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

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

Para todo proyecto orientado a objetos que requiera algún tipo de persistencia de objetos, Ud. tiene las siguientes alternativas:

1) persistir datos en archivos

2) serialización de objetos

3) uso de bases de datos via jdbc

4) uso de entidades a ser persistidas utilizando JPA (Java Persistence API)

Para el punto 1) se debe utilizar el package java.io allí podemos encontrar clases para el uso de todo tipo de streams; así como también la clase RandomAccessFile que nos permitirá trabajar con archivos de acceso directo.

Para el punto 2) se debe contar con objetos que implementen la interfase Serializable y combinarlo con clases del package java.io, se grabarán en archivos todos los atributos no `transient` de los objetos serializables (por ejemplo, una contraseña, representada por un atributo de tipo String, por más que éste sea privado, al ser serializado, su contenido será visible por un editor de texto. Si el atributo se declara con el cualificador `transient` , ello indica que dicho atributo no será serializado en el archivo). Posteriormente se puede hacer el proceso inverso: recuperar (des-serializar) los objetos grabados previamente.

Tanto el punto 1) como el 2) tienen serias dificultades derivado de la arquitectura file - server, de la organización de archivos, control de concurrencia, seguridad, etc. cuestiones cuya resolución queda a cargo del programador .

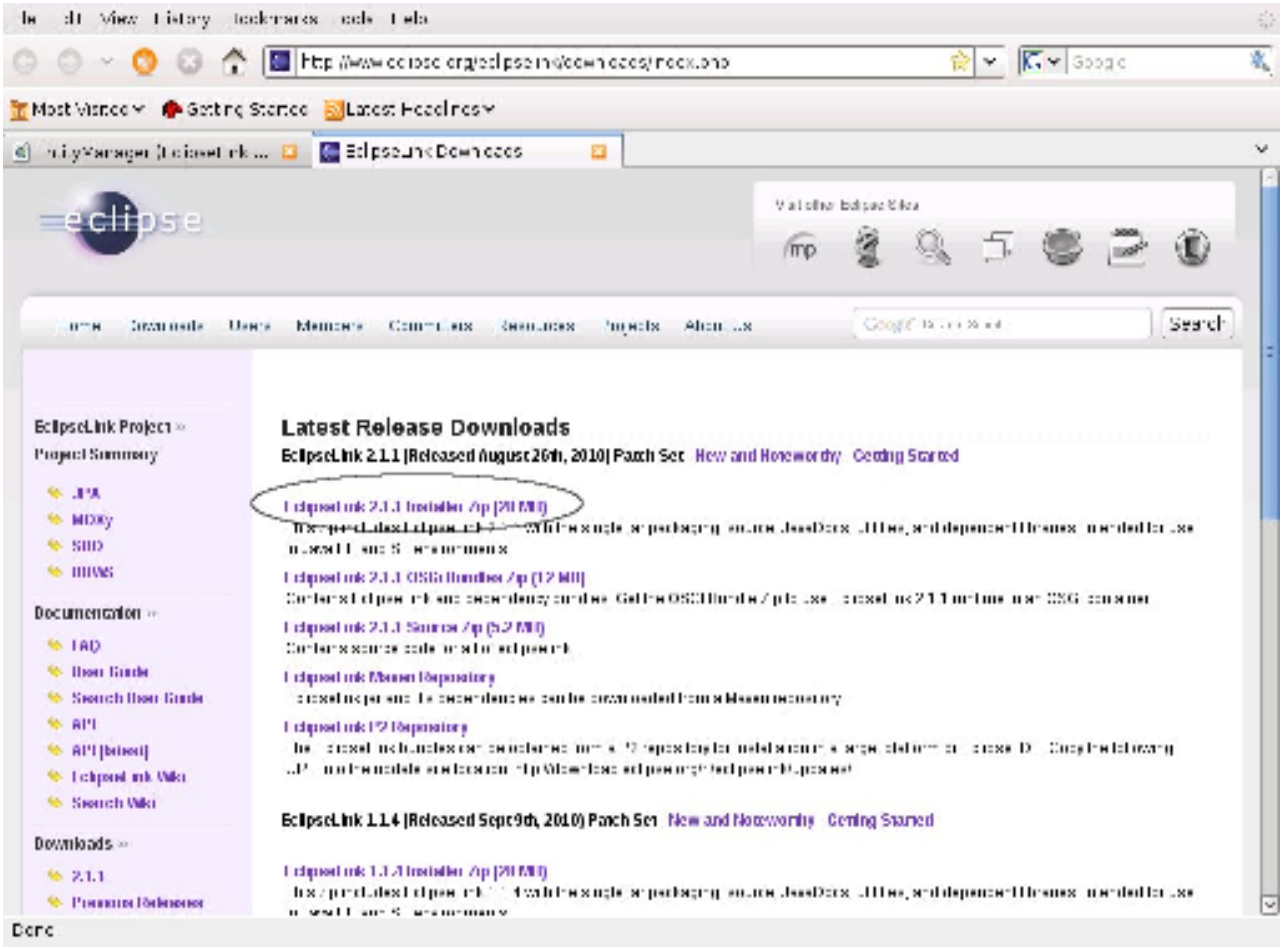
Para el punto 3) se debe contar con un driver jdbc para la base de datos a utilizar, en el caso de Firebird se puede utilizar Jaybird el cual puede descargarse desde www.firebirdsql.org :

básicamente se trata de un archivo .jar (ejemplo: jaybird-full-2.1.6.jar) que contiene todas las clases del driver jdbc más documentación, ejemplos, etc. Este archivo lo encontrará junto con otros dentro del archivo .zip que descargue, ya que el driver viene en distintas presentaciones. Puede tomar cualquiera de los ejemplos allí contenidos para establecer una conexión con la base de datos que desee, leerla y actualizarla. Al distribuir su aplicación, la misma también deberá incluir, por ejemplo, al archivo jaybird-full-2.1.6.jar. El punto 3) implica toda una superación de las dificultades indicadas para el punto 1) y 2), pero considere que además su aplicación requerirá de un Sistema Gestor de Bases de Datos (SGBD) previamente instalado y una base de datos en particular implementada para ser usada por su aplicación.

