Introducción al uso de JPA 2.0 y Swing con Firebird Database en Java 6 Standard Edition

Lic. Guillermo Cherencio – UNLu – Programación III – Base de Datos

En el artículo anterior que escribí para Uds. "Introducción al uso de JPA 2.0 con Firebird Database en Java 6 Standard Edition" se mostró una manera de persistir un objeto en una base de datos firebird utilizando JPA 2.0, cabe preguntarse ¿Cómo se hacen otro tipo de operaciones, tales como bajas, cambios y consultas?.

Por otro lado, todo proyecto Java que pretenda realizar algún tipo de interfase gráfica de usuario (GUI), tiene dos frameworks disponibles: AWT (Abstract Windowing Toolkit) y Swing (que esta basado en AWT). Las aplicaciones más nuevas se escriben en Swing, el cual ofrece una serie de componentes, en nuestro caso, si queremos "visualizar" todas las operaciones posibles sobre una base de datos, el componente que más nos facilita esta tarea es `JTable` (una especie de grilla, formada por filas y columnas).

`JTable` opera bajo el patrón de diseño MVC (Model View Controller Pattern), es el patrón clásico utilizado en las interfases gráficas, simplificando, podemos decir que todo nuestro trabajo se divide en tres partes:

- Una clase que actúe como vista (View), responsable de todo "lo visual" (componentes, contenedores, etc.)

- Una clase que actúe como modelo (Model) la cual proveerá de datos (en ambos sentidos -input, output-) a la vista y será ésta quien decida como mostrarlos.

- Una clase que actúe como controlador (Controller), la cual será responsable del tralado de datos entre la vista y el modelo, muchas veces utilizada para implementar los eventos que se produzcan en la vista y responda a éstos.

`JTable` viene a representar la vista (view), posee los metodos getModel() y setModel() para asociar una instancia que actuará como modelo (model) y es la fuente de una serie de eventos que pueden ser capturados por una instancia controlador.

La clase model deberá proveer todos los datos a visualizar en la `JTable`, en este caso, podría ser una colección de objetos de tipo `Cliente`, ello lo podemos obtener de varias formas, una forma posible es a través de una consulta usando JPQL , dentro de la entidad Cliente, podemos incluir varias consultas escritas en JPQL (similar a SQL, pero orientado a objetos), estas

consultas se llaman named queries y si son varios, pueden anotarse de la siguiente forma:

```
...
import javax.persistence.*;
@Entity
@Table(name="trcliente")
@NamedQueries({
 @NamedQuery(name="cliente.todos",query="SELECT c FROM Cliente c"),
 @NamedQuery(name="cliente.porNombre",query="SELECT c FROM Cliente c
WHERE c.nombre LIKE :nombre")
})
public class Cliente {
      ....
}
```

La porcion de codigo de la entidad `Cliente` es bastante autodescriptiva gracias al uso de anotaciones, entonces puedo recuperar de la base de datos una colección de clientes de la siguiente forma:

```
...
       Query qry = em.createNamedQuery("cliente.todos");
       List<Cliente> clientes = qry.getResultList();
...
```

`clientes` es una colección de tipo `List` que en su interior sólo contiene instancias de tipo `Cliente`. Nuestra clase model debería tener conocimiento de esta lista para poder alimentar a la `JTable` con su contenido y del administrador de entidades (`EntityManager`) que usará para persistir los objetos de tipo `Cliente` en la base de datos firebird (en este caso, la referencia estática `Main.em`):

```
...
      JTable tabla = new JTable();
...
      ClienteModel cm = new ClienteModel(clientes,Main.em,tabla);
...
```

Para crear una interfase gráfica, debemos crear un contenedor (`JFrame`), establecerle una layout manager[1] (clase que indica de qué forma se mostrarán sus componentes), en este caso, `BorderLayout`, el cual divide al contenedor en 5 regiones (norte, sur, este, oeste y centro), agregar al contenedor los componentes y/o otros contenedores, establecer un tamaño al contenedor principal y por último visualizarlo:

---

[1] Existen muchos Layout Managers (`FlowLayout`, `BorderLayout`, `GridBagLayout`, `GridLayout`, etc. hasta incluso podemos crear nuestro propio layout manager implementando la interfase `LayoutManager`) que indican el comportamiento de un contenedor en cuanto a cómo acomodar los componentes que contienen y la combinación de éstos junto con los contenedores y componentes que hay dentro de dichos contenedores nos permiten armar la interfaz gráfica. La interfaz grafica se arma como "cajas dentro de cajas".

```
import javax.persistence.*;
import java.awt.BorderLayout;
import java.awt.event.*;
import javax.swing.*;
import java.util.*;
@SuppressWarnings("unchecked")
public class Main {
...
   public static void pantalla() {
       JFrame frame = new JFrame("J2SE + JPA 2.0");
       JTable tabla = new JTable();
       JScrollPane scroll = new JScrollPane(tabla);
       frame.getContentPane().setLayout(new BorderLayout());
       Query qry = em.createNamedQuery("cliente.todos");
       List<Cliente> clientes = qry.getResultList();
       ClienteModel cm = new ClienteModel(clientes,Main.em,tabla);
       tabla.setModel(cm);
       tabla.addKeyListener(cm);
       frame.addWindowListener(new WindowAdapter() { public void
windowClosing(WindowEvent e) { Main.salir(); } });
       frame.getContentPane().add(scroll,"Center");
       frame.setSize(400,400);
       frame.setVisible(true);
   }
}
```

La clase Main posee el metodo estatico `pantalla()` que realiza lo descripto anteriormente, creando un `JFrame` (contenedor principal), en la region centro ubica otro contenedor `JScrollPane` y éste contiene una `JTable` (podemos pensar que un objeto `JTable` es envuelto (wrapped) por un objeto de tipo `JScrollPane`) que permite la visualización de los clientes a través de una instancia de tipo `ClienteModel` que, en este caso, actúa como Model y Controller al mismo tiempo. La linea en color rojo indica que cuando se presione el botón cerrar de la ventana principal (`JFrame`) se ejecutará el método estático `salir()`[2] de la clase `Main`, quien ejecutará los metodos `close()` de `EntityManager` y `EntityManagerFactory` que requerimos para persistir objetos usando JPA 2.0. En Java esto se conoce con el nombre de "modelo de delegación de eventos".

La clase Model-Controller ClienteModel

`JTable` visualizará aquello que le indique `ClienteModel` y para ello, `JTable` necesitará saber muchas cosas: ¿cuántas filas hay?, ¿cuántas columnas?, ¿qué tipo de dato (mejor dicho, de qué clase) es cada columna? ¿qué celdas de la grilla son editables? etc. ¿Cómo puede `ClienteModel` proveer toda esta información? derivando de una clase abstracta que se llama

[2] Aquí se crea una instancia (que se pasa como argumento del método `addWindowListener()` de la clase `JFrame`) de una clase anónima que deriva de `WindowApdater` y de ésta clase anónima se sobreescribe (overwrite) el método `windowClosing()`. Las clases anónimas terminan guardadas en el disco como <Nombre de Clase Contenedora>$<numero correlativo>.class , en este caso: Main$1.class

AbstractTableModel, la cual requiere de la implementación de los siguientes métodos:

Método	Descripción
public String getColumnName(int col)	Devuelve el nombre de la columna col (donde col, va desde 0 a n-1 columnas)
public int getColumnCount()	Devuelve el número total de columnas
public Object getValueAt(int row, int col)	Devuelve una referencia del objeto contenido en la celda ubicada en la fila row y en la columna col
public boolean isCellEditable(int row, int col)	Devuelve true si la celda ubicada en la fila row y en la columna col es editable (modificable), caso contrario, devuelve false si no queremos que el usuario pueda cambiar su contenido
public void setValueAt(Object value, int row, int col)	Idem anterior, pero en este caso, JTable nos indicará que valor debemos poner en la celda -si lo consideramos apropiado-. Permite actualizar el modelo de datos que está "por detrás" de la JTable
public Class getColumnClass(int col)	Debemos retornar una objeto de tipo Class asociado con cada columna col. JTable utilizará el editor de celdas apropiado acorde con el contenido que puede contener la columna.

La correcta implementación de estos métodos hace a la provisión de "ida y vuelta" de datos entre la clase Model (ClientModel) y la clase View (JTable), pero además de esto, generalmente necesitaremos de que el usuario nos pueda indicar qué acciones pretende realizar:

Tecla presionada	Acción a realizar
Ins (Insert, Insertar)	La JTable para a modo "alta", se le pide confirmación al usuario para agregar nuevo registro, si dice que si, sólo se mostrará una fila "en blanco" para ser editada y posteriormente confirmada
Ins (Insert, Insertar)	Presionada en modo "alta" significa que el usuario terminó de cargar los datos de todas las columnas obligatorias y pretende guardar su contenido en la base de datos. También se pide confirmación y se persisten los datos
Del (Delete,Suprimir)	No estando en modo "alta" y teniendo -al menos- una fila seleccionada, borrará de la base de datos la primera fila seleccionada. Pide confirmación.
Cualquier tecla	Se asume como una edición sobre la celda que esta seleccionada, presionando Intro (Enter) se dá por ingresado el cambio y éste se persiste en la base de datos.

