Asignatura .Programación de Aplicaciones

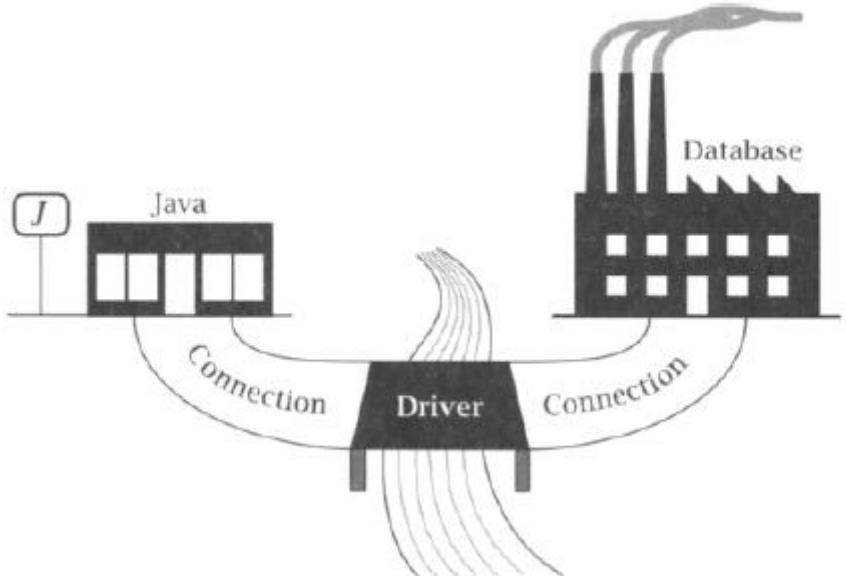
Anexo .JDBC Java DataBase Conectivity

Índice de contenidos

1. Introducción JDBC

La API JDBC (Java DataBase Conectivity) permite acceder a cualquier clase de base de datos relacional desde el lenguaje Java.

La siguiente figura muestra una analogía para entender mejor el concepto de JDBC



Esencialmente una interacción con JDBC consta de los siguientes pasos

1. Abrir una conexión con la base de datos.
2. Enviar una sentencia SQL al gestor de base de datos.
3. Recuperar y procesar los resultados devueltos como resultado de la ejecución del comando SQL.

El siguiente código muestra un ejemplo de estos tres pasos

```
4. Class.forName("com.mysql.jdbc.Driver").newInstance();
5. Connection con = DriverManager.getConnection
6.            ( " jdbc:mysql://localhost/laboratorio ", "root","root");
7.
8. Statement stmt = con.createStatement();
9. ResultSet rs = stmt.executeQuery("select * from usuarios ");
10.  while (rs.next()) {
11.          int id = rs.getInt("Id");
12.          String pass = rs.getString("password");
13.                  }
```

- La línea 4 carga el driver para interactuar con una base de datos MySQL
- La línea 5 establece una conexión con la base de datos MySQL
- La línea 8 crea una sentencia SQL sobre la conexión

- Línea 9 envía la sentencia al SGBD para su ejecución , el resultado de la sentencia se devuelve en forma de ResultSet
- Las líneas 10,11,12,13 procesan cada una de la filas de la tabla devuelta en el objeto ResultSet

2. Arquitecturas JDBC

Modelos de dos y tres capas

JDBC soporta modelos de dos y tres capas para el acceso a base de datos

- En el modelo de dos capas la aplicación Java interactúa con la base de datos a través de un controlador cargado en la propia aplicación .La base de datos puede residie en la misma máquina o en una máquina en red (configuración cliente/servidor).

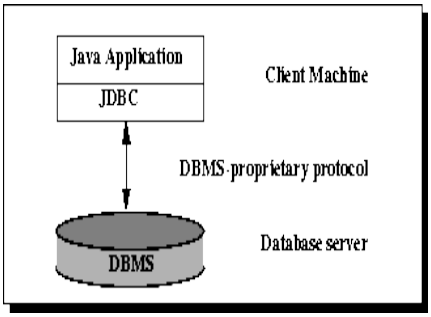


Figura 2: arquitectura de acceso a datos de dos capas.

- En el modelo de tres capas la aplicación Java interactúa con la base de datos a través de una capa de servicio intermedia, ubicada en un servidor que optimiza este tipo de interacción. Esto tiene la ventaja de desacoplar al cliente de los detalles del gestor de base de datos. Esta arquitectura permite la implementación de transacciones distribuidas, fuentes desconectadas y gestión de pool de conexiones incrementado el rendimiento.