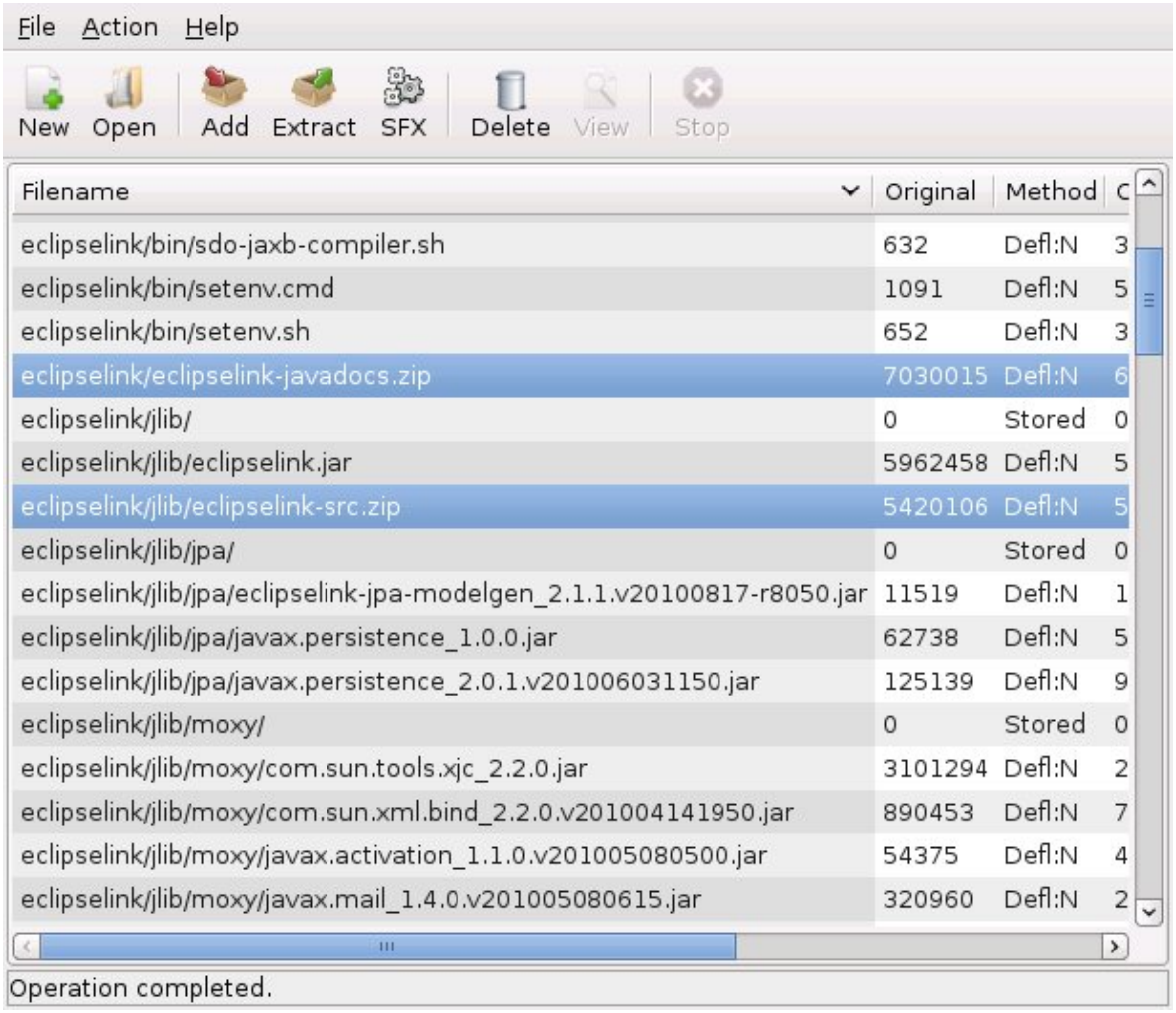
Para el punto 4) se requiere de un proveedor de persistencia que implemente la especificación correspondiente a JPA 2.0 –en este caso-, esto nos permite realizar de forma muy simple un mapeo objeto – relacional a través del uso de un driver jdbc como el del punto 3) hasta incluso sin necesidad de incrustar código sql dentro de nuestra aplicación (la generación de código sql estará delegada en el proveedor de persistencia y será transparente para nosotros). Permite configurar el driver y los parámetros de conexión en forma externa a nuestra aplicación. Hace uso intensivo de clases anotadas y utiliza el concepto de programación por excepción (programming by exception), es decir, que sólo debemos realizar anotaciones a modo de corregir el comportamiento por defecto que tiene JPA. Las entidades a persistir son clases POJO's (Plain Old Java Objects) –clases simples, que no derivan de otra, con constructor nulo, setters y getters públicos (setXXX,getXXX),

atributos (XXX) privados. Este tipo de clases conforman lo que en java se denominan “beans”.). Se pueden crear / recuperar instancias de clases anotadas con @javax.persistence.Entity que se usarán al interior de la aplicación como simples objetos, iguales a cualquier otro, permitiendo un fuerte desacoplamiento entre la aplicación y la base de datos (algo difícil de lograr optando por el punto 3)). Habitualmente, JPA se utiliza en desarrollos profesionales de aplicaciones distribuidas de mediana a gran complejidad utilizando Java Enterprise Edition (J2EE 6), esta implementación de java nos provee de un modelo de programación de n capas distribuidas, en donde la capa intermedia es un servidor de aplicaciones que implementa la especificación J2EE 6 como por ejemplo Glassfish versión 3. J2EE nos provee de distintos contenedores y nuestra aplicación se ejecutará bajo el control de un contenedor, es decir, que no son aplicaciones que se ejecutan directamente a través de la JRE desde la línea de comandos, sino que son descargadas en un contenedor (deployment) y ejecutadas por éste. En J2SE no hay contenedores y nuestra aplicación se ejecuta desde la línea de comandos utilizando la JRE, no obstante ello, es posible utilizar JPA, haciendo lo siguiente:

1) Bajar un driver jdbc para la base de datos a utilizar, en este caso, Firebird, por lo tanto, usaremos un driver denominado Jaybird que puede descargar desde www.firebirdsql.org como se describió anteriormente.
2) Bajar un proveedor de persistencia, existen varios, los más conocidos son TopLink (Oracle) y EclipseLink, en este caso, optamos por EclipseLink , esto puede hacerse desde la página www.eclipse.org/eclipselink/downloads/index.php , de allí bajar la primera opción que se indica, que nos permite la ejecución de JPA tanto en J2SE como en J2EE:



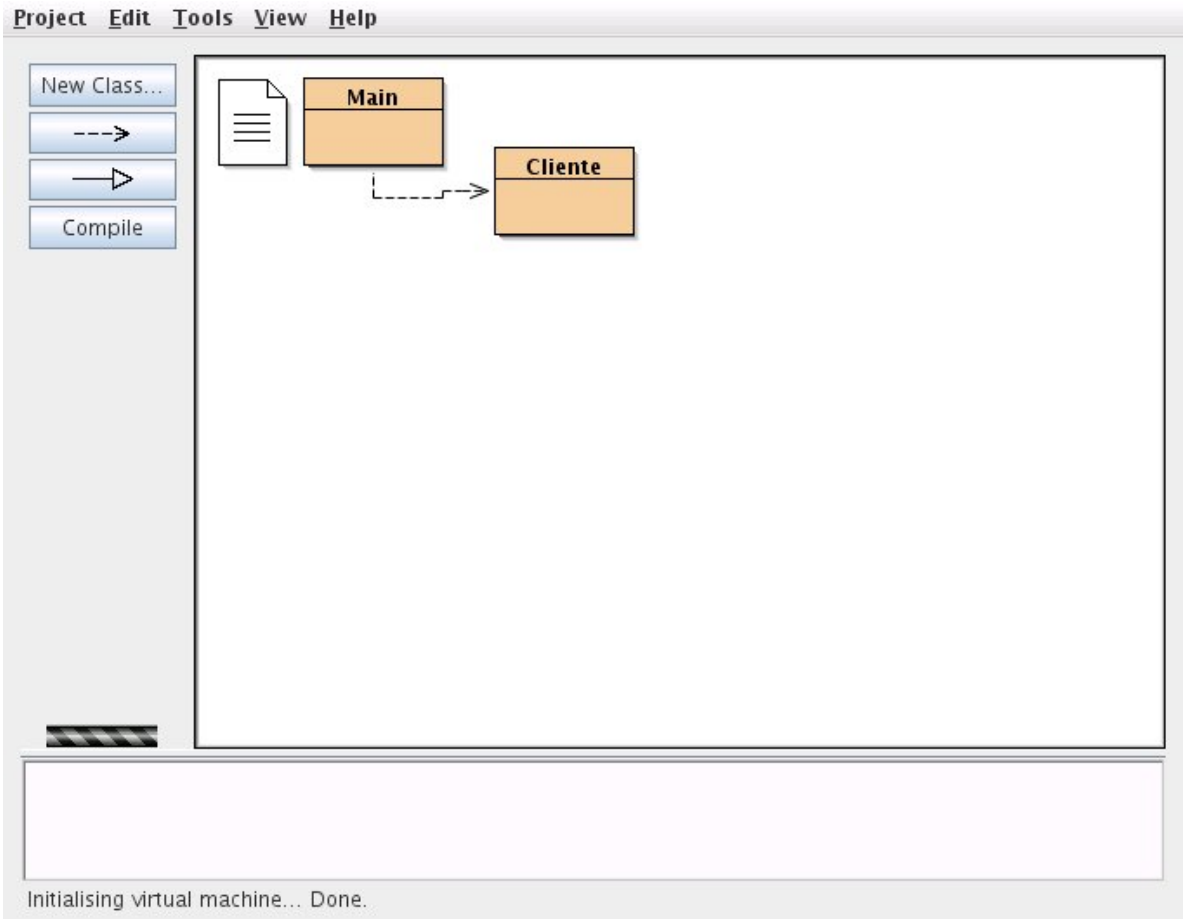
descomprima el archivo .zip descargado en alguna ubicación conocida, en mi caso utilice la ubicación /media/KINGSTON/grchere/java/eclipselink dentro del archivo encontrará otros dos archivos .zip:



