

UNIDAD 8: GESTIÓN DE BASES DE DATOS RELACIONALES

Módulo Profesional: Programación

ÍNDICE

| | |
|---|----|
| RESUMEN INTRODUCTORIO..... | 3 |
| INTRODUCCIÓN..... | 3 |
| CASO INTRODUCTORIO | 3 |
| 1. INTRODUCCIÓN | 5 |
| 2. CONEXIÓN CON BASES DE DATOS RELACIONALES. CARACTERÍSTICAS, TIPOS Y MÉTODOS DE ACCESO | 7 |
| 2.1 Tipo 1: driver puente JDBC-ODBC..... | 8 |
| 2.1.1 Ventajas | 9 |
| 2.1.2 Desventajas..... | 9 |
| 2.2 Tipo 2: driver API nativo/parte Java | 10 |
| 2.2.1 Ventajas | 11 |
| 2.2.2 Desventajas..... | 11 |
| 2.3 Tipo 3: driver protocolo de red/todo Java..... | 12 |
| 2.3.1 Ventajas | 13 |
| 2.3.2 Inconvenientes..... | 13 |
| 2.4 Tipo 4: driver protocolo nativo/todo Java | 14 |
| 2.4.1 Ventajas | 14 |
| 2.4.2 Desventajas..... | 15 |
| 3. ESTABLECIMIENTO DE CONEXIONES. COMPONENTES DE ACCESO A DATOS | 16 |
| 3.1 Los URL de JDBC..... | 16 |
| 3.2 Clase DriverManager | 17 |
| 4. RECUPERACIÓN DE INFORMACIÓN. SELECCIÓN DE REGISTROS. USO DE PARÁMETROS | 21 |
| 5. MANIPULACIÓN DE LA INFORMACIÓN. ALTAS, BAJAS Y MODIFICACIONES. EJECUCIÓN DE CONSULTAS SOBRE LA BASE DE DATOS | 24 |
| RESUMEN FINAL | 28 |

RESUMEN INTRODUCTORIO

En esta unidad se va a tratar en primer lugar el concepto de driver para realizar una conexión con bases de datos. Pasaremos después a ver las características, tipos y métodos de acceso a una base de datos desde Java. Después veremos cómo establecer una conexión para pasar después a tratar cómo se recupera la información. 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 recuperarlos. 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 la unidad el alumnado:

- Será capaz de realizar la conexión con distintos tipos de bases de datos gracias a la API JDBC.
- Conocerá los tipos de drivers JDBC existentes.
- Realizará consultas y modificaciones sobre bases de datos utilizando la API JDBC.

1. INTRODUCCIÓN

Los fabricantes proporcionan APIs que permitan acceder a sus bases de datos. Un hecho que podría dar problemas a los programadores si no existiera JDBC: una 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 del modelo de base de datos que se utilice. 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.

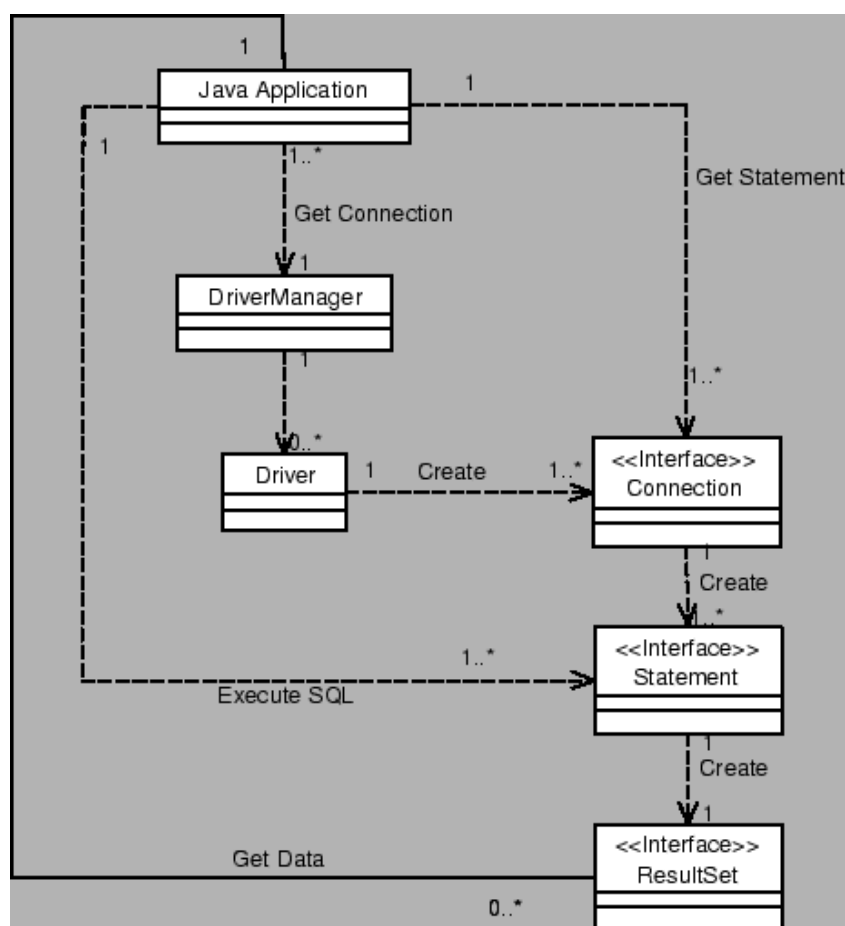


Imagen: Arquitectura general de una aplicación JDBC



ENLACE DE INTERÉS

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

- [Java JDBC API](#)