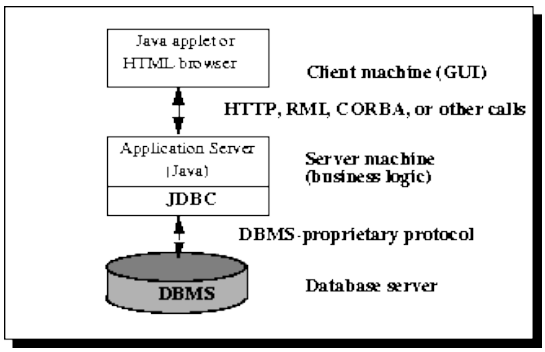


Figura 3: arquitectura de acceso a datos de tres capas.

3. Abrir la conexión con la Base de Datos

3.1. Cargar el Driver

Sólo implica una línea de código. Si, por ejemplo, queremos utilizar el driver de MySQL seria de la siguiente forma:

```
Class.forName("com.mysql.jdbc.Driver ").newInstance();
```

Una vez cargado el driver, es posible establecer una conexión con un controlador de base de datos.

3.2. Obtener una conexión

Una vez cargado el Driver de la base de datos, el segundo paso para establecer una conexión es tener el driver apropiado conectado al controlador de base de datos. La siguiente línea de código ilustra la idea general para realizar esto.

```
Connection con = DriverManager.getConnection

           ( " jdbc:mysql://localhost/laboratorio ", "root","root");
```

Dónde:

- **jdbc:mysql** indica que se va usar un driver nativo para mysql **.**Si estamos utilizando un puente JDBC desarrollado por una tercera parte, la documentación nos dirá el subprotocolo a utilizar, es decir, qué poner después de **jdbc:** en la URL
- **//localhost** indica la máquina donde reside el sistema gestor de base de datos.En este ejemplo es la máquina local pero en caso de ser remota irá definida con una IP
- **/laboratorio** indica el nombre de la base de datos a la cual se desea conectar
- **root,root** indica las credenciales (nombre y contraseña) que se usarán en la conexión.

La clase **DriverManager**, como su nombre indica, maneja todos los detalles del establecimiento de la conexión. A menos que estemos escribiendo un driver, posiblemente nunca utilizaremos ningún método del interface **Driver**, y el único método de **DriverManager** que realmente necesitaremos conocer es **DriverManager.getConnection**.(…)

La conexión devuelta por el método **DriverManager.getConnection** es una conexión abierta que se puede utilizar para crear sentencias JDBC que pasen nuestras sentencias SQL al controlador de la base de datos.

3.3. Ejecutar una sentencia SQL

Las sentencias SQL de manipulación de datos (DML) que se pueden ejecutar sobre una base de datos son :

- SELECT selección de filas
- INSERT inserción de filas
- UPDATE actualización de filas
- DELETE borrado de filas

La API JDBC permite enviar cualquiera de estas sentencias a través de la clase Statement. A continuación se muestra la forma general de crear una sentencia

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

Donde `con` es un objeto de tipo `Connection`

Statement abstrae el concepto de comando SQL y siempre debe estar asociado a una conexión activa

A continuación simplemente ejecutamos la sentencia SQL(en este caso una sentencia SEECT) de la siguiente forma:

```
ResultSet rs = stmt.executeQuery("select * from usuarios ");
```

Donde el objeto `rs` es un objeto `ResultSet` que representa a la tabla que se obtiene como resultado de ejecutar la sentencia SQL.

El objeto `Statement` admite la ejecución de sentencias SQL a través de los siguientes métodos

- `ResultSet executeQuery(comando_SQL)`
  - Devuelve una tabla representada por un `ResultSet`
  - Se utiliza para la sentencia SELECT.
- `int executeUpdate(comando_SQL)`
  - Devuelve un entero indicando el número de filas afectadas por la sentencia.
  - Se usa para sentencias UPDATE,DELETE,INSERT,CREATE

Ejemplo de sentencias de modificación de datos

- `int n=stmt.executeUpdate("update usuarios set id=2 where id=12");`
- `smt.executeUpdate("INSERT INTO USUARIOS(Id,nombre,contraseña) VALUES (234,"root","root")")`

3.4. Procesar los resultados de una consulta

JDBC devuelve los resultados de una sentencia SELECT en un objeto `ResultSet`, por eso si se ha ejecutado una sentencia SELECT el resultado será un objeto de tipo `ResultSet` que podrá ser recorrido fila por fila y en sentido hacia delante solamente.

```
while (rs.next()) {

      int id = rs.getInt("Id");

      String pass = rs.getString("password");

            }
```

Donde rs es un objeto de tipo `ResultSet` y para obtener el valor de columna ("nombre") se utiliza el método getXX donde XX indica el tipo de la columna con ese "nombre"

Los métodos **getXXX** del tipo apropiado se utilizan para recuperar el valor de cada columna. Por ejemplo, la primera columna de cada fila de **rs** es **Id**, que almacena un valor del tipo **INTEGER** de SQL. El método para recuperar un valor **INTEGER** es **get Int** sin embargo para un tipo VARCHAR sería **getString**.

