Embedded databases 3
Embedded Firebird

Michaël Van Canneyt

January 28, 2006

Abstract

In this third article about embedded database, the Embedded Firebird is tested. It will be shown how to use set up Embedded firebird, and how to configure some database access layers (for Delphi and Lazarus) to work with Embedded Firebird.

1 Introduction

The Firebird SQL server (currently at version 1.5.3) was started from the open source version of Interbase (6.0). It has remained compatible to a large part to Interbase, but has put other accents than the Interbase Developers. One of these differences is that Firebird offers an embedded build of the Firebird server. This means essentially that the Firebird Client Library and the Firebird Server have been built into a single library (fbembed), which can be used directly in an application.

No separately running Firebird server is needed to use the embedded server; The sole restriction is that a database can be accessed by only 1 application at a time. Other than that, all functionality of the normal server version is available. This makes Firebird an excellent choice as a back-end for a small application which nevertheless needs a database. At any time, the application can be upscaled to a real client-server application. No code changes will be necessary other than perhaps specifying the new location of the database.

In the next sections, the setup of embedded firebird will be discussed. After that, it will be explained how to configure some existing Delphi and Lazarus data-access technologies so they use the embedded version of firebird. Finally, the pupil tracker application introduced in the previous articles will be tested with Firebird, and the performance will be compared with SQLite.

2 Installation on Windows

The windows version of Embedded firebird comes as a separate zip file, which contains installation instructions. Normally, it is sufficient to unzip the contents of this file in the directory of the application which will use the embedded Firebird engine. Strictly speaking, only the following files are needed:

```
fbembed.dll
ib_util.dll
firebird.conf
firebird.msg
intl\fbintl.dll
```

```
udf\fbudf.dll
```

If the data access layer of the application needs to have the traditional name of the Interbase/Firebird client library, then the file fbembed.dll can be renamed to fbclient.dll or gds32.dll.

The fbintl.dll is only needed when the application needs to support international character sets. Similarly, the `fbudf.dll` is only needed when the application needs to access the standard UDF libraries.

In principle it is possible to split out the application files and the embedded firebird files, putting the embedded firebird files in a separate directory:

```
c:\embeddedfb\firebird.msg
c:\embeddedfb\intl\fbintl.dll
c:\embeddedfb\udf\fbudf.dll
```

In that case, the firebird.conf file located next to the application needs to have an entry

```
RootDirectory=c:\embeddedfb\
```

This allows to share the embedded firebird files between applications.

Note that the security database is not used. This does not mean that the database can be used as any user: SQL privileges are still checked ! It just means that passwords are not checked. But if a user has no SELECT rights on a table, then this restriction will still be enforced by the embedded server.

3 Installation on Linux

On linux the installation is similar to that on Windows, but is nevertheless slightly different. There is no separate archive with the embedded version. Instead, the embedded version is distributed together with the classic build of the Linux Firebird server.

The classic build contains the libfbembed.so library, which contains the embedded server. The linux version does need the security database security.fdb, and it additionally needs the lock manager. That means that the following files should be distributed:

```
lib/libfbembed.so
firebird.conf
security.fdb
intl/fbintl
udf/fbudf.so
bin/fb_lock_mgr
```

As can be seen from the list above, the lock manager also needs to be distributed. This is because the embedded version is derived from the classic build of the Firebird server.

Additionally, 2 environment variables need to be set when the application is started. Supposing the application is installed in /opt/myapp, then the following variables should be set:

```
LD_LIBRARY_PATH=/opt/myapp/lib
FIREBIRD=/opt/myapp
```

The first variable allows the dynamic library loader to find the libfbembed.so library. The second variable tells the embedded server where it can find it's message file, configuration file etc. It's best to have an entry in firebird.conf which points to the application directory:

```
RootDirectory=/opt/myapp
```

Similar to the Windows version it's possible to share an embedded server between various applications by moving the above files to a directory of their own. If the 2 variables mentioned above here point to this directory then everything should be working just fine.

Contrary to the Windows version, the security database file *is* needed and passwords *are* checked.

