**Conexión Java y SQL**

Para conectar Java y SQL utilizaremos **JBDC** (Java Databases Connectivity) que **es** una **interfaz** de **programación** **de** **aplicaciones** (API) que **permite** a los **programas** **Java** **acceder** **a** **bases** **de** **datos**.

El ciclo de ejecución básico para conectarse a una bdd y hacer consultas consiste en:

1. Cargar el driver JDBC
2. Establecer una conexión a la bdd
3. Ejecutar sentencias mientras la conexión este abierta
4. Cerrar conexión

Las sentencias pueden ser consultas o actualizaciones

**Consulta**:

Armamos una sentencia SQL que generalmente será un String, la creamos y la ejecutamos, esto nos devolverá un resultado sobre el cual iteraremos. Ejemplo: mostrar los datos por una interfaz gráfica.

**Actualización**:

Armamos una sentencia SQL, la creamos y ejecutamos, en este caso no devuelve un resultado. Ejemplo: Cuando utilizamos insert, update, delete.

**Veremos cuales son las clases involucradas para crear una conexión, crear una sentencia, ejecutar una sentencia y recuperar el resultado.**

**Clase** **DriverManager**

Es una clase estática (no requiere instanciación de objetos) que administra los drivers de jdbc para iniciar conexiones.

**Principales métodos:**

**1.Connection getConnection(String url):** Intenta iniciar una conexión con una bdd según los parámetros especificados en el url.

Generalmente el url tiene el siguiente formato:

jbdc:<driver> : <propiedades de la conexion>

Para mysql:

jbdc:mysql://<servidor>:<puerto>/<base\_datos>?<parametros>

Ejemplo concreto, donde nos conectamos al servidor que está en la misma maquina donde se está ejecutando la aplicación y escucha al puerto 3306 (puerto por defecto), donde queremos acceder a la bdd batalla pasandole también el usuario y contraseña:

jbdc:mysql://localhost:3306/batallas?&user=barco&password=pwbarco

**2.Connection getConnection(String url, String usuario, String clave):** idem al anterior pero separando url, usuario y clave.

**3.setLoginTimeOut(int segundos):** Configura la cantidad de segundos de espera para intentar establecer la próxima conexión a una bdd.

O sea, cuando intentamos hacer una conexión le seteamos cuantos segundos espera si no recibe una respuesta del servidor para iniciar nuevamente la conexión.

**Clase Connection**

Es una interfaz para implementar una sesión cliente-servidor con una bdd.

**Principales métodos:**

**Statement createStatement():** crea una nueva sentencia para ejecutar código SQL en forma directa en el servidor a través de la conexión

**PreparedStatement prepareStatement(String sql):** Crea una sentencia preparada con una estructura predeterminada.

**boolean isValid(int timeout):** Verifica que la conexión este abierta y disponible para ejecutar una operación. Es útil para determinar si la conexión de la red permanece activa desde la ejecución de la última sentencia SQL.

También se puede setear un timeout que es el tiempo máximo que va a esperar para ver si esta conectado o no, pasado ese tiempo devuelve verdadero o falso en base a si está conectado o no.

**close():** Cierra la conexión y libera los recursos utilizados.

**setAutoCommit(boolean autoCommit) / commit() / rollback():** Se utilizan para el manejo de transacciones en la conexión actual.

**Ejemplo:**

Como establecer una conexión mysql a través de jdbc:

// Intento de conexión a una base de datos

String servidor = "localhost:3306";

String baseDatos "batallas";

String usuario admin\_batallas";

String clave = "pwbatallas";

String url = "jdbc:mysql://" + servidor + "/" +baseDatos+ "?serverTimezone=America/Argentina/Buenos\_Aires";

java.sql.Connection cnx; //declaramos var de tipo Connection

try

{

cnx = java.sql.DriverManager.getConnection(url, usuario, clave);

} catch (java.sql.SQLException ex)

{System.out.println("SQLException: " + ex.getMessage());

System.out.println("SQLState: " + ex.getSQLState());

System.out.println("VendorError: " + ex.getErrorCode());}

Si no falla nada, entonces creamos una conexión con el servidor MySQL a partir de ahí podemos crear sentencias y luego ejecutarlas.

A partir de un objeto de tipo Connection podemos crear statements

**Clase Statement**

Se utiliza para ejecutar una sentencia SQL en base a un String estático.

**Principales métodos:**

**boolean execute(String sql):** **Ejecuta** cualquier **sentencia** de **SQL**. **Si** **es** una **consulta** se debe **recuperar** el **resultado** **con** **ResultSet** **getResulSet().**

**int executeUpdate(String sql):** Ejecuta solo comandos de actualización de datos (insert, delete, update). Devuelve un entero que es la cantidad de filas que fueron modificadas por la sentencia.

**ResultSet excuteQuery(String sql):** Ejecuta solo consultas que retornan un resultado en filas o registros.

Básicamente **sirve** para **ejecutar** **sentencias** **como** **select** (mayormente para esto la utilizaremos) o show tables.

**addBatch(String sql) / int [] executeBatch() / clearBatch()** : Ejecuta secuencia de comandos.

**void setQueryTimeOut(int seconds) :** Seteamos un límite máximo de espera para esperar por el resultado de la sentencia.

