Bases de datos con JDBC

Sesión 4: Opciones avanzadas

Índice

- Fuentes de datos
- Pool de conexiones
- Transacciones
- RowSets

Fuentes de datos (*DataSources*)

- Hasta ahora, cada vez que queríamos conectarnos con una BD debíamos proporcionar su dirección URL
- Si cambiamos la BD, debemos recompilar nuestra aplicación
- Esto puede ser crítico en sistemas con gran cantidad de programas
- El objetivo de las fuentes de datos es realizar un enlace “lógico” con una base de datos

Uso de las fuentes de datos: Paso 1

- Crearemos un objeto DataSource con los datos de la conexión (driver, dirección, datos de usuario, etc.)
- Enlazaremos el objeto creado con un nombre lógico, mediante JNDI
- JNDI nos permite asociar un nombre (“miDS”) con el objeto *DataSource* que se encarga de realizar la conexión

Uso de las fuentes de datos: Paso 2

- En nuestra aplicación pediremos a JNDI que nos devuelva el objeto asociado a “miDS”
- Si cambiamos la BD no es necesario recompilar nuestra aplicación, sólo decirle a JNDI que “miDS” ahora enlaza a otra fuente de datos
- El objeto *DataSource* nos permite crear una conexión y entonces usarla tal como lo hacíamos antes
- Estos pasos los suele hacer el servidor de aplicaciones (TomCat lo hace también)

Ejemplo de código para obtener una conexión

```
// Se crea un contexto inicial (JNDI)
Context contexto = new InitialContex();
// JNDI nos proporciona el objeto DataSource
DataSource ds =
        (DataSource)context.lookup("jdbc/MySQL");
// Obtenemos la conexión del objeto DataSource
conexion = ds.getConnection();
```

Pool de conexiones

- El proceso de conexión a una BD es el más costoso temporalmente
- En una aplicación de servidor, normalmente el número de conexiones es enorme
- Si continuamente estamos abriendo y cerrando conexiones, el tiempo (y los recursos del sistema) se disparan
- Parece razonable que se intente buscar algún mecanismo para gestionar los recursos críticos

Pool de conexiones

- En este caso, la idea es crear un número de conexiones predeterminado que se abrirán cuando iniciemos nuestra aplicación y permanecerán abiertas durante el tiempo de ejecución
- Si un cliente necesita una conexión se la pide al pool de conexiones
- El cliente usa esa conexión y cuando termina no la cierra, sino que la devuelve al pool para que pueda ser utilizada por otra aplicación

Esquema a seguir para el uso del pool

- Obtener una referencia a la reserva o a un objeto que gestione la reserva.
- Consigue una conexión de una reserva (*getConnection*)
- Utiliza la conexión. Aquí se realizarán todas las consultas y actualizaciones en la base de datos. Un aspecto muy importante aquí son las transacciones. Si una aplicación empieza una transacción y no la termina o la deshace, puede que se produzca una pérdida de consistencia de datos.
- Devuelve la conexión a la reserva. Es muy importante que la aplicación que solicita la reserva no cierre la conexión.

Restricciones

- El pool de conexiones se recomienda para las aplicaciones que cumplan lo siguiente:
  - Las conexiones acceden con un conjunto reducido de cuentas de usuario (el pool necesita que todas las conexiones accedan con el mismo usuario)
  - El acceso a la BD se realiza en una única sesión. Para un carrito de la compra es preferible asignar una conexión dedicada

Servidor de aplicaciones

- Tanto el *DataSource* como el *ConnectionPool* son interfaces que no pueden ser instanciadas directamente
- Los servidores de aplicaciones disponen de sus propias clases que implementan estas interfaces, por lo que veremos su funcionamiento en el bloque de Servidores de aplicaciones
- Normalmente estas características son transparentes al programador. Éste no tiene que preocuparse de realizar él pool o crear la fuente de datos

Transacciones

- En determinadas aplicaciones, cuando tenemos que realizar varias acciones, se requiere que o bien se hagan todas a la vez o bien ninguna
- Reserva de vuelo de Alicante a Osaka:
  - Vuelo Alicante-Madrid
  - Vuelo Madrid-Amsterdam
  - Vuelo Amsterdam-Osaka
- Si alguna de estas reservas no se produce no queremos que se reserve ninguna

Transacción

- Tratamos un conjunto de acciones como una unidad
- Las transacciones hacen que la BD pase de un estado consistente al siguiente
- La mayoría de la BD incorporan un sistema llamado *commit* (“confirmación”)
- Por defecto, se asume que cada vez que realicemos una acción se realiza el *commit* (autocommit)

