación:

```
Iván Samuel Tejera Santana
root
1000000
-----
```

En el código anterior, la única sentencia relevante es el método `statement.executeUpdate()`, invocado para insertar datos en la tabla bank (dicho método devuelve el número de registros insertados).

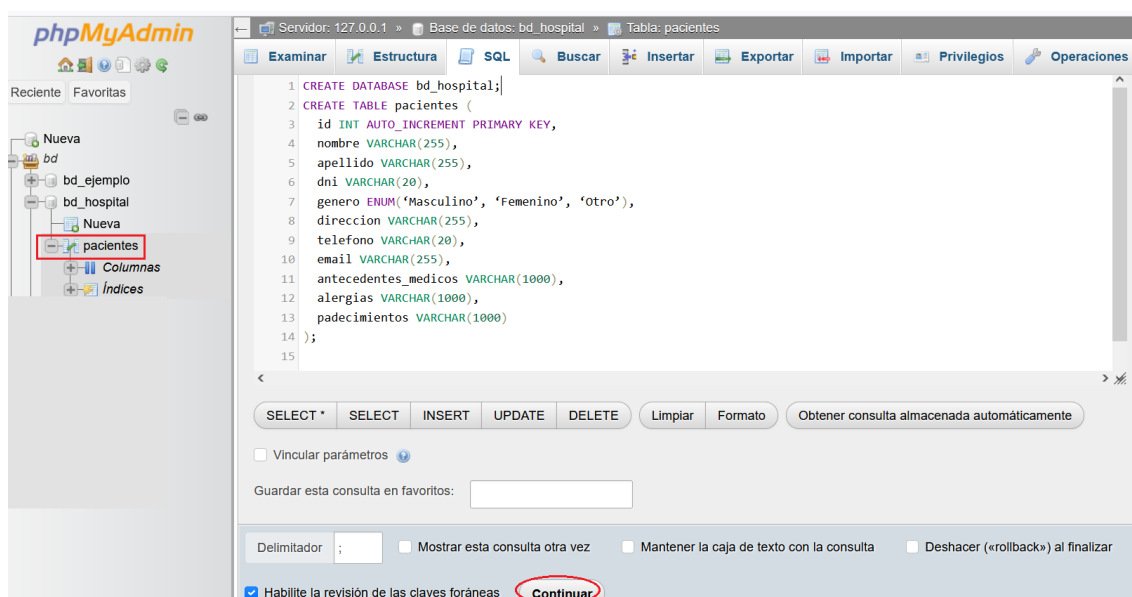
En este ejemplo se mostrará como realizar una pequeña aplicación Java de tipo CRUD (Create, Read, Update, Delete) con la información que tendremos almacenada en una base de datos llamada `bd_hospital` que estará compuesta por una sola tabla que se llama `pacientes`.

Comenzaremos creando desde phpMyAdmin, la base de datos, la tabla y la definición de sus campos ejecutando las sentencias SQL adecuadas, que colocaremos según aparece en la imagen y haciendo clic en el botón Continuar. (En un ejemplo anterior esto lo hicimos, pero desde la propia aplicación Java).

Si todo ha ido bien se debe de ver en la parte izquierda, la base de datos y la tabla creada, si no la ves, prueba a refrescar la interfaz haciendo clic de nuevo en la pestaña “Examinar”.

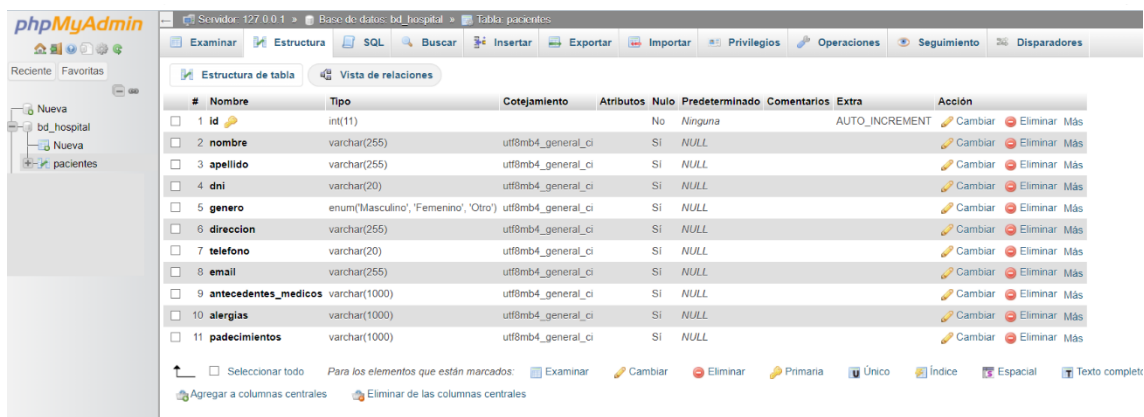
El siguiente paso sería rellenar la tabla con datos para que podamos realizar lecturas, modificaciones y eliminaciones desde la aplicación Java.