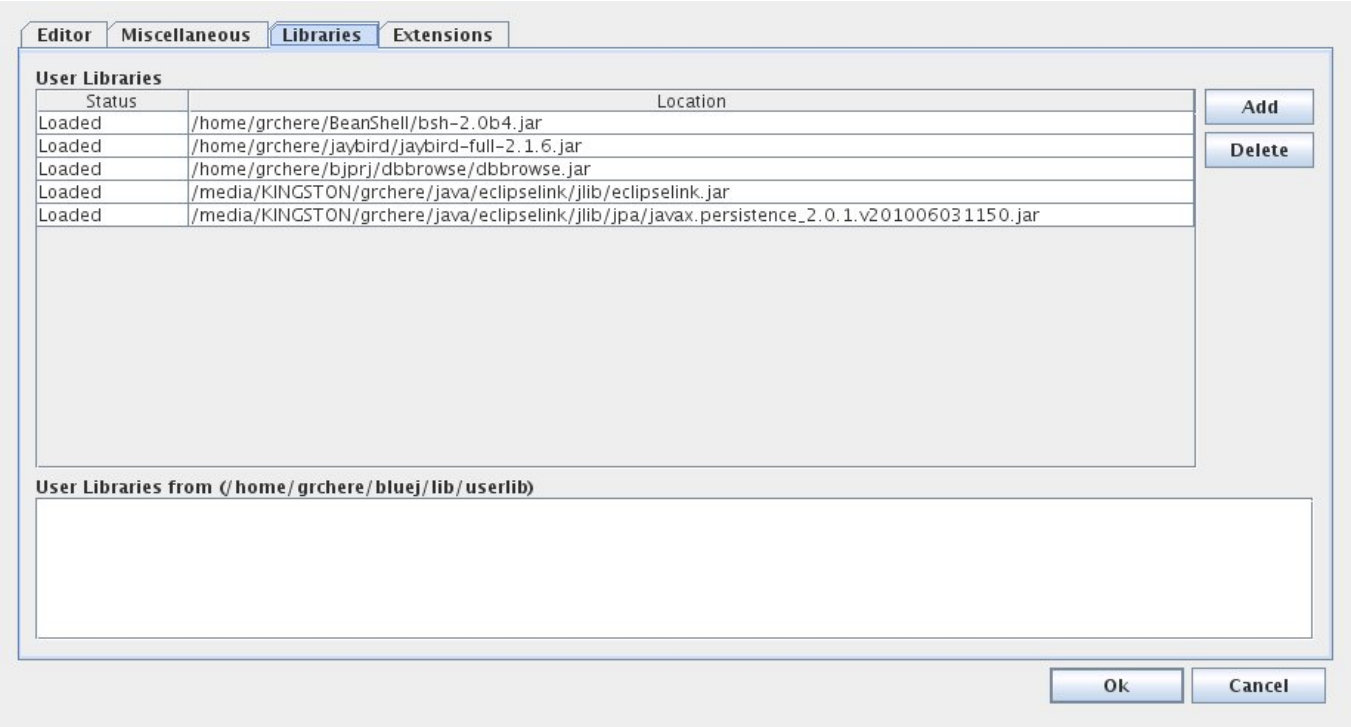
eclipselink-src.zip que contiene el código fuente correspondiente a esta implementación de JPA y eclipselink-javadocs.zip contiene toda la documentación de la API de JPA, lo cual es muy útil para el desarrolador, descomprima este archivo en una ubicación conocida y abra la pagina html principal de la documentación que le permitirá explorar –entre otros- el package javax.persistence , allí encontrará las interfases EntityManager , EntityManagerFactory ,etc. que serán de gran utilidad:

3) Crear una aplicación J2SE y crear –manualmente- instancias de un EntityManagerFactory, para luego crear una instancia de un EntityManager y con éste poder persistir las entidades que deseamos.

Para crear la aplicación utilicé un entorno muy simple de programación que se llama BlueJ www.bluej.org (teniendo el J2SE 6 SDK instalado, puede descargar la versión de BlueJ acorde con su plataforma y utilizarlo. Si Ud. utiliza NetBeans, observe la grilla de plug-ins, existe un plug-in para BlueJ que permite leer y grabar proyectos BlueJ desde NetBeans; puede instalarse este plug-in y tomar el proyecto jpa -que se adjunta con este documento - desde NetBeans), he creado un nuevo proyecto llamado “jpa”:



, luego en la opción Tools – Preferences… / Libraries se deben indicar los packages (.jars) correspondientes a Jaybird y EclipseLink que serán requeridos por este proyecto (se debe cerrar BlueJ y volver a entrar para que tome estos cambios):



Obsérvese que los archivos .jars requeridos por este proyecto están indicados en la segunda línea (jaybird-full-2.1.6.jar para Jaybird jdbc driver para bases de datos Firebird) y la cuarta y quinta línea (eclipselink.jar, javax.persistence_2.*.jar para EclipseLink JPA 2.0 provider).

Implementé sólo dos clases: la clase Main que lanza la aplicación java a través de su método estático main() y persiste una instancia de la entidad Cliente:

```
/**
 * Main class inicia aplicacion java que utiliza JPA en standard edition
 *
 * @author G.Cherencio
 * @version 1.0
 */
import javax.persistence.*;
public class Main {
   private static EntityManagerFactory emf = null;
   private static EntityManager em = null;

   public static void main(String arg[]) {
        System.out.println("Main:inicio");
        emf = Persistence.createEntityManagerFactory("tapas");
        System.out.println("Main:emf creado");
        em = emf.createEntityManager();
        System.out.println("Main:em creado");
        EntityTransaction emt = em.getTransaction();
        System.out.println("Main:emt creada");
        emt.begin();
        System.out.println("Main:emt.begin() hecho");
        Cliente c = new Cliente();
        c.setCodigo(1234);
```

```
        c.setNombre("alta desde JPA 2.0");
        c.setDirec("eclipseLink");
        c.setPostal(6700);
        String smsg = "persist()";
        try {
            em.persist(c);
            System.out.println("Main:em.persist(c) hecho");
            smsg = "commit()";
            emt.commit();
            System.out.println("Main:emt.commit() hecho");
        } catch(IllegalArgumentException iae) {
            System.out.println("Main:Error en "+smsg+ " persistiendo
cliente, posiblemente sea null");
        } catch(EntityExistsException eee) {
            System.out.println("Main:Error en "+smsg+" persistiendo
cliente, esta entidad ya existe");
        } catch(TransactionRequiredException tre) {
            System.out.println("Main:Error en "+smsg+" persistiendo
cliente, se requiere de una transaccion");
        } catch(Exception e) {
            System.out.println("Main:Error en "+smsg+ " persistiendo
cliente, error: "+e.getMessage());
        }
        salir();
        System.out.println("Main:fin");
    }
    public static EntityManagerFactory getEntityManagerFactory() { return
emf; }
    public static EntityManager getEntityManager() { return em; }
    public static void salir() {
        if ( em != null ) em.close();
        if ( emf != null ) emf.close();
    }
}
```