La implementación de estas acciones son típicas de una clase Controller, en este caso, para simplificar el apunte, se implementó también dentro de la clase `ClienteModel`, para ello, necesitaremos que `ClienteModel` atienda a los eventos del teclado, se requiere de la implementación de la interfase `KeyListener` (que otro nombre podria tener!), la cual requiere la implementación de los siguientes métodos:

Método	Descripción
`public void keyTyped(KeyEvent e)`	Se tipeó una tecla, no voy a tomar acción, por lo tanto, haré una implementación vacía del método: { }
`public void keyPressed(KeyEvent e)`	Se asumió por presionada una tecla, aquí voy a controlar la tecla presionada y si fue alguna de mi interés (Ins, Del) tomaré acción
`public void keyReleased(KeyEvent e)`	Se soltó una tecla, no voy a tomar acción, implementación vacía: { }

Además de esto, debemos notificar a la vista `JTable` que `ClienteModel` atenderá a los eventos de teclado que genere el usuario que está manipulando la grilla (esto debemos hacerlo en el metodo `pantalla()` de la clase `Main`, código que ya mostramos anteriormente):

```
...
public class Main {
...
   public static void pantalla() {
...
       ClienteModel cm = new ClienteModel(clientes,Main.em,tabla);
       tabla.setModel(cm);
       tabla.addKeyListener(cm);
...
   }
}
```

Implementación Model de ClienteModel

`ClienteModel` recibe en su constructor la lista de los actuales clientes, en su interior se declara su referencia como:

```
    private List<Cliente> lista;
```

Cuando se pasa al modo alta, se utiliza otra lista que contendrá como máximo un solo objeto de tipo `Cliente`:

```
    private List<Cliente> listaAlta = new ArrayList<Cliente>();
```

La clase debe saber cuando esta en "modo alta" o no, cuáles son los nombres de las columnas y de qué clase es cada una, para persistir objetos

debemos contar con la referencia de tipo `EntityManager` (pasada al constructor) y cuando sea necesario crear una referencia de tipo `EntityTransaction`:

```
...
    private boolean modoAlta = false;
    private String columnas[] =
{"Codigo","Nombre","Direccion","Postal","TE"};
    private Class clase_columnas[] = { Integer.class, String.class,
String.class, Integer.class, String.class };
    private EntityTransaction emt;
    private EntityManager em;
...
```

A través de modoAlta se podrá saber a cada momento, cuántas filas tendrá la grilla, si la columna de código es editable o no, etc., veamos la implementación de los métodos de `AbstractTableModel`:

```
    public String getColumnName(int col) { return columnas[col]; }
    public int getRowCount() {return (!modoAlta) ? lista.size() : 1;}
    public int getColumnCount() { return columnas.length; }
    public Object getValueAt(int row, int col) {
        if ( !modoAlta && row > lista.size() ) return null;
        if ( modoAlta && row > 0 ) return null;
        List<Cliente> l = (!modoAlta) ? lista : listaAlta;
        switch(col) {
            case 0:
                 return l.get(row).getCodigo();
            case 1:
                 return l.get(row).getNombre();
            case 2:
                 return l.get(row).getDirec();
            case 3:
                 return l.get(row).getPostal();
            case 4:
                 return l.get(row).getTel();
        }
        return null;
    }
    public boolean isCellEditable(int row, int col){
        if (!modoAlta && (col == 0 || row > lista.size()))return false;
        if ( modoAlta && row > 0 ) return false;
        return true;
    }
    public void setValueAt(Object value, int row, int col) {
        if ( row > lista.size() ) return;
        if ( !modoAlta && col == 0 ) return;
        List<Cliente> l = (!modoAlta) ? lista : listaAlta;
        if ( !modoAlta ) emt.begin();
        switch(col) {
            case 0:
                 l.get(row).setCodigo((Integer) value);
                 break;
            case 1:
```

```
                    l.get(row).setNombre((String) value);
                    break;
               case 2:
                    l.get(row).setDirec((String) value);
                    break;
               case 3:
                    l.get(row).setPostal((Integer) value);
                    break;
               case 4:
                    l.get(row).setTel((String) value);
                    break;
          }
          if ( !modoAlta ) emt.commit();
          fireTableCellUpdated(row, col);
     }
     public Class getColumnClass(int c) { return clase_columnas[c];
}
```

En rojo pueden todo el codigo necesario para persistir el objeto `Cliente` que acaba de ser modificado o se pretende dar de alta!. En azul pueden ver el código necesario para indicarle a view JTable que debe volver a dibujar (refresh) la celda ubicada en la fila row y en la columna col.