4 Configuring DBExpress

The DBExpress components from Delphi offer a fast and simple way to connect to many databases, including Interbase or Firebird. At the same time all advantages of using a `TClientDataset` are available, making these components a good choice for any database application.

The connection component `TSQLConnection` represents a connection to a Database. The details of using this component will not be discussed here: mainly this means setting the path to the database, and specifying the username and password. Note that specifying a username and password is needed: the password will not be checked, but the username is needed to determine the SQL permissions.

So far there is no difference with using this component for use with a regular firebird or Interbase database. However, to use the `TSQLConnection` with embedded firebird, it's sufficient to set the `vendorlib` property to fbembed.dll. The DBExpress driver will then load fbembed.dll when it establishes a connection to the database.

In the previous 'Embedded Databases' article, a pupil track data browser application was shown. A version of this application that uses embedded firebird can be found on the CD accompagnying this issue. The source code will not be examined, as it's very straightforward and simple.

5 Configuring IBExpress (IBX)

Delphi's IBExpress components, which connect straight to the Interbase client library, can also be configured to use the embedded firebird engine. This is slightly more involved than for DBExpress.

The set of IBX components have a unit called IBIntf. This unit contains the binding to the Interbase client library. It exposes an interface varIGDSLibrary defined as:

```
IGDSLibrary = interface
  ['{BCAC76DD-25EB-4261-84FE-0CB3310435E2}']
  procedure LoadIBLibrary;
  procedure FreeIBLibrary;
 ...
end;
```

Not all methods are shown. Each `TIBDatabase`, `TIBTransaction`, `TIBQuery` component uses a reference to this interface to execute calls. This reference is obtained using the `GetGDSLibrary` call in the IBIntf unit.

The actual interface returned by the `GetGDSLibrary` call can be configured using the following call:

```
Type
  RegisterGDSLibrary = function : IGDSLibrary;
Procedure RegisterGDSLibraryFactory(ARegisterGDSLibrary :
                                    TRegisterGDSLibrary);
```

The `ARegisterGDSLibrary` callback will be used when `GetGDSLibrary` is called. It should return an instance of a class which implements the `IGDSLibrary` interface.

The unit IBXembedded implements a callback which loads a configurable Firebird or Interbase client library, and constructs an `IGDSLibrary` interface with the entry points found in the library.

It exposes the following calls:

```
Var
{$ifndef linux}
  CustomIBXlibraryName : String = 'fbembed.dll';
{$else}
  CustomIBXlibraryName : String = 'fbembed.so';
{$endif}

Procedure UseEmbeddedLibrary(LibraryName : String);
```

When this unit is used, it will register a callback with `RegisterGDSLibraryFactory`. By default, this callback will load the library named in `CustomIBXLibraryname`, and use that for all connections to Firebird databases.

By issuing the `UseEmbeddedLibrary` call in the initialization section of a unit, the name of the library to be used can be changed. For instance, using the call

```
Initialization
  UseEmbeddedLibrary('c:\myapp\fbembed.dll');
end.
```

One can force the use of the fbembed.dll library distributed alongside the application.

Note that it is not possible to force IBX to use a different library for different connections. Only one library can be used. This need not be a problem, because the embedded library is a full-fledged client library, and can hence be used to make connections to remote servers.

6 Configuring SQLDB

The `SQLDB` components that come with Free Pascal (or Lazarus) have been designed similar to Delphi's DBExpress technology. There is a `TIBConnection` component to connect to Interbase or Firebird databases. This component makes use of the `ibase60dyn` unit, distributed with Free Pascal: this unit does run-time loading of the Interbase/Firebird client library. (The unit `ibase60` links to the client library at compile-time).

The `ibase60dyn` unit has the following declarations in it:

```
Var
  UseEmbeddedFirebird : Boolean = False;

const
{$IFDEF Win32}
   fbembedlib = 'fbembed.dll';
```

```
{$ELSE}
   fbembedlib = 'libfbembed.so';
{$endif}
```

If `UseEmbeddedFirebird` is set to `True` then the unit will try to load the embedded firebird client library, instead of the normal client library. So to use the embedded firebird library, it is sufficient to add the following code to an initialization section of a unit:

```
Initialization
  UseEmbeddedFirebird:=True;
end.
```

After that, all connections will be made using the embedded firebird client library (as specified in fbembedlib).