Por último, tenemos que implementar el código necesario en nuestra aplicación Java para poder acceder a los datos y realizar operaciones con ellos.



Sentencias SQL-tabla pacientes

Fuente: Elaboración propia

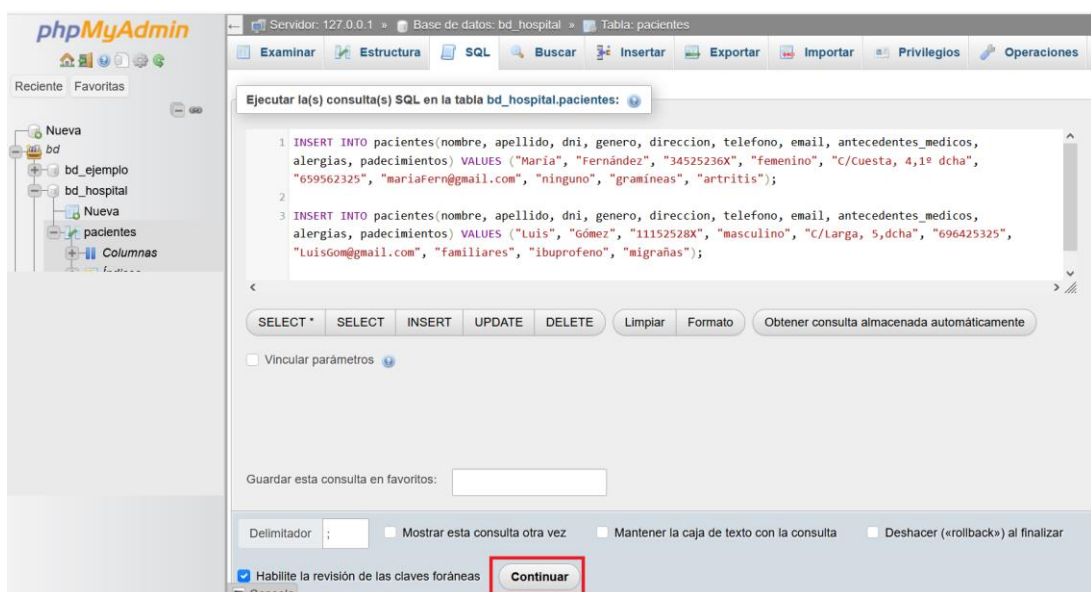


phpMyAdmin - Servidor: 127.0.0.1 - Base de datos: bd_hospital - Tabla: pacientes

#	Nombre	Tipo	Cotejamiento	Atributos	Nulo	Predeterminado	Comentarios	Extra	Acción
1	id	int(11)			No	Ninguna		AUTO_INCREMENT	Cambiar Eliminar Más
2	nombre	varchar(255)	utf8mb4_general_ci		Si	NULL			Cambiar Eliminar Más
3	apellido	varchar(255)	utf8mb4_general_ci		Si	NULL			Cambiar Eliminar Más
4	dni	varchar(20)	utf8mb4_general_ci		Si	NULL			Cambiar Eliminar Más
5	genero	enum('Masculino', 'Femenino', 'Otro')	utf8mb4_general_ci		Si	NULL			Cambiar Eliminar Más
6	direccion	varchar(255)	utf8mb4_general_ci		Si	NULL			Cambiar Eliminar Más
7	telefono	varchar(20)	utf8mb4_general_ci		Si	NULL			Cambiar Eliminar Más
8	email	varchar(255)	utf8mb4_general_ci		Si	NULL			Cambiar Eliminar Más
9	antecedentes_medicos	varchar(1000)	utf8mb4_general_ci		Si	NULL			Cambiar Eliminar Más
10	alergias	varchar(1000)	utf8mb4_general_ci		Si	NULL			Cambiar Eliminar Más
11	padecimientos	varchar(1000)	utf8mb4_general_ci		Si	NULL			Cambiar Eliminar Más