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

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.

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.



PARA SABER MÁS

Existen otros estándares para la conexión con bases de datos como por ejemplo ODBC del que puede ampliar información realizando la siguiente lectura:

- [Open Database Connectivity](#)

2.1 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 la siguiente figura:

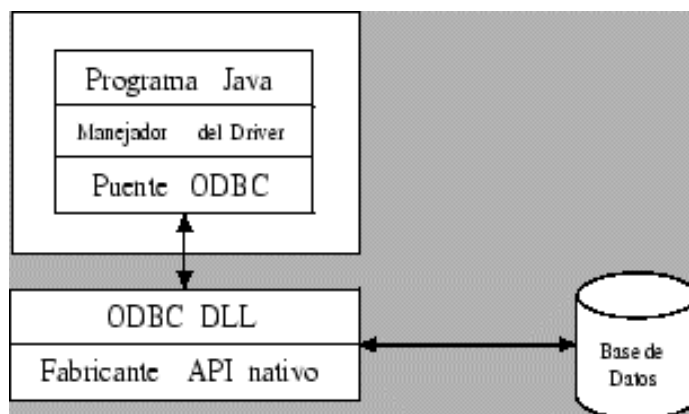


Imagen: Driver JDBC Tipo 1

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

2.1.1 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.

2.1.2 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.

2.2 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 la siguiente figura:

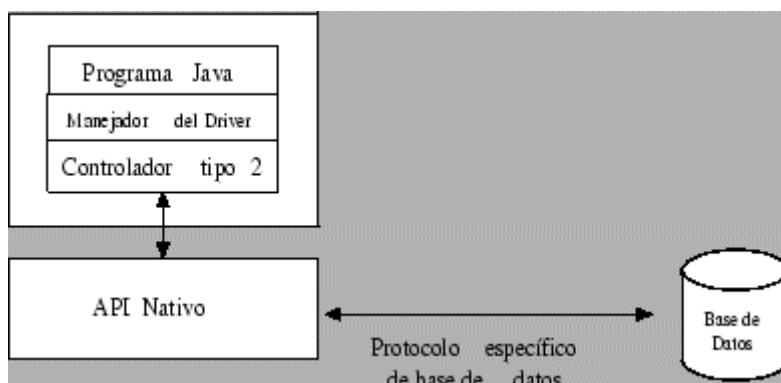


Imagen: Driver JDBC Tipo 2

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.

2.2.1 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.



ENLACE DE INTERÉS

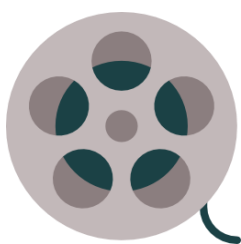
A continuación, se deja un enlace donde ampliar información sobre los cuatro tipos de drivers JDBC existentes:

- [DriversJDBC](#)

2.2.2 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.



VIDEO DE INTERÉS

En el siguiente apartado se estudiará cómo realizar la conexión con Java a bases de datos. Se recomienda visionar este vídeo para tener una visión general de lo que se va a estudiar.

- [Acceso a BBDD. JDBC I](#)

2.3 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.

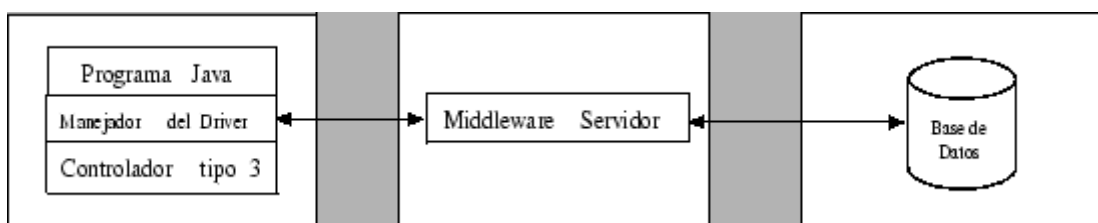


Imagen: Driver JDBC Tipo 3

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.