Utilización de *commit*

- Primero debemos decir que no realice el autocommit en cada acción
- Luego realizamos cada una de las acciones en esta transacción
- Por último realizamos *commit* para hacer los cambios persistentes
- Si se produjo un error (podemos capturar las excepciones) llamaremos a *rollback* para que elimine los cambios realizados

Ejemplo

```
try {
    con.setAutoCommit(false);
    stmt.executeUpdate("INSERT INTO RESERVAS(pasajero,
origen, destino) VALUES('Paquito', 'Alicante', 'Madrid')");
    stmt.executeUpdate("INSERT INTO RESERVAS(pasajero,
origen, destino) VALUES('Paquito', 'Madrid', 'Amsterdam')");
    stmt.executeUpdate("INSERT INTO RESERVAS(pasajero,
origen, destino) VALUES('Paquito', 'Amsterdam', 'Osaka')");
    // Hemos podido reservar los tres vuelos correctamente
    con.commit();
} catch(SQLException e) {
    // Alguno de los tres ha fallado, deshacemos los cambios
    try {
       con.rollback();
     } catch(SQLException e) {};
}
```

Concurrencia en el acceso

- ¿Qué sucede cuando varias aplicaciones acceden a la vez a los mismo datos (para modificarlos)?
- Una aplicación abre una transacción y modifica un dato de una tabla. Justo en ese momento otra aplicación lee y/o modifica ese dato …
- El manejo de la concurrencia es gestionada de distinta manera por los distintos SGBD

Obtención de información

- Podemos saber cuál es el nivel de concurrencia de la BD, llamando a
  - int con.getTransactionIsolation();
- Este método devolverá:
  - Connection.TRANSACTION_NONE No soporta transacciones
  - Connection.TRANSACTION_READ_UNCOMMITTED Soporta transacciones, pero sólo en lectura (MySQL)
  - Connection.TRANSACTION_READ_COMMITTED Si una aplicación ha escrito en un registro, bloquea el registro y ninguna otra aplicación puede escribir en él hasta que la primera no haga *commit* (PostGres)

Interbloqueo

- Un interbloqueo se produce cuando:
  - Aplicación 1: modifica registro A y después B
  - Aplicación 2: modifica registro B y después A
- Se ejecutan las dos a la vez y quedan esperando a que se libere el registro que se quiere modificar
- MySQL no detecta esta situación (no se produce)
- En PostGres se produce, pero detecta la situación y produce una excepción

Puntos de almacenamiento

- Esta característica no está implementada de momento ni en MySQL ni en PostGres
- Permiten definir puntos de control en la transacción
- El método *commit* hacía persistente todas las acciones de la transacción o ninguna
- Podemos definir puntos entre acciones para que se pueda persistir las acciones hasta ese punto

Ejemplo

```
Savepoint sp = null;
boolean realizar_commit = false;
try {
  insertarTabla1 (datos); //Esa acción es más importante. Si se
                          //realizó se debe hacer commit sino
no.
  realizar_commit = true;
  sp = con.setSavepoint();
  insertarTabla2 (datos2);
}
catch {
  // Si sp es null es que falló la primera inserción
  if (sp==null) con.rollback();
  else con.rollback(sp);
}
finally {
  if (realizar_commit) con.commit();
}
```

RowSet

- Un *RowSet* es un intento por simplificar el acceso mediante JDBC
- Con un solo objeto RowSet vamos a ser capaces de gestionar la conexión, la consulta y el resultado
- RowSet es una interfaz. Sun ha realizado una implementación de cinco subclases:
  - por un lado, JdbcRowSet
  - por otro, CachedRowSet, WebRowSet, JoinRowSet y FilteredRowSet
- La principal ventaja de los RowSet es que aunque el driver no lo permite, todos los RowSet son *scrollables* y actualizables
- Veremos los métodos comunes para después comentar las características de estas clases

Datos de conexión

- Como siempre cargamos el driver con Class.forName(“”)
- Para conectarnos a una BD debemos indicar los datos de conexión:
  - crs.setUrl ("jdbc:mysql://localhost/bd");
    crs.setUsername ("miguel");
    crs.setPassword ("cazorla");

Consulta

- Con el mismo objeto creamos el comando y lo ejecutamos
  - crs.setCommand ("Select * from vuelo");
    crs.execute ();
- También disponemos de sentencias preparadas
  - crs.setCommand ("select numero from vuelo where aero_inic=?");
    crs.setString (1, "ALC");
- Desde este punto ya podemos realizar las mismas acciones que hacíamos con ResultSet

Actualización

- Disponemos también de métodos updateXXX para modificar el valor de un campo
- Para que el o los cambios tengan efecto debemos llamar a updateRow () y a continuación a acceptChanges()