Estructura tabla pacientes

Fuente: Elaboración propia



phpMyAdmin - Servidor: 127.0.0.1 - Base de datos: bd_hospital - Tabla: pacientes

Ejecutar la(s) consulta(s) SQL en la tabla bd_hospital.pacientes:

```

1 INSERT INTO pacientes(nombre, apellido, dni, genero, direccion, telefono, email, antecedentes_medicos,
alergias, padecimientos) VALUES ("María", "Fernández", "34525236X", "femenino", "C/Cuesta, 4,1ª dcha",
"659562325", "mariafern@gmail.com", "ninguno", "gramíneas", "artritis");
2
3 INSERT INTO pacientes(nombre, apellido, dni, genero, direccion, telefono, email, antecedentes_medicos,
alergias, padecimientos) VALUES ("Luis", "Gómez", "11152528X", "masculino", "C/Larga, 5,dcha", "696425325",
"LuisGom@gmail.com", "familiares", "ibuprofeno", "migrañas");

```

SELECT * SELECT INSERT UPDATE DELETE Limpiar Formato Obtener consulta almacenada automáticamente

☐ Vincular parámetros

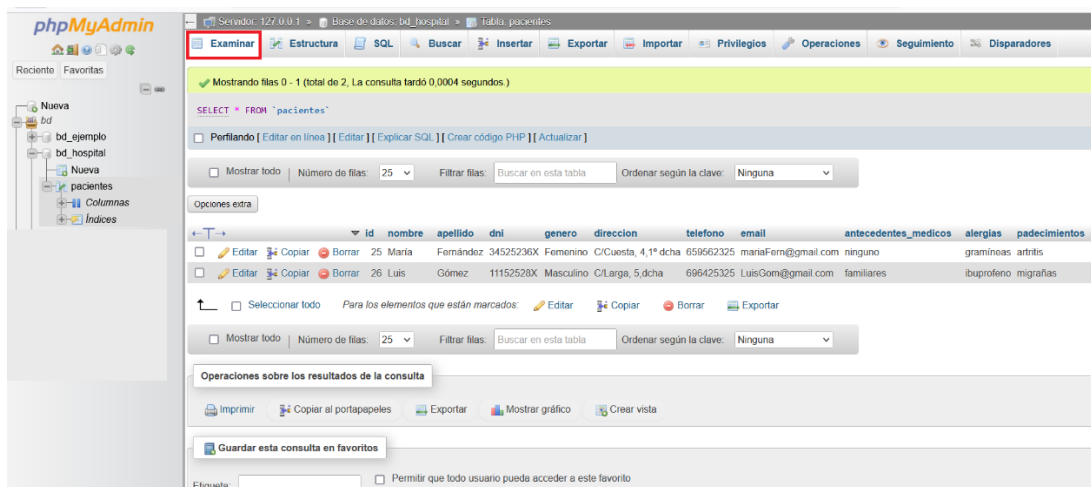
Guardar esta consulta en favoritos:

Delimitador: ; ☐ Mostrar esta consulta otra vez ☐ Mantener la caja de texto con la consulta ☐ Deshacer («rollback») al finalizar

☒ Habilita la revisión de las claves foráneas **Continuar**

Insertar datos en tabla pacientes

Fuente: Elaboración propia



phpMyAdmin - Servidor: 127.0.0.1 - Base de datos: bd_hospital - Tabla: pacientes

Examinar Estructura SQL Buscar Insertar Exportar Importar Privilegios Operaciones Seguimiento Disparadores

Mostrando filas 0 - 1 (total de 2, La consulta tardó 0,0004 segundos)

SELECT * FROM "pacientes"

☐ Perfilando [[Editar en línea](#)] [[Explicar SQL](#)] [[Crear código PHP](#)] [[Actualizar](#)]

☐ Mostrar todo | Número de filas: 25 | Filtrar filas: Buscar en esta tabla | Ordenar según la clave: Ninguna

Opciones extra

	id	nombre	apellido	dni	genero	direccion	telefono	email	antecedentes_medicos	alergias	padecimientos
<input type="checkbox"/> Editar Copiar Borrar	25	María	Fernández	34525236X	Femenino	C/Cuesta, 4,1ª dcha	659562325	mariafern@gmail.com	ninguno	gramíneas	artritis
<input type="checkbox"/> Editar Copiar Borrar	26	Luis	Gómez	11152528X	Masculino	C/Larga, 5,dcha	696425325	LuisGom@gmail.com	familiares	ibuprofeno	migrañas

☐ Seleccionar todo Para los elementos que están marcados: [Editar](#) [Copiar](#) [Borrar](#) [Exportar](#)

☐ Mostrar todo | Número de filas: 25 | Filtrar filas: Buscar en esta tabla | Ordenar según la clave: Ninguna

Operaciones sobre los resultados de la consulta

[Imprimir](#) [Copiar al portapepeles](#) [Exportar](#) [Mostrar gráfico](#) [Crear vista](#)