Note that, similarly to IBExpress, only one client library can be used for all connections in the application.

7 SQL executor

The SQL executor program introduced in the previous articles required only three functions to be able work with any database engine:

```
procedure StartDatabase;
procedure StopDatabase;
procedure ExecuteStatement(SQL: String;
                           IgnoreError: Boolean);
```

These functions can easily be implemented to work with Firebird. All that is needed is a `TIBConnection` component, a `TSQLTransaction` and `TSQLQuery` query. They act together like the corresponding components from IBX: A `TSQLQuery` component is connected to a `TIBConnection` component (called `DBFB`) and a `TSQLTransaction` component (called `TRFB`).

The functions to start and stop the database are then simple:

```
procedure TMainForm.StartDatabase;

begin
  DBFB.Connected:=True;
  TRFB.StartTransaction;
end;

procedure TMainForm.StopDatabase;

begin
  TRFB.Commit;
  DBFB.Connected:=False;
end;
```

The `TSQLQuery` component (called `FBQry`) works like any `TQuery` or `TIBQuery` would work: It has a `ExecSQL` method to execute the SQL statement in it's `SQL` property. This makes it easy to implement the `ExecuteStatement` function:

```
procedure TMainForm.ExecuteStatement(SQL : String;
                                     IgnoreError : Boolean);

begin
  Try
    FBQry.SQL.Text:=SQL;
    FBQry.ExecSQL;
  Except
    On E: Exception do
      begin
      MLog.Lines.add('Error executing statement: '+E.Message);
      If Not IgnoreError then
        Raise;
      end;
  end;
end;
```

This will make the program work with the default Firebird client library. To make sure that it works with the embedded version of Firebird, the following call is added to the initializaion section of the main form unit:

```
initialization
  {$I frmmain.lrs}
  useEmbeddedFirebird:=True;
end.
```

8 Pupil Track Data browser

The pupil tracking data browser application can be adapted similarly to the use of SQLDB components . This time, not one query component is needed, but 2: `QPupils` and `QPupilTrackData`. They are both connected to a `DBTracker` connection component and a `TRTracker` transaction component.

The functions that need to be implemented are similar to the ones in the SQL executor:

```
function DatabaseOpen: Boolean;
procedure CloseDatabase;
procedure OpenDatabase;
```

Their implementation is straightforward:

```
function TMainForm.DatabaseOpen: Boolean;
begin
  Result:=DTracker.Connected;
end;

procedure TMainForm.CloseDatabase;
begin
  QPupilTrackdata.Close;
  QPupils.Close;
  DTracker.Connected:=False;
end;
```

```
procedure TMainForm.OpenDatabase;
begin
  DTracker.Connected:=True;
  QPupils.Open;
end;
```

SQLDB currently does not yet have support for Master-Detail coupling of queries (although this is planned for the near future). To make the `QPupilTrackData` query display the data for the current pupil, the following handler is implemented for the `AfterScroll` event of the `QPupils` query:

```
procedure TMainForm.QPupilsAfterScroll(DataSet: TDataSet);

Var
  ID : Integer;

begin
  With QPupilTrackData do
    begin
    Close;
    If Not Dataset.fieldByName('PU_ID').IsNull then
      begin
      ID:=Dataset.fieldByName('PU_ID').AsInteger;
      Params.ParamByNAme('PU_ID').asInteger:=ID;
      Open;
      end;
    end;
end;
```

Again, to make the browser application use the embedded version of Firebird, the initialization code of the mail form unit is changed in the same way as the SQL Executor program. The end result can be seen in figure 1 on page 8