ENLACE DE INTERÉS

Aunque en esta unidad se ha hablado de cuatro tipos de drivers JDBC, en algunas fuentes introduce un tipo más. En el siguiente enlace se puede obtener información sobre este quinto tipo:

- [What Are the Types of JDBC Drivers?](#)

2.3.1 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 proxis y los drivers del tipo 3 ofrecen normalmente estas características.

2.3.2 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.

2.4 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.



ARTÍCULO DE INTERÉS

En la revista Javaworld hay un tutorial en inglés sobre cómo crear nuestros propios drivers JDBC:

- [Create your own type 3 JDBC driver](#)

2.4.1 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.

2.4.2 Desventajas

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

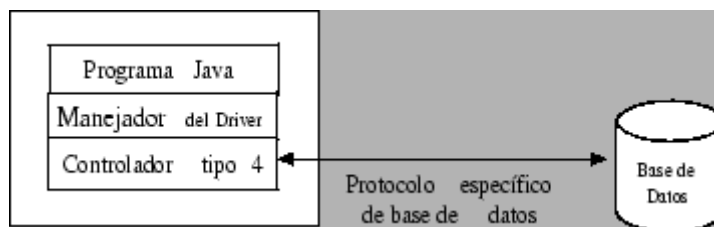


Imagen: Driver JDBC Tipo 4



ENLACE DE INTERÉS

En las siguientes referencias, encontrarás cómo realizar la conexión entre MySQL y JAVA en el entorno de Desarrollo NetBeans:

- [Conectar una base de datos en MySQL con NetBeans](#)
- [JDBC: Conexión a Bases de datos Mysql](#)
- [Los 7 pasos a seguir para el manejo de MySQL con Java](#)

3. ESTABLECIMIENTO DE CONEXIONES. COMPONENTES DE ACCESO A 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.

3.1 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.

3.2 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.



ENLACE DE INTERÉS

En la siguiente web encontrará un ejemplo de conexión a MySQL usando las clases `Class`, `Connection`, `DriverManager`, `ClassNotFoundException` y `SQLException`:

- [Aplicación de ejemplo](#)

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()`).

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 encontrará en el enlace:

- [Class DriverManager](#)

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:

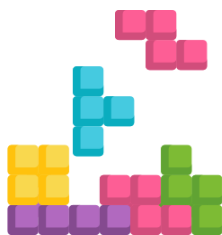
- **java.sql.DriverManager**. Recupera el driver apropiado del conjunto de drivers registrados.
- **public static Connection getConnection(String url) throws SQLException**. El URL de la base de datos está especificado en la forma de `jdbc:subprotocol:subname`. Para poder obtener una conexión a la base 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.



EJEMPLO PRÁCTICO

Fíjese en el siguiente 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.

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.

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

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
```

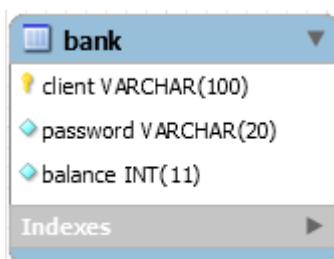
La interfaz java.sql.ResultSet ofrece varios métodos para recuperar los datos que se obtienen de realizar una consulta:

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

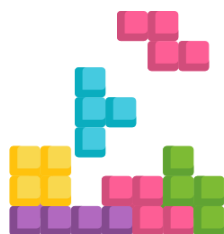
Todos estos métodos requieren el nombre de la columna (tipo String) o el índice de la columna (tipo int) como argumento. La sintaxis para las dos variantes del método getString() es la siguiente:

```
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 la siguiente 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.

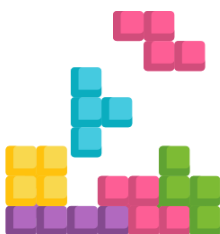
Conocido el número de columnas, se puede obtener la información de tipo asociada a cada una de ellas.



ENLACE DE INTERÉS

En mi zona Java podrá encontrar información sobre el objeto ResultSet y ejemplo de uso:

- [ResultSet](#)



EJEMPLO PRÁCTICO

Por ejemplo, el siguiente 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.

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

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.



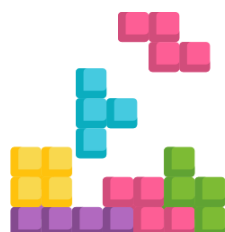
ENLACE DE INTERÉS

Para conocer más sobre la interfaz Statement visite:

- [Interface Statement](#)

Para crear la tabla bank, hay que ilustrar el API JDBC y así 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.



EJEMPLO PRÁCTICO

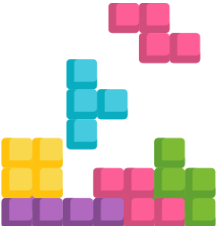
Clase `CreateTableBank`

```
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 como uno de los ejemplos prácticos.



EJEMPLO PRÁCTICO

Método insertData():

```

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).

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 las características, tipos y formas de realizar una conexión con una base de datos relacional. Posteriormente se han visto los distintos tipos de drivers existentes para esta conexión.

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.