**close():** cierra la sentencia liberando recursos utilizados.

**Clase ResultSet**

Contiene el resultado de una consulta SQL, estructurado en filas y columnas con el comportamiento de un iterador.

**boolean next():** Avanza el índice interno del iterador a la próxima fila (o tupla). Retorna false si no hay más filas.

**String getString(int columnIndex) / String getString(String columnLabel) :** permiten recuperar los valores de las columnas como un String según su posición en la fila (la primer fila es 1) o mediante su nombre respectivamente.

**int getInt(num\_col) / long getLong(..) / … / boolean getBoolean(..) / Date getDate(..)**, etc: existe una función para cada tipo de Java.

**boolean wasNull():** Verifica si el ultimo valor recuperado de una columna con el valor Null de SQL

**boolean previous() / boolean first() / boolean last() / boolean absolute(int row) /**

**boolean relative(int rows):** Funciones para navegar en el resultado. Con absolute(10) vamos a la fila 10, con relative() si estamos parados en la fila 5 y ponemos relative(5) iríamos a la fila 10.

**ResultSetMetaData getMetaData():** para recuperar los meta datos (cantidad de columnas, cantidad de tipos, etc) del resultado y de las columnas.

**Ejemplo**:

try {

// Se crea una sentencia jdbc para realizar la consulta

java.sql.Statement stmt = cnx.createStatement();

// Se prepara el string SQL dde la consulta

String sql = "SELECT nombre\_barco, id, capitan FROM barcos”;

// Se ejecuta la sentencia y se recibe un resultado

java.sql.ResultSet rs = stmt.executeQuery(sql);

// Se recorre el resultado

while (rs.next()){

String nombreBarco = rs.getString("nombre\_barco");

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

String capitan = rs.getString("capitan”);

}

rs.close();

stmt.close();

)

catch (java.sql.SQLException ex) {}

**Clase ResultSetMetaData**

Si no conocemos el resultado de lo que devolverá un select podemos utilizar esta clase

Esta clase nos permite obtener los tipos y propiedades de las columnas de un conjunto resultado (ResultSet)

Principales métodos:

**int getColumnCount():** Retorna la **cantidad** de **columnas** en el resultado.

**String getColumnLabel(int column), String getColumnName(int column**): **Recupera** el **nombre** modificado o el nombre real de una **columna** respectivamente.

**String getColumnClassName(int column):** **recuperar** la **clase** de **Java** determinada como predefinida para el tipo de dato de una **columna**.

**int getColumnType(int column), String getColumnTypeName(int column):** **recuperar** el **tipo** de dato **SQL** estándar y SQL específico de una **columna** respectivamente. Ejemplo: retorna char o var char, etc.

**int isNullable(int column), boolean isAutoIncrement(int column):** propiedades particulares de una columna.

**boolean isSearchable(int column):** **determina** **si** **es** **posible** **ejecutar** un filtro en el **WHERE** de una **consulta** sobre la columna indicada

**Ejemplo** manipulando datos que no retornan ningún resultado (insert, update, delete)

try {

// Se crea una sentencia jdbc para realizar la consulta

java.sql.Statement stmt = cnx.createStatement();

// Se prepara el string SQL de la inserción

String sql = "INSERT INTO barcos (nombre\_barco, id, capitan) " +

"VALUES ('Bismark', 22, 'Ernst Lindeman')";

// Se ejecuta la inserción

stmt.execute(sql);

// Se retornan los recursos utilizados cerrando la sentencia

stmt.close();

}

catch (java.sql.SQLException ex){

System.out.println("Mensaje: " + ex.getMessage()); // Mensaje retornado por MySQL

System.out.println("Código: " + ex.getErrorCode()); // Código de error de MySQL

System.out.println("SQLState: " + ex.getSQLState()); // Código de error del SQL standart

}

Acá simplemente **insertamos una tupla en la base de datos**

**Ejemplo de Sentencias Preparadas**

* Se utilizan cuando una misma sentencia (consulta o actualización) debe ejecutarse repetidamente con la misma estructura, pero distintos valores.
* Más comúnmente usado en inserciones de registros en masa.
* Acelera la ejecución al evitar la interpretación del SQL de manera individual. Una vez preparada la sentencia en el servidor, sólo se envían los datos efectivos.
* El método puede interpretarse como una pre-compilación de las sentencias para un posterior uso repetitivo. No sólo se ahorran ciclos de cpu en el servidor, sino también ancho de banda para la trasmisión de comandos y datos

String sql = "INSERT INTO barcos (nombre\_barco, id, capitan)

VALUES (?, ?, ?)"; //Lo que haremos luego sera insertar para cada signo de pregunta un //valor

// Se crea una sentencia preparada

java.sql.PreparedStatement stmt = cnx.prepareStatement(sql);

// Se ligan los parámetros efectivos

stmt.setString(1, "Bismark"); //Al valor de la posición 1 le asignamos “Bismark”

stmt.setInt(2, 22); //idem al anterior

stmt.setString(3, "Ernst Lindeman"); //idem al anterior

// Se ejecuta la inserción

stmt.executeUpdate();

// se cierra la sentencia

stmt.close();