- Veremos a continuación las cinco clases antes comentadas

JdbcRowSet

- Es la implementación más sencilla
- Es una capa por encima de ResultSet para que parezca y pueda ser utilizado como un JavaBean
- Dispone de todas las funcionalidades de RowSet
- Podemos crear un objeto de esta clase a partir de un ResultSet
  - ResultSet rs = stmt.executeQuery(q);
  - JdbcRowSet jrs = new JdbcRowSetImpl(rs);

CachedRowSet

- Mantiene en memoria los datos obtenidos en una consulta
- Permite obtener datos de distintas fuentes de datos (fichero de texto, hoja de cálculo, etc.)
- Cuando invocamos a *execute* se conecta con la BD, obtiene los datos y cierra la conexión

JoinRowSet

- Permite crear uniones SQL (Join):
  - select * from vuelo,disponibilidad where vuelo.numero=disponibilidad.numVuelo
- Si tenemos un CachedRowSet con la tabla vuelo y otro con disponibilidad, podemos usar un JoinRowSet:
  - JoinRowSet jrs = new JoinRowSetImpl();
  - jrs.addRowSet(crst,2);
  - jrs.addRowSet(crsu,"DNI");
- Podemos añadir tantos RowSet como queramos. El segundo parámetro indica el campo que tiene que coincidir en los dos RowSet
- La actual implementación da un error

FilteredRowSet

- Permite filtrar los datos de una consulta
- Podemos tener una tabla entera en memoria y aplicando un determinado filtro, que nos muestre sólo aquellos que cumplan un determinado criterio
- La creación se realiza de la misma manera que antes
- Debemos hacer uso de filtros, que son implementaciones de la clase *Predicate*

FilteredRowSet: Ejemplo de filtro

```
public class FiltroDisp implements Predicate{
	private double val;
	public FiltroDisp (double vali) {
		val=vali;
	}
	public boolean evaluate (Object obj, String
	str) {
		return true;
	}
	public boolean evaluate (Object obj, int i) {
		return true;
	 }
	 public boolean evaluate (RowSet rs) {
	    try {
		return (rs.getDouble("Precio") <=
	val);
	    }
	    catch (SQLException e) {
		System.out.println(e); return false;
	    }
    }
}
```

- Este filtro se aplica a la tabla disponibilidad e indica que el precio debe ser menor que el valor pasado al crear el objeto
- En el método *evaluate* podemos poner lo que queramos. Sólo serán visibles aquellos registros que su evaluación sea cierta

FilteredRowSet: asignación de filtro

```
FiltroDisp fd = new FiltroDisp (200.00);
frs.setFilter(fd);
```

- Si el objeto *frs* tiene toda la tabla disponibilidad, después de asignar el filtro sólo estarán visibles aquellos registros cuyo valor sea inferior a 200
- Si después creamos otro filtro y se lo asignamos al mismo objeto, serán visibles los registros que cumplan la nueva condición

WebRowSet

- Esta implementación se realizó para dar soporte a la tecnología Web
- Imaginemos una aplicación cliente-servidor
- El servidor es el encargado de crear el objeto WebRowSet, conectarse a la BD y realizar la consulta
- La consulta se puede volcar a un fichero XML, que es el que se le manda al cliente
- El cliente lee el fichero XML con otro objeto WebRowSet, y gestiona el resultado como si fuera una consulta a una BD
- Permite la comunicación con clientes como móviles, applets, etc.

Ejemplo

- Creamos un objeto WebRowSet
- Ejecutamos la consulta y después el siguiente código
  - FileWriter FW = new FileWriter("ejemplo.xml");
    crs.writeXML (FW);
- Después podemos recuperar la consulta con:
  - FileReader FR = new FileReader("ejemplo.xml");
    crs.readXML (FR);
- En la actual implementación de esta clase no funciona la lectura del fichero

Eventos

- Otra característica importante de los RowSet (a partir de Cached) es el manejo de eventos
- Podemos definir una clase escuchante, que ejecute unas acciones cuando se produzca un evento
- La interfaz permite modificar los siguientes métodos:
  - public void cursorMoved (RowSetEvent evento)
    public void rowChanged (RowSetEvent evento)
    public void rowSetChanged (RowSetEvent evento)

Implementación de la interfaz

- ```
  import javax.sql.*;
  import java.io.*;

  public class Escuchante implements RowSetListener {
    public void cursorMoved (RowSetEvent evento) {
              System.out.println("Se ha movido el cursor");
    }
    .
    .
    .
  }
  ```
- Para asociar un escuchante al RowSet
  - ```
    Escuchante escucha = new Escuchante();
    //crs es un objeto de tipo RowSet
    crs.addRowSetListener(escucha);
    ```

¿Preguntas...?