Implementación Controller de ClienteModel

Todo el código controller se resume a la implementación del método `keyPressed()` de la interfase `KeyListener`:

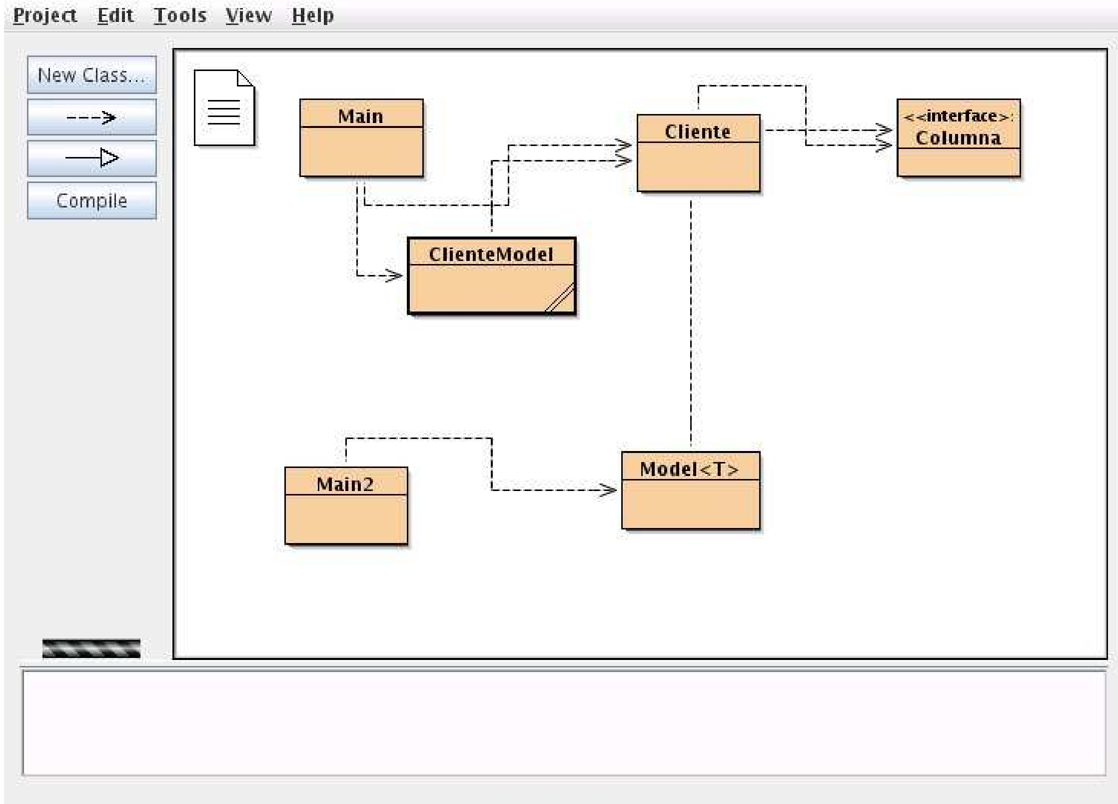
```
     public void keyPressed(KeyEvent e) {
          int rta,fila;
          switch(e.getKeyCode()) {
               case KeyEvent.VK_INSERT: // pulso Ins, hago alta
                  if ( !modoAlta ) {
                      rta = JOptionPane.showConfirmDialog(null,"¿Agrega
Nuevo Registro?"); // yes=0,no=1,cancel=2
                      if ( rta == 0 ) {
                            modoAlta=true;
                            if (listaAlta.size() > 0) listaAlta.remove(0);
                            listaAlta.add(new Cliente());
                            fireTableDataChanged();
                      }
                  } else {
                      rta = JOptionPane.showConfirmDialog(null,"¿Confirma
Nuevo Registro?");
                      switch(rta) {
                           case 0:
                                // persisto listaAlta, agrego listaAlta a
lista
                                try {
                                     emt.begin();
                                     em.persist(listaAlta.get(0));
                                     emt.commit();
                                     modoAlta=false;
                                     lista.add(listaAlta.get(0));
```

```
                            fireTableDataChanged();
                        } catch(RuntimeException re) {

System.err.println("ClienteModel:RuntimeException persistiendo
Cliente:"+re.getMessage());

JOptionPane.showMessageDialog(null,"Error(RT) Confirmando Alta
Cliente:\n"+re.getMessage(),"Error",JOptionPane.ERROR_MESSAGE);
                        } catch(Exception ex) {

System.err.println("ClienteModel:Exception persistiendo
Cliente:"+ex.getMessage());

JOptionPane.showMessageDialog(null,"Error(E) Confirmando Alta
Cliente:\n"+ex.getMessage(),"Error",JOptionPane.ERROR_MESSAGE);
                        }
                        break;
                    case 1:
                        break;
                    case 2:
                        modoAlta=false;
                        fireTableDataChanged();
                        break;
                }
            }
            break;
        case KeyEvent.VK_DELETE: // pulso Del, hago baja
            fila=tabla.getSelectedRow();
            if ( modoAlta || fila == -1 ) break;
            rta = JOptionPane.showConfirmDialog(null,"¿Borra el
Registro Seleccionado?");
            if ( rta == 0 ) {
                // remover jpa, remover lista
                try {
                    emt.begin();
                    em.remove(lista.get(fila));
                    emt.commit();
                    lista.remove(fila);
                    fireTableRowsDeleted(fila,fila);
                } catch(RuntimeException re) {

System.err.println("ClienteModel:RuntimeException borrando
Cliente:"+re.getMessage());
                    JOptionPane.showMessageDialog(null,"Error(RT)
Borrando
Cliente:\n"+re.getMessage(),"Error",JOptionPane.ERROR_MESSAGE);
                } catch(Exception ex) {
                    System.err.println("ClienteModel:Exception
borrando Cliente:"+ex.getMessage());
                    JOptionPane.showMessageDialog(null,"Error(E)
Borrando
Cliente:\n"+ex.getMessage(),"Error",JOptionPane.ERROR_MESSAGE);
                }
            }
            break;
    }
```

```
    }
```