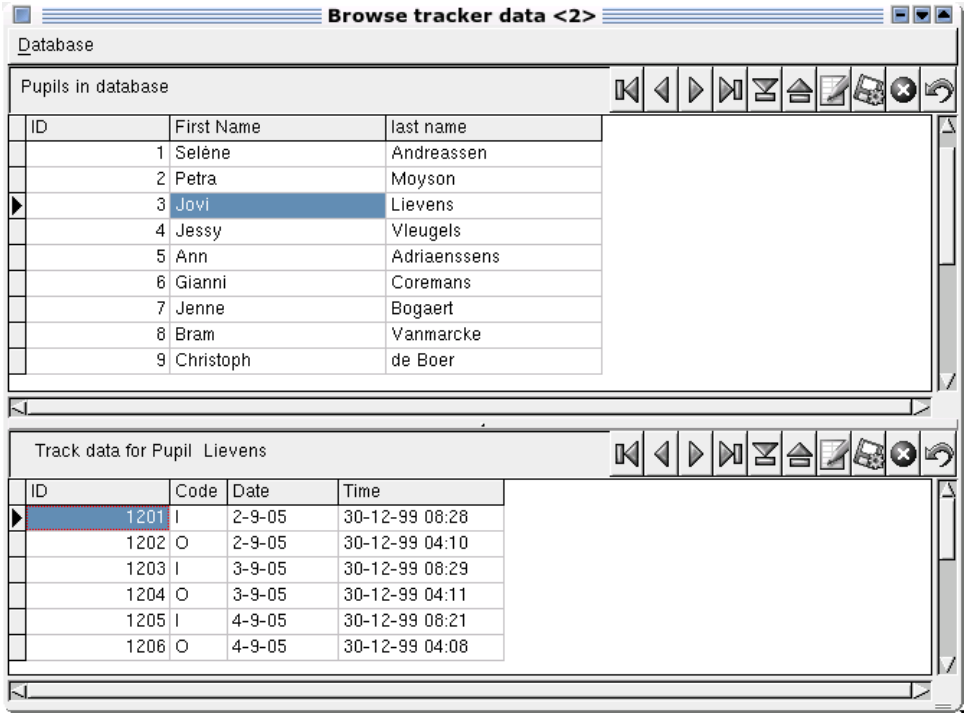
9 Performance

Exactly the same queries were run on the embedded firebird database as were run on the SQLite version:

1. Inserting the pupil tracking data, about 600.000 records.
2. Retrieving the number of entries per pupil.
3. Retrieving the number of entries before 8:26h on a given date (776 pupils on September 6, 2005) using a left join.
4. Retrieving the number of different pupils that entered school before 8:26h on a given date (776 pupils on September 6, 2005). This is different from the previous query.

The queries were run using the isql tool that comes with Firebird, configured so it uses the embedded version. The queries were run a second time on a Client/Server installation of Firebird, to determine the overhead induced by the extra transport. The result is given in the following table:

Figure 1: The browser application



Test	SQLite	Firebird	Emb. Firebird
1	0:0:42.00	0:3:19.79	0:1:29.47
2	0:6:12.75	0:0:26.72	0:0:02.58
3	0:0:00.67	0:0:00.64	0:0:00.44
4	0:5:59.38	0:0:00.45	0:0:00.48

From these timings one can see that inserts are slower in Firebird than in SQLite by a factor 2. However, the performance of SQLite is really very bad when it comes to complicated queries: here Firebird outperforms SQLite by several orders of magnitude. The difference between the embedded and client/server version of firebird is very small, except for the case of the inserts. This is logical, since there the overhead of sending 600.000 sql statements to the server will affect performance. The result sets of the other queries are quite small, so the influence of the client-server aspect is relatively unimortant. Larger result sets will obviously have a larger impact on the timings.

10 Conclusion

As can be seen from this article, using embedded Firebird is no harder than using the usual Client/Server setup of Firebird, both in Delphi and Lazarus: All standard delivered data access technologies are in more or less degree configurable to use various client libraries, making a switch to embedded Firebird easy: It's just a matter of distributing the right files with the application. The code changes required to switch from embedded to complete client-server are negligible. When it comes to choosing between SQLite and Firebird, the choice will probably depend on what kind of application needs to be programmed. For applications that need to run complicated queries, Firebird may prove a better choice than SQLite. For applications that just need to record a lot of data, SQLite can be preferrable, as the inserts are faster. If upscaling is an option for the future, then Firebird is better suited, since SQLite offers no remote-access technology.