La siguiente tabla muestra la correspondencia de los métodos de acceso sobre tipos java y su equivalencia de tipos SQL

	LONGVARCHAR	BINARY	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP	VARCHAR
getByte	x	x						
getShort	x	x						
getInt	x	x						
getLong	x	x						
getFloat	x	x						
getDouble	x	x						
getBigDecimal	x	x						
getBoolean	x	x						
getString	X	x	x	x	x	x	x	x
getBytes			X	X	x			
getDate	x	x				X		x
getTime	x	x					X	x
getTimestamp	x	x				x	x	X
getAsciiStream	x	X	x	x	x			
getUnicodeStream	x	X	x	x	x			
getBinaryStream			x	x	X			
getObject	x	x	x	x	x	x	x	x

Otra cuestión importante es que el objeto `ReseultSet` **es de sólo lectura**, lo que impide realizar cualquier modificación sobre los datos que tiene.

Si la sentencia ejecutada es de modificación de datos, el resultado devuelto por el JDBC será el número de filas afectadas

```
int n=stmt.execute("update usuarios set id=2 where id=12");

if (n==0) { System.out.println("Error en la actualización")};
```

4. Sentencias precompiladas (Prepared Statement)

Algunas veces es más eficiente tener la sentencia SQL precompilada en el gestor de base de datos y simplemente invocarla con nuevos parámetros. Esto es útil cuando queremos ejecutar repetidamente la misma sentencia SQL con solamente cambios en los parámetros. Para ello se utiliza el objeto `PreparedStatement` en lugar de `Statement`

```
PreparedStatement stmtPre = con.prepareStatement( "select * from
usuarios where id=?");
```

Donde el `carácter ?` indica que es un parámetro de la consulta

Ahora necesitamos proporciona un valor al parámetro indicado en la consulta

```
 stmtPre.setString(1,"12");
```

Donde 1 indica el número de parámetro y "12" el valor del parámetro.

5. Procedimientos almacenados

JDBC permite crear y llamar a procedimientos almacenados en el sistema gestor de base de datos.

Para realizar esto primero creamos el procedimiento

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

Smt.executeUpdate("create procedure MOSTRAR_USUARIOS " + "as " +
"select * from usuarios");
```

Ahora el procedimiento ya estaría creado, para llamarlo debemos usar un objeto `Callablestatement`

```
CallableStatement cs = con.prepareCall("{call MOSTRAR_USUARIOS}");

ResultSet rs = cs.executeQuery();
```

6. Transacciones

Cuando se crea una conexión por defecto se configura en modo auto confirmación (auto-commit). Esto significa que cada sentencia SQL que se lance sobre esa conexión es

tratada como una transacción única y será automáticamente confirmada y acometida justo después de ser ejecutada.

Establecer la confirmación de forma explicita

La forma de permitir que dos o más sentencia sean agrupadas en una transacción es desactivar el modo auto-confirmación (auto commit)

A continuación se muestra la instrucción que realiza esto, donde **con** es una conexión activa.

con.setAutoCommit(false);

Una vez que se ha desactivado la auto-confirmación, no se confirmará ninguna sentencia SQL hasta que no llamemos explícitamente al método **commit (confirmar)**. Todas las sentencias ejecutadas después de la anterior llamada al método **commit** serán incluidas en la transacción actual. El siguiente código, en el que **con** es una conexión activa, ilustra una transacción.

```
con.setAutoCommit(false);

PreparedStatement updateSales = con.prepareStatement( "UPDATE COFFEES
SET SALES = ? WHERE COF_NAME LIKE ?");

updateSales.setInt(1, 50);

updateSales.setString(2, "Colombian");

updateSales.executeUpdate();

PreparedStatement updateTotal = con.prepareStatement( "UPDATE COFFEES
SET TOTAL = TOTAL + ? WHERE COF_NAME LIKE ?");

updateTotal.setInt(1, 50);

updateTotal.setString(2, "Colombian");

updateTotal.executeUpdate();

con.commit();

con.setAutoCommit(true);
```

7. Bibliográfica

- Trail JDBC(TM) Database Access (The Java™ Tutorials)