[Guardar esta consulta en favoritos](#)

Etiqueta: ☐ Permitir que todo usuario pueda acceder a este favorito

Registros insertados en tabla pacientes

Fuente: Elaboración propia



PARA SABER MÁS

Si quieres volver a repetir el script de creación de tabla SQL, porque te has equivocado en la definición de los datos, etc. tendrás que eliminar la instrucción de creación de la base de datos (porque ya existe y no se puede volver a crear con el mismo nombre) e incluir la instrucción siguiente al comienzo del script anterior, de esta forma se elimina la tabla y a continuación se volverá a crear.

```
DROP TABLE pacientes;  
CREATE TABLE pacientes (  
    ...  
    ...  
);
```

Si lo que quieres es borrar la base de datos y todas sus tablas, utiliza la siguiente:

```
DROP DATABASE bd_hospital;  
CREATE DATABASE bd_hospital;  
CREATE TABLE pacientes (  
    ...  
    ...  
);
```

Como en la base de datos se han estado realizando pruebas, el id se ha ido incrementado cada vez y no se inicializará, aunque borremos todos los registros de la Base de datos. Si queremos que el id vuelva a inicializarse a “1” podemos borrar la tabla y crearla de nuevo como se ha indicado anteriormente. Para probar el funcionamiento del código que estamos desarrollando, se ha vuelto a borrar la tabla paciente para que la numeración del id del paciente comience en “1”.

El usuario de nuestra base de datos será “root” y la contraseña la dejaremos en blanco. Utilizaremos consultas preparadas de tipo preparedStatement.

Empezamos creando la estructura siguiente en nuestra aplicación:

- **Clase ConexionBD:** Contendrá los atributos de conexión y un método getConnection().
- **Clase Paciente:** Con 10 atributos, los mismos que en la tabla paciente pero aquí no vamos a incluir el atributo id, ya que este es un dato que autoincrementa la base de datos por tanto no lo gestionamos desde Java, aunque sí lo usaremos para buscar o eliminar algún paciente por su id.

- **Clase PacienteDAO.java:** Un atributo de tipo ConexionBD (ConexionBD es una clase que ya hemos creado). En el constructor de esta clase PacienteDAO, se crea la conexión a la base de datos bd_hospital, desde el puerto 3306, con el usuario root y la contraseña vacía.

Esta clase contendrá cada uno de los métodos que nos servirán para gestionar la base de datos. Usamos la notación **DAO** que significa (Data Access Object). Es un patrón de diseño de software que se utiliza para separar la lógica de negocio de la lógica de acceso a datos en una aplicación. Encapsula el acceso a los datos en una serie de métodos que la capa de aplicación puede llamar para realizar operaciones en la base de datos, como guardar, actualizar, recuperar o eliminar datos.

- **Clase Main:** Contiene un método main(), desde el que instanciaremos objetos de tipo Paciente y pasaremos datos en sus constructores. Una vez tengamos un objeto paciente instanciado y con información, lo insertaremos en la base de datos usando el método correspondiente definido en la clase PacienteDAO. También se harán actualizaciones de un paciente, eliminaciones y búsquedas.

Clase ConexionBD.java

```
package EjercicioBDPacientes;

import java.sql.Connection;

import java.sql.DriverManager;
import java.sql.SQLException;

public class ConexionBD {
    /*atributos de la clase ConexionBD, que guardarán datos
    de la
        conexión a la base de datos*/
    private String jdbcUrl;
    private String user ;
    private String password;

    /*constructor de 3 parámetros que recibe los datos de
    conexión y los almacena en sus correspondientes
    atributos*/
    public ConexionBD(String jdbcUrl, String user, String
        password) {
        this.jdbcUrl = jdbcUrl;
        this.user = user;
        this.password = password;
    }
    //método getter que retorna un objeto de tipo conexión
    public Connection getConnection() throws SQLException {
        /*Establecemos la conexión. La clase DriverManager
        proporciona métodos para administrar las conexiones
```

```
        a la base de datos*/
        Connection connection =
            DriverManager.getConnection(this.jdbcUrl,
this.user,
this.password);

        /*Se retorna el objeto que representa nuestra
conexión a la base de datos*/
        return connection;
    }
}
```