el trabajo principal lo realiza una instancia de tipo `EntityManager`, ésta se obtiene -en este caso, puesto que no hay contenedor J2EE que nos lo provea- a través del método `createEntityManagerFactory`() al cual le pasamos el nombre de la unidad de persistencia (`tapas`, en este caso) que luego deberá ser configurada en el archivo `persistence.xml`. Una vez obtenida una instancia de tipo `EntityManager`, creamos una transacción, la iniciamos con `begin`() y luego la damos por terminada con `commit`(), a través del método `persists`() podemos persistir cualquier tipo de instancia que sea una entidad (es decir una instancia de la una clase POJO anotada con @`Entity`), en este caso, algo de tipo Cliente. Todo el código es muy simple, la llamada a `persist`() y `commit`() se encerró en un bloque `try` - `catch` para controlar los posibles errores que se produzcan en la persistencia. Veamos ahora el código correspondiente a la entidad Cliente:

```
/**
 * Entidad Cliente que pretendo persistir en tabla trcliente
 * de la base de datos firebird 2.0 /var/lib/firebird/2.0/data/tapasv.gdb
 * persisance.xml
```

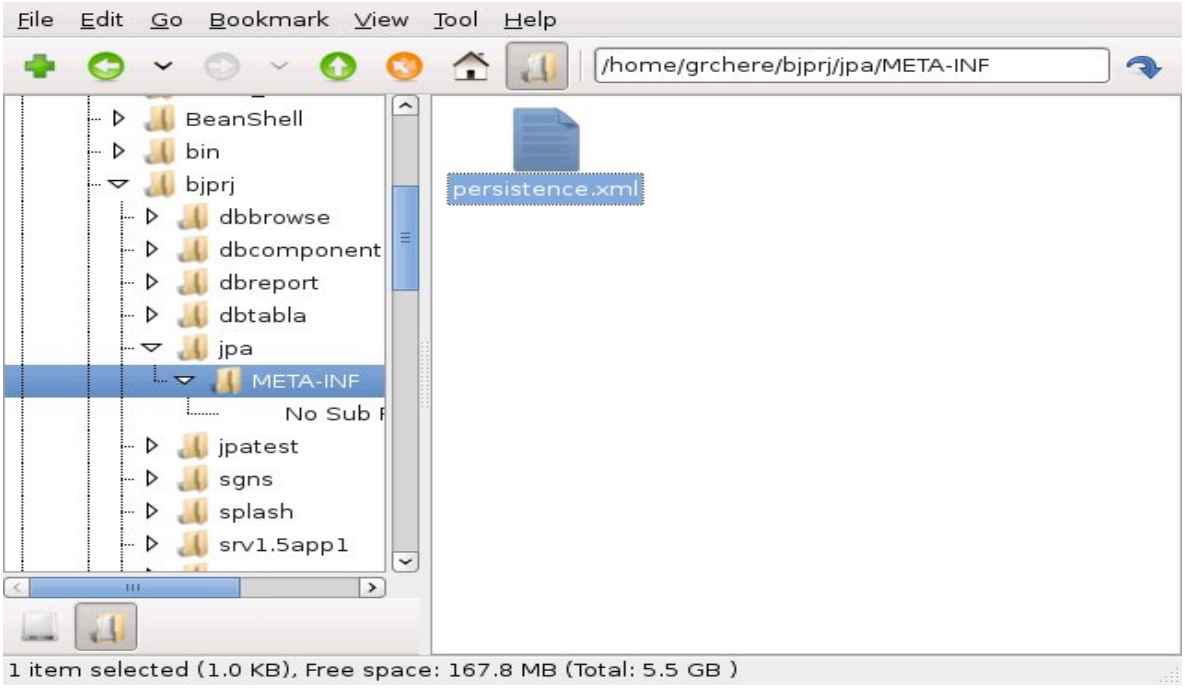
```
 *    debe estar configurado para usar driver jdbc jaybird
 *    debe estar configurado para usar proveedor de persistencia
EclipseLink
 *    debe estar dentro de la carpeta META-INF del archivo jar a ejecutar
 *
 * @author G.Cherencio
 * @version 1.0
 */
import javax.persistence.*;
@Entity
@Table(name="trcliente")
public class Cliente
{
    // instance variables - replace the example below with your own
    @Id
    @Column(nullable=false)
    private int codigo;
    @Column(nullable=false,length=40)
    private String nombre;
    @Column(nullable=true,length=30)
    private String direc;
    @Column(nullable=true)
    private int postal;
    @Column(nullable=true,length=20)
    private String tel;

    /**
     * Constructor for objects of class Cliente
     */
    public Cliente()
    {
        // initialise instance variables
    }

//  SETTERS
    public void setCodigo(int c) { codigo=c; }
    public void setNombre(String s) { nombre=s; }
    public void setDirec(String s) { direc=s; }
    public void setPostal(int p) { postal=p; }
    public void setTel(String t) { tel=t; }
//  GETTERS
    public int getCodigo() { return codigo; }
    public String getNombre() { return nombre; }
    public String getDirec() { return direc; }
    public int getPostal() { return postal; }
    public String getTel() { return tel; }
}
```