En rojo nuevamente se puede ver todo el codigo para hacer altas y bajas usando JPA 2.0!, en azul la notificación de eventos.

Con esto lograrán una grilla para hacer altas, bajas, modificaciones sobre clientes utilizando JPA 2.0. El proyecto bluej tiene el siguiente aspecto:



Aunque del mismo, solo utilizamos la clase `Main`, `ClienteModel` y `Cliente`. Las restantes clases, las dejo para el próximo artículo. Deberán compilar el proyecto y generar en otro directorio los archivos `.jar` requeridos (tal como se indició en el artículo anterior), luego de ello, el código puede ejecutarse: `java -jar jpaqrytest.jar` (suponiendo que empaquetaron todo el proyecto en archivo `jpaqrytest.jar` el cual incluye a la clase `Main` como la clase ejecutable del proyecto y los restantes archivos `.jar` de JPA y jaybird se encuentran en el mismo directorio). La grilla contenida en una ventana debería verse como:

Codigo	Razon Social	Direccion	Codigo Postal	Telefono
1	que se yo	LOBOS	6700	421111
2	¡¡¡¡vue cambia...	mitre 2111	6700	1234566
3	ACOSTA LUIS	LUJAN	0	433329
4	ALCIDES	MALVINAS	0	02374-851835
5	AUTOMOTORE...	LUJAN	0	434718
6	ANGUERA C...	LUJAN	0	427792
8	BAVA JORGE	LUJAN	0	430402
9	BAVA ALBERTO	LUJAN	0	428787
10	BAVA TOTI	LUJAN	0	423061
11	BAVA EDUARD...	LUJAN	0	433452
12	BAUSADA	GRAL RODRIG...	0	0237-4840380
13	BUGIANESI	LUJAN	0	424209
14	BOLONI	MERCEDES	0	02324-424213
15	BIANCHI JUAN ...	LUJAN	0	422794
16	BALBO ROBER...	GRAL RODRIG...	0	0237-4840732
17	BOSCOVICH A...	S.A.ARECO	0	02326-452713
18	BRAIOTA HNOS	LUJAN	0	421143
19	BELLIDO	LOBOS	0	02227-423549
20	BERNARDO M...	G.RODRIGUEZ	0	0237-4842899
21	BERNARDO M...	G.RODRIGUEZ	0	0237-4842804
22	BLATER Y JULIA	LOBOS	0	02227-422524

Este codigo se adjunta y obviamente, requiere de algunas mejoras para una implementación profesional (por ejemplo, cuando el número de tuplas a recuperar de la base de datos es muy grande), además de esto, hay otra cuestión que veremos en el próximo artículo: este código sólo sirve para persistir objetos de tipo Cliente! (¿no les parece poco generico y reusable?).

Atte. Lic. Guillermo Cherencio