Clase Paciente.java

```
package EjercicioBDPacientes;

public class Paciente {
    private String nombre;
    private String apellido;
    private String dni;
    private String genero;
    private String direccion;
    private String telefono;
    private String email;
    private String antecedentes_medicos;
    private String alergias;
    private String padecimientos;

    // Constructor
    public Paciente() {

    }

    public Paciente(String nombre, String apellido, String dni,
String genero, String direccion, String telefono,String
email,String antecedentes_medicos, String alergias, String
padecimientos) {
        this.nombre = nombre;
        this.apellido = apellido;
        this.dni = dni;
        this.genero = genero;
        this.direccion = direccion;
        this.telefono = telefono;
        this.email = email;
        this.antecedentes_medicos = antecedentes_medicos;
        this.alergias = alergias;
        this.padecimientos = padecimientos;
    }

    public String getNombre() {
        return nombre;
    }

    public String getApellido() {
        return apellido;
    }
}
```

```
public String getDni() {
    return dni;
}

public String getGenero() {
    return genero;
}

public String getDireccion() {
    return direccion;
}

public String getTelefono() {
    return telefono;
}

public String getEmail() {
    return email;
}

public String getAntecedentesMedicos() {
    return antecedentes_medicos;
}

public String getAlergias() {
    return alergias;
}

public String getPadecimientos() {
    return padecimientos;
}

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

public void setApellido(String apellido) {
    this.apellido = apellido;
}

public void setDni(String dni) {
    this.dni = dni;
}

public void setGenero(String genero) {
    this.genero = genero;
}

public void setDireccion(String direccion) {
    this.direccion = direccion;
}

public void setTelefono(String telefono) {
    this.telefono = telefono;
}

public void setEmail(String email) {
    this.email = email;
}
```



```
}

public void setAntecedentesMedicos(String antecedentes_medicos) {
    this.antecedentes_medicos = antecedentes_medicos;
}

public void setAlergias(String alergias) {
    this.alergias = alergias;
}

public void setPadecimientos(String padecimientos) {
    this.padecimientos = padecimientos;
}

@Override
public String toString() {
    return "Nombre: " + nombre + "\n" + "Apellido: " +
        apellido + "\n" + "DNI: " + dni + "\n" + "Genero: " +
        genero + "\n" + "Direccion: " + direccion +
        "\n" + "Telefono: " + telefono + "\n" + "Email: " +
        email + "\n" + "Antecedentes_medicos: " +
        antecedentes_medicos + "\n" + "Alergias: " +
        alergias + "\n" + "Padecimientos: " +
        padecimientos + "\n";
}
}
```

Ya que tenemos los getters de cada atributo se pueden usar en este método toString () para mostrar el contenido, sin embargo, se ha optado por poner solo los atributos.

Clase PacienteDAO.java

```
package EjercicioBDPacientes;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;

public class PacienteDAO {

    private ConexionBD conexionBD;

    // constructor sin parámetros
    public PacienteDAO() {
        this.conexionBD = new
            ConexionBD("jdbc:mysql://localhost:3306/bd_hospital", "root", "");
    }

    /*
```

```

    * En caso de que exista una conexión abierta,
    * se llamará al método close() del objeto para
    * cerrar la conexión. La excepción SQLException
    * puede ser lanzada por el método close()
    */

    private void close(Connection connection) {
        if (connection != null) {
            try {
                connection.close();
            } catch (SQLException e) {
                /* No se hace nada en caso de error
            */
            }
        }
    }

    /*
    * Si el objeto Statement no es null, es decir,
    * si existe un objeto para ejecutar consultas,
    * se llama al método close() del objeto para
    * liberar los recursos utilizados por la consulta
    */

    private void close(PreparedStatement preparedStatement) {
        if (preparedStatement != null) {
            try {
                preparedStatement.close();
            } catch (SQLException e) {
                /* No se hace nada en caso de error
            */
            }
        }
    }

    public void insertarPaciente(Paciente paciente) throws
        SQLException {
        /* este método recibe un objeto de la clase Paciente
        que usaremos para acceder a los getters de cada atributo de
        dicho objeto*/

        Connection connection = null;
        PreparedStatement pstmt = null;

        try {
            connection = ConexionBD.getConnection();
            String query = "INSERT INTO pacientes (nombre,
                apellido, dni, genero, direccion,
                telefono, email, antecedentes_medicos, alergias,
                padecimientos) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?,
                ?) ";

            pstmt = connection.prepareStatement(query);
            pstmt.setString(1, paciente.getNombre());
            pstmt.setString(2, paciente.getApellido());
            pstmt.setString(3, paciente.getDni());