Obsérvese cómo sólo se indica la excepción, por ejemplo, si no indicara el largo de un atributo, JPA asumiría, para el caso de `private String nombre;` un `varchar(255)` en la base de datos, pero como ello no coincide con la estructura que este caso tiene la tabla `trcliente` de la base de datos firebird `tapasv.gdb` que se encuentra en la carpeta `/var/lib/firebird/2.0/data` con la cual esta vinculada la entidad Cliente:

```
grchere@debian:~/bjprj/jpatest$ isql-fb -u sysdba -p masterkey
/var/lib/firebird/2.0/data/tapasv.gdb
Database:  /var/lib/firebird/2.0/data/tapasv.gdb, User: sysdba
SQL> show table trcliente;
CODIGO                          INTEGER Not Null
NOMBRE                          VARCHAR(40) Not Null
DIREC                           VARCHAR(30) Nullable
POSTAL                          INTEGER Nullable
TEL                             VARCHAR(20) Nullable
CONSTRAINT INTEG_5:
  Primary key (CODIGO)
SQL>
```

debo indicar esta excepción a la regla JPA (programming by exception). Otro ejemplo puede ser la anotación `@Table` en donde indico el nombre de la tabla a usar, caso contrario, JPA persistirá las instancias de Cliente en la tabla Cliente, que en esta base de datos no existe. Todas las anotaciones (@...) anteceden al elemento que pretenden anotar, para indicarle algo, en este caso, a JPA, cómo debe tratar a este elemento. El mecanismo de anotaciones es sumamente potente y útil, en especial en J2EE 6, pero es bien aplicable en J2SE. Las anotaciones me han cautivado, pues es un mecanismo genial que ha extendido el imperio de la `METADATA` hasta el código propio de la aplicación y no sólo es un concepto aplicable a archivos y bases de datos.

Por fuera de BlueJ, dentro del directorio `jpa`, he creado la carpeta `META-INF` y dentro de la misma el archivo `persistence.xml`:

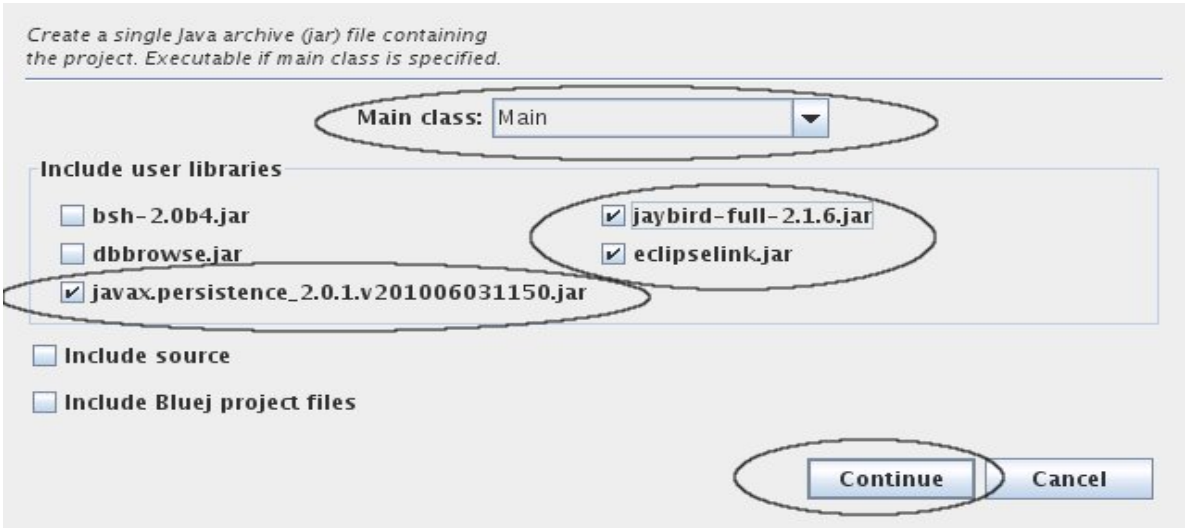


dentro de `persistence.xml` debemos configurar JPA para que utilice el driver jaybird y pueda conectarse con la base de datos firebird `tapasv.gdb`. Veamos el contenido de este archivo xml:

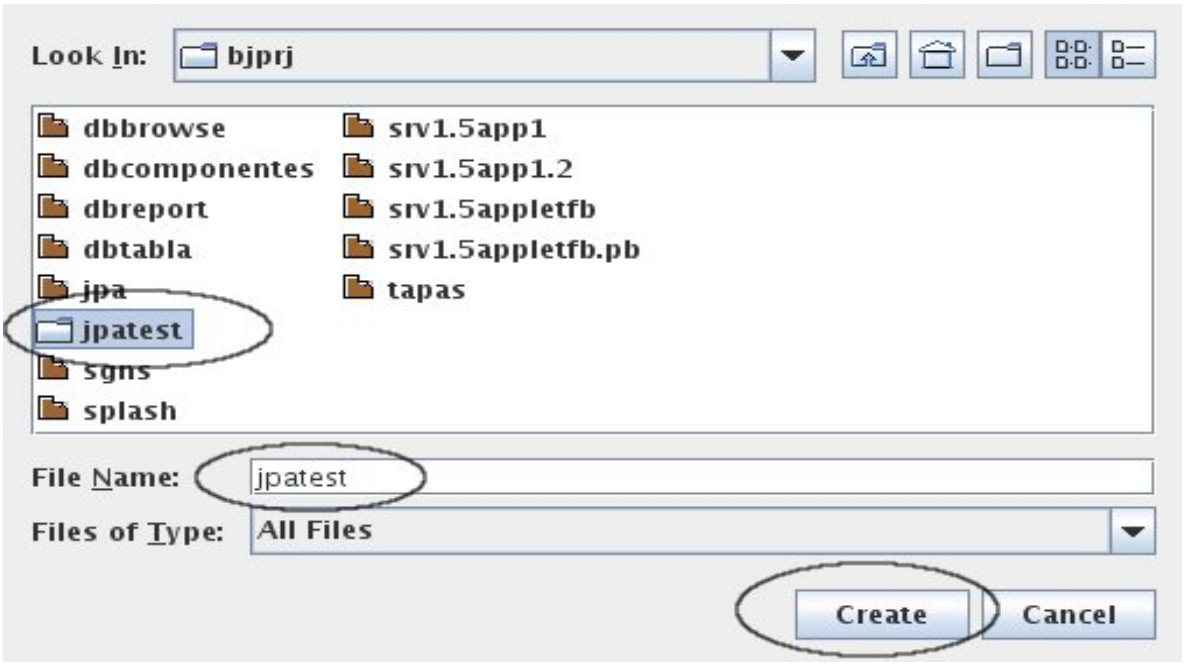
```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
persistence_1_0.xsd"
        version="1.0">
    <persistence-unit name="tapas" transaction-type="RESOURCE_LOCAL">

<provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
        <exclude-unlisted-classes>false</exclude-unlisted-classes>
        <properties>
            <property name="eclipselink.logging.level" value="INFO"/>
            <property name="javax.persistence.jdbc.driver"
value="org.firebirdsql.jdbc.FBDriver"/>
            <property name="javax.persistence.jdbc.url"
value="jdbc:firebirdsql:localhost/3050:/var/lib/firebird/2.0/data/tapasv.
gdb"/>
            <property name="javax.persistence.jdbc.user" value="sysdba"/>
            <property name="javax.persistence.jdbc.password"
value="masterkey"/>
        </properties>
    </persistence-unit>
</persistence>
```

Obsérvese que aquí se configura –por fuera de la aplicación- que tipo de proveedor de persistencia que vamos a utilizar (EclipseLink), el driver jdbc, base de datos (Jaybird), servidor (localhost), usuario, contraseña, etc. (esto también puede hacerse desde el interior de la aplicación si desea ocultar estos datos).

Una vez terminado el código en BlueJ, se compila el proyecto y se utiliza la opción Project – Create Jar File … y se empaqueta la aplicación en un archivo `.jar` de la siguiente forma:



Se indican los archivos `.jar` requeridos por la aplicación. El archivo `.jar` de la aplicación se crea dentro de la carpeta `jpatest` –en este caso- :



Una vez empaquetada la aplicación, ésta se encuentra lista para su distribución y ejecución. En esta carpeta se encuentra el archivo `.jar` de nuestra aplicación (`jpatest.jar`) más los `.jars` previamente seleccionados; éstos fueron copiados por BlueJ para facilitar la distribución de nuestra aplicación:

```
grchere@debian:~/bjprj/jpatest$ ls
eclipselink.jar                      jaybird-full-2.1.6.jar
javax.persistence_2.0.1.v201006031150.jar  jpatest.jar
grchere@debian:~/bjprj/jpatest$
```

La aplicación esta lista para su distribución y ejecución, pero ella requiere contar con el sistema gestor de base de datos (SGBD) Firebird instalado y la base de datos `tapas.gdb`, en mi caso, firebird 2.0 bajo Debian, para arrancar este servidor debemos hacer:

```
$ sudo /etc/init.d/firebird2.0-super start
```

Ahora, nos paramos en la carpeta `jpatest` y comenzamos a probar la aplicación.

Verifico si el objeto existe en la base de datos:

```
grchere@debian:~/bjprj/jpatest$ isql-fb -u sysdba -p masterkey
/var/lib/firebird/2.0/data/tapasv.gdb
Database:  /var/lib/firebird/2.0/data/tapasv.gdb, User: sysdba
SQL> show table trcliente;
CODIGO                          INTEGER Not Null
NOMBRE                          VARCHAR(40) Not Null
DIREC                           VARCHAR(30) Nullable
POSTAL                          INTEGER Nullable
TEL                             VARCHAR(20) Nullable
CONSTRAINT INTEG_5:
  Primary key (CODIGO)
SQL>
SQL> select * from trcliente where codigo = 1234;
SQL> exit;
grchere@debian:~/bjprj/jpatest$
```

El cliente `1234` no existe en la tabla `trcliente`. Ejecuto la aplicación:

```
grchere@debian:~/bjprj/jpatest$ java -jar jpatest.jar
Main:inicio
Main:emf creado
[EL Info]: 2010-09-09 23:53:49.637--ServerSession(5938662)--
EclipseLink, version: Eclipse Persistence Services -
2.1.1.v20100817-r8050
[EL Info]: 2010-09-09 23:53:51.684--ServerSession(5938662)--
file:/home/grchere/bjprj/jpatest/jpatest.jar_tapas login
successful
Main:em creado
Main:emt creada
Main:emt.begin() hecho
Main:em.persist(c) hecho
Main:emt.commit() hecho
[EL Info]: 2010-09-09 23:53:53.618--ServerSession(5938662)--
file:/home/grchere/bjprj/jpatest/jpatest.jar_tapas logout
successful
Main:fin
grchere@debian:~/bjprj/jpatest$
```