```

```
        pstmt.setString(4, paciente.getGenero());
        pstmt.setString(5, paciente.getDireccion());
        pstmt.setString(6, paciente.getTelefono());
        pstmt.setString(7, paciente.getEmail());
        pstmt.setString(8,
paciente.getAntecedentesMedicos());
        pstmt.setString(9, paciente.getAlergias());
        pstmt.setString(10, paciente.getPadecimientos());

        int rowsInserted = pstmt.executeUpdate();
        if (rowsInserted > 0) {
            System.out.println("Se ha insertado el paciente

                                correctamente");
        }
    } finally {
        close(pstmt);
        close(connection);
    }
}
```

```
public void obtenerPacientePorID(int id) throws SQLException {
    /* este método recibe el id del paciente a buscar*/
    Connection connection = null;
    PreparedStatement pstmt = null;

    try {
        connection = conexionBD.getConnection();
        String query = "SELECT * FROM pacientes
                        WHERE id = ?";
        pstmt = connection.prepareStatement(query);
        pstmt.setInt(1, id);

        /*Ejecutar sentencia SQL y cargar el ResultSet
        con los datos del registro encontrado en la
        tabla*/
        ResultSet rs = pstmt.executeQuery();

        if (rs.next()) {
            /*leer los valores del ResultSet*/
            int idPaciente = rs.getInt("id");
            String nombre = rs.getString("nombre");
            String apellido = rs.getString("apellido");
            String dni = rs.getString("dni");
            String genero = rs.getString("genero");
            String direccion = rs.getString("direccion");
            String telefono = rs.getString("telefono");
            String email = rs.getString("email");
            String antecedentes_medicos =
                rs.getString("antecedentes_medicos");
            String alergias = rs.getString("alergias");
            String padecimientos =
                rs.getString("padecimientos");

            // Crear un objeto Paciente con los
```

```

valores obtenidos
    Paciente paciente = new Paciente(nombre,
                                      apellido,
                                      dni, genero, direccion, telefono, email,
                                      antecedentes_medicos, alergias, padecimientos);
    System.out.println("\nPACIENTE
                        ENCONTRADO\n" + "-----\n");
    System.out.println("id: " + idPaciente);
    System.out.println(paciente.toString());
}
} finally {
    close(pstmt);
    close(connection);
}

}

public void obtenerTodosLosPacientes() throws SQLException {
    // este método muestra todos los pacientes
    Connection connection = null;
    PreparedStatement pstmt = null;

    try {
        connection = conexionBD.getConnection();
        String query = "SELECT * FROM pacientes ORDER
                        BY id ASC ";

        pstmt = connection.prepareStatement(query);
        ResultSet rs = pstmt.executeQuery();
        if(!rs.isBeforeFirst()) {
            /*Si isBeforeFirst() devuelve true,
            no hay registros en el conjunto de resultados*/
            System.out.println("No hay pacientes
                              registrados");
        } else {
            /*Si isBeforeFirst() devuelve false, hay al
            menos un registro en el conjunto de
            resultados*/
            while (rs.next()) {
                // Se muestran todos los pacientes
                String nombre = rs.getString("nombre");
                String apellido = rs.getString("apellido");
                String dni = rs.getString("dni");
                String genero = rs.getString("genero");
                String direccion =
                    rs.getString("direccion");
                String telefono = rs.getString("telefono");
                String email = rs.getString("email");
                String antecedentes_medicos =
                    rs.getString("antecedentes_medicos");
                ;
                String alergias = rs.getString("alergias");
                String padecimientos =
                    rs.getString("padecimientos");
            }
        }
    }
}

```

```
// Crear un objeto Paciente con los valores
// obtenidos

    Paciente paciente = new Paciente(nombre,
    apellido, dni, genero, direccion, telefono,
    email, antecedentes_medicos, alergias,
    padecimientos);
    System.out.println("\nPACIENTE\n" +
        "-----\n");
    System.out.println("id: " +
rs.getInt("id"));
    System.out.println(paciente.toString());
    System.out.println();
    }

} finally {
    close(pstmt);
    close(connection);
}

void eliminarPaciente(int id) throws SQLException {
    /* este método recibe un objeto de la clase Paciente
    que usaremos para acceder a los getter de cada atributo*/
    Connection connection = null;
    PreparedStatement pstmt = null;

    try {
        connection = conexionBD.getConnection();
        String query = "DELETE FROM pacientes WHERE
                                                                id = ?";
        pstmt = connection.prepareStatement(query);
        pstmt.setInt(1, id);

        int rowsDeleted = pstmt.executeUpdate();
        if (rowsDeleted > 0) {
            System.out.println("Se ha eliminado el
                paciente" + " con idPaciente = " + id);
        }
    } finally {
        close(pstmt);
        close(connection);
    }
}

public void actualizarPaciente(Paciente paciente, int id)
                                throws SQLException {
    Connection connection = null;
    PreparedStatement pstmt = null;

    try {
        connection = conexionBD.getConnection();
        String query = "UPDATE pacientes SET nombre=?,
```

```
        apellido=?, dni=?, genero=?,
        direccion=?, telefono=?, email=?,
        antecedentes_medicos=?, alergias=?,
        padecimientos=? WHERE id=?";

        pstmt = connection.prepareStatement(query);
        pstmt.setString(1, paciente.getNombre());
        pstmt.setString(2, paciente.getApellido());
        pstmt.setString(3, paciente.getDni());
        pstmt.setString(4, paciente.getGenero());
        pstmt.setString(5, paciente.getDireccion());
        pstmt.setString(6, paciente.getTelefono());
        pstmt.setString(7, paciente.getEmail());
        pstmt.setString(8, paciente.getAntecedentesMedicos());
        pstmt.setString(9, paciente.getAlergias());
        pstmt.setString(10,
            paciente.getPadecimientos());
        pstmt.setInt(11, id);

        int rowsUpdated = pstmt.executeUpdate();
        if (rowsUpdated > 0) {
            System.out.println("Paciente
                                actualizado con éxito.");
        } else {
            System.out.println("No se pudo
                                actualizar el
                                paciente.");
        }
    } finally {
        close(pstmt);
        close(connection);
    }
}

}
```

Clase Main.java

```
package EjercicioBDPacientes;

import java.sql.SQLException;

public class Main {

    public static void main(String[] args) {
        PacienteDAO pacienteDAO = new PacienteDAO();
        try {
            // Crear objetos de tipo Paciente con nuevos datos
            Paciente paciente1 = new Paciente("Purificación",
                "Lombo", "30810256A", "Femenino", "Calle Nueva
                560",
                "652366569", "puri.barr@hotmail.com", "Ninguno",
                "Ninguna", "Ninguno");
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

```
Paciente paciente2 = new Paciente("Lola", "Bueno",
    "96956789B", "Femenino", "Calle Ancha, 33",
    "987654321", "Lola.mar@gmail.com", "Otras",
    "Ninguna", "muchos");

Paciente paciente3 = new Paciente("Francisco",
    "Campos", "23456789B", "Masculino", "Calle Ancha,
    33",
    "987654321", "estivaliz.mar@gmail.com", "Ninguno",
    "Ninguna", "Ninguno");

//Obtener un listado de todos los pacientes
pacienteDAO.obtenerTodosLosPacientes();

/*Buscar un paciente con un id determinado y
que exista en la base de datos*/
pacienteDAO.obtenerPacientePorID(1);

//Actualizar un paciente existente
pacienteDAO.actualizarPaciente(paciente3, 1);

/*Eliminar un paciente por su id,
debe existir en la Base de datos*/
pacienteDAO.eliminarPaciente(2);

//insertar los objetos paciente
pacienteDAO.insertarPaciente(paciente1);
pacienteDAO.insertarPaciente(paciente2);