Verifico si el objeto `cliente` fue persistido en la base de datos:

```
grchere@debian:~/bjprj/jpatest$ isql-fb -u sysdba -p masterkey
/var/lib/firebird/2.0/data/tapasv.gdb
Database:  /var/lib/firebird/2.0/data/tapasv.gdb, User: sysdba
SQL> select * from trcliente where codigo = 1234;

      CODIGO NOMBRE                                     DIREC
POSTAL TEL
============ ========================================
============================== ============ ====================
        1234 alta desde JPA 2.0                        eclipseLink
6700 <null>

SQL> exit;
grchere@debian:~/bjprj/jpatest$
```

si el objeto cliente ya existe en la base de datos y vuelvo a ejecutar la aplicación:

```
grchere@debian:~/bjprj/jpatest$ java -jar jpatest.jar
Main:inicio
Main:emf creado
[EL Info]: 2010-09-09 23:50:14.126--ServerSession(20228056)--
EclipseLink, version: Eclipse Persistence Services -
2.1.1.v20100817-r8050
[EL Info]: 2010-09-09 23:50:16.264--ServerSession(20228056)--
file:/home/grchere/bjprj/jpatest/jpatest.jar_tapas login
successful
Main:em creado
Main:emt creada
Main:emt.begin() hecho
Main:em.persist(c) hecho
[EL Warning]: 2010-09-09 23:50:18.482--UnitOfWork(25919971)--
Exception [EclipseLink-4002] (Eclipse Persistence Services -
2.1.1.v20100817-r8050):
org.eclipse.persistence.exceptions.DatabaseException
Internal Exception: org.firebirdsql.jdbc.FBSQLException: GDS
Exception. 335544665. violation of PRIMARY or UNIQUE KEY
constraint "INTEG_5" on table "TRCLIENTE"
Error Code: 335544665
Call: INSERT INTO trcliente (CODIGO, NOMBRE, POSTAL, DIREC, TEL)
VALUES (?, ?, ?, ?, ?)
     bind => [1234, alta desde JPA 2.0, 6700, eclipseLink, null]
Query: InsertObjectQuery(Cliente@145c859)
Main:Error en commit() persistiendo cliente, error: Exception
[EclipseLink-4002] (Eclipse Persistence Services -
2.1.1.v20100817-r8050):
org.eclipse.persistence.exceptions.DatabaseException
```

```
Internal Exception: org.firebirdsql.jdbc.FBSQLException: GDS
Exception. 335544665. violation of PRIMARY or UNIQUE KEY
constraint "INTEG_5" on table "TRCLIENTE"
Error Code: 335544665
Call: INSERT INTO trcliente (CODIGO, NOMBRE, POSTAL, DIREC, TEL)
VALUES (?, ?, ?, ?, ?)
     bind => [1234, alta desde JPA 2.0, 6700, eclipseLink, null]
Query: InsertObjectQuery(Cliente@145c859)
[EL Info]: 2010-09-09 23:50:18.586--ServerSession(20228056)--
file:/home/grchere/bjprj/jpatest/jpatest.jar_tapas logout
successful
Main:fin
grchere@debian:~/bjprj/jpatest$
```

Se genera una excepción y ésta es capturada por la aplicación, contando con todos los datos que nos provee Firebird para determinar el tipo de error ocurrido. Obsérvese que se genera automáticamente el código sql necesario sin que nosotros lo hayamos tipeado en el código de la aplicación.

Podemos usar el cliente `isql-fb` (en el caso de Linux, en Windows es `isql.exe`) de Firebird para ingresar en la base de datos y borrar el cliente persistido desde la aplicación:

```
grchere@debian:~/bjprj/jpatest$ isql-fb -u sysdba -p masterkey
/var/lib/firebird/2.0/data/tapasv.gdb
Database:  /var/lib/firebird/2.0/data/tapasv.gdb, User: sysdba
SQL> delete from trcliente where codigo = 1234;
SQL> commit;
SQL> exit;
```

Ahora vuelvo a ejecutar la aplicación:

```
grchere@debian:~/bjprj/jpatest$ java -jar jpatest.jar
Main:inicio
Main:emf creado
[EL Info]: 2010-09-09 23:53:49.637--ServerSession(5938662)--
EclipseLink, version: Eclipse Persistence Services -
2.1.1.v20100817-r8050
[EL Info]: 2010-09-09 23:53:51.684--ServerSession(5938662)--
file:/home/grchere/bjprj/jpatest/jpatest.jar_tapas login
successful
Main:em creado
Main:emt creada
Main:emt.begin() hecho
Main:em.persist(c) hecho
Main:emt.commit() hecho
[EL Info]: 2010-09-09 23:53:53.618--ServerSession(5938662)--
file:/home/grchere/bjprj/jpatest/jpatest.jar_tapas logout
successful
```

```
Main:fin
grchere@debian:~/bjprj/jpatest$
```

Podemos observar que la aplicación se ejecuta como la primera vez, cuando el objeto persistido no existía en la base de datos.

Esta es una muy simple introducción a JPA 2.0 usado desde J2SE 6, el articulo tiene por objeto que Ud. pueda acceder a esta tecnología sin necesidad de usar J2EE 6, escribiendo, por ejemplo, una aplicación de consola o swing que interactúe con una base de datos firebird usando JPA. Esta tecnología nos permite hacer todo tipo de I/O sobre cualquier base de datos y distintos mapeos que puedan involucrar a relaciones entre objetos que se traducirán a una forma relacional, incluso consultarla (por ejemplo, usando el método .find() de EntityManager o usando sql o usando JPQL), obtener colecciones de objetos que luego puedan ser mostradas en una grilla, modificarlos y volver a persistirlos (método .merge() de EntityManager), etc.

Atte. Lic. Guillermo Cherencio