} catch (SQLException e) {
    e.printStackTrace();
}

}
```

Una vez realizado todo el código ejecutamos la clase Main y se obtendrá la siguiente información en consola.

PACIENTE

id: 1

Nombre: María

Apellido: Fernández

DNI: 34525236X

Genero: Masculino

Direccion: C/Cuesta, 4,1º dcha

Telefono: 659562325

Email: mariaFern@gmail.com

Antecedentes_medicos: ninguno

Alergias: gramíneas

Padecimientos: artritis

PACIENTE

id: 2

Nombre: Luisa

Apellido: Gómez

DNI: 11152528X

Genero: Femenino

Direccion: C/Larga, 5,dcha

Telefono: 696425325

Email: LuisaGom@gmail.com

Antecedentes_medicos: familiares

Alergias: ibuprofeno

Padecimientos: migrañas

PACIENTE ENCONTRADO

id: 1

Nombre: María

Apellido: Fernández

DNI: 34525236X

Genero: Masculino

Direccion: C/Cuesta, 4,1º dcha

Telefono: 659562325

Email: mariaFern@gmail.com

Antecedentes_medicos: ninguno

Alergias: gramíneas

Padecimientos: artritis

Paciente actualizado con éxito.

Se ha eliminado el paciente con idPaciente = 2

Se ha insertado el paciente correctamente

Se ha insertado el paciente correctamente

Si volvemos a phpMyAdmin y actualizamos la información de la tabla pacientes, podemos observar que se ha actualizado el paciente con id = 1, eliminado el paciente con id = 2 e insertado dos nuevos pacientes.

phpMyAdmin

Mostrando filas 0 - 2 (total de 3, La consulta tardó 0,0003 segundos)

SELECT * FROM `pacientes`

Perfilando [Editar en línea] [Editar] [Explicar SQL] [Crear código PHP] [Actualizar]

Mostrar todo | Número de filas: 25 | Filtrar filas: Buscar en esta tabla | Ordenar según la clave: Ninguna

Opciones extra

	id	nombre	apellido	dni	genero	direccion	telefono	email	antecedentes_medicos	alergias	padecimientos
<input type="checkbox"/>	1	Francisco	Campos	23456789B	Masculino	Calle Ancha, 33	987654321	estivaliz.mar@gmail.com	Ninguno	Ninguna	Ninguno
<input type="checkbox"/>	3	Purificación	Lombo	30810256A	Femenino	Calle Nueva 560	652386569	puri.barr@hotmail.com	Ninguno	Ninguna	Ninguno
<input type="checkbox"/>	4	Lola	Bueno	96956789B	Femenino	Calle Ancha, 33	987654321	Lola.mar@gmail.com	Otras	Ninguna	muchos

Seleccionar todo | Para los elementos que están marcados: Editar | Copiar | Borrar | Exportar

Mostrar todo | Número de filas: 25 | Filtrar filas: Buscar en esta tabla | Ordenar según la clave: Ninguna

Operaciones sobre los resultados de la consulta

Imprimir | Copiar al portapapeles | Exportar | Mostrar gráfico | Crear vista

Guardar esta consulta en favoritos

Etiqueta: ☐ Permitir que todo usuario pueda acceder a este favorito

Resultado de ejecutar clase Main

Fuente: Elaboración propia

Para el ejemplo solo hemos usado la consola para mostrar mensajes, no se han usado una interface gráfica, pero una vez tengas claro cómo funciona el acceso a la base de datos desde Java, puedes incorporar una interfaz gráfica usando por ejemplo JavaFX.



NOTA DE INTERÉS

Dentro de una clase podemos hacer referencia a los atributos `this` o simplemente no lo ponemos. Ej: `this.nombre` ó `nombre`.

RESUMEN FINAL

Al desarrollar una aplicación en Java, normalmente se tendrá que manejar una cantidad de datos grande. Estos datos normalmente estarán almacenados en un sistema gestor de bases de datos. En esta unidad se ha comenzado analizando el concepto de bases de datos y sus tipos, en concreto, las bases de datos orientadas a objeto y las bases de datos relacionales.

A continuación, se ha profundizado en el concepto y la utilidad de las bases de datos en general, que son estructuras de almacenamiento utilizadas para almacenar, organizar y gestionar grandes cantidades de datos, proporcionar acceso rápido y eficiente a la información y garantizar la seguridad y privacidad de los datos almacenados.

Se ha tratado la forma de acceso a bases de datos, pero centrándonos principalmente en el acceso a bases de datos relacionales desde una aplicación Java. Hemos podido comprobar que existen diferentes características, tipos y formas de realizar una conexión con una base de datos relacional utilizando la API JDBC que ofrece a los desarrolladores Java un modo de conectar con dichas bases de datos. Posteriormente se han visto los distintos tipos de drivers existentes para esta conexión, como son el Tipo 1: Puente JDBC-ODBC, el tipo 2: Driver API nativo/parte Java, el tipo 3: Driver protocolo de red/todo Java y el tipo 4: Driver protocolo nativo/todo Java, cada uno con sus ventajas y desventajas.

Se ha aprendido a configurar un entorno de desarrollo para trabajar con el servidor local XAMPP, usando bases de datos de tipo MySQL.

Para finalizar esta unidad nos hemos adentrado en la forma en la que se recuperan los datos y cómo se manipulan: Altas, bajas y modificaciones, utilizando el lenguaje SQL en combinación con las interfaces Statement y ResultSet muy importantes en Java que son utilizadas para interactuar con bases de datos a través de la API JDBC. ResultSet es una interfaz en Java que representa un conjunto de resultados de una consulta SQL ejecutada en una base de datos. Proporciona métodos para acceder y manipular los datos recuperados de la base de datos. Un objeto ResultSet se crea a partir de un Statement o PreparedStatement (ejecuta instrucciones SQL dinámicas que contienen parámetros cuyos valores pueden cambiar en tiempo de ejecución) después de ejecutar una consulta y se utiliza para iterar a través de los registros resultantes y acceder a los valores de las columnas. En resumen, ResultSet y Statement son componentes clave en la API JDBC que permiten a los desarrolladores interactuar con bases de datos relacionales desde aplicaciones Java. ResultSet se utiliza para recuperar y procesar datos resultantes de consultas, mientras que Statement se utiliza para enviar instrucciones SQL a la base de datos.