C++ Wrapper Library for Firebird Embedded SQL

Written by:

Eugene Wineblat,

Software Developer of Network Security Team, ApriorIT Inc.

www.apriorit.com

1. Introduction
2. Embedded Firebird
   2.1. Limitations
   2.2. Building fbclient.dll
3. C++ Wrapper
   3.1. Work with DB
   3.2. Classes Forge and Procedure
   3.3. Work with types
   3.4. Classes BufferReader and BufferWriter
4. Structure of the project files
5. List of the recommended literature

1. Introduction

This article is devoted to the Embedded Firebird database usage and also development of C++ wrapper of this database.

Advantage of Embedded Firebird database is that it enables to save your application user's time on database installation and also reduce requirements to his technical knowledge level.

2.1. Limitations of Embedded Firebird

If you want to use Embedded Firebird you should know about those limitations it applies to your work (version 2.0.5):

- THERE IS NO SUPPORT OF UNICODE. If you want to store information in unicode then you will have to care about the strings to be twice shorter than the maximal string length, and also store them as the arrays of bytes. You will have to convert such array to the Unicode when you get it.

- THERE IS NO SUPPORT OF UNICODE PATHS. Especially sad thing here is that library functions for creation and opening the Firebird databases do not support the Unicode paths.

- SIZE OF THE STORED PROCEDURE NAMES IS LIMITED. Such limitation is present in each database; however, in Firebird this size is depressingly small - just 30 characters. Usually this size is enough, but if you use the automatically generated names of the stored

procedures, which include the names of the module and class, functional meaning and possibly limitations, it can be not enough. Anyway you should remember about this.

2.2. Building fbclient.dll

In order to proceed to the use of Firebird possibilities, you need to build the source code into the library (source code can be taken from the site http://firebirdsql.org).

To build it you should go to the directory *\builds\win32* and use .bat file make_all.bat. The result will be files with the generated base definitions, constants and descriptions of the functions which will be located in the *\output* directory.

In your work you will need the libraries fbclient_ms.lib (*\output\lib*), icu*.dll (*\output\bin*) and fbclient.dll (*\output\bin*). However, it is necessary to remember that if you want to use Embedded Firebird then you should rename fbembed.dll in fbclient.dll and use this library.

For our wrapper we chose the least of files from *\output* and *\src* sufficient to use Firebird. These chosen in experimental way files can be seen in the directory of our project *\fbembed_provider_include*.

To work with the Firebird database you should link library fbclient_ms.lib (*\fbembed_provider_include\lib*) to your project and place all dll-s (*\fbembed_provider_include\dll*) in the system directory (*/windows/system32*).

3. C++ Wrapper

Our wrapper allows to:
- Put aside the details of FireBird database implementation
- Control resources during the work
- Control and process exceptional situations and errors
- Work with DB through the stored procedures
- Work with all types of data in the same way.

3.1. Work with DB

There are such functions to work with the database files:

- CreateDatabase – creates a file with empty database. Full path to the file with its name and extension should be specified.

It is also possible to specify login and password to access the base. If they are not specified then the standard login "SYSDBA" and password – "MASTERKEY" are used.

UNICODE_FSS is the base encoding; however it does not mean that a database will begin to work with unicode.

- AttachDatabase – sets connection with DB. UNICODE_FSS encoding is set by default.

- CreateFirebirdDatabaseFromFile – creates a database from a file.

To execute firebird-SQL constructions in different transactions the "GO$" separator is used.

Possible constructions that can't be used in one transaction are simultaneous table creation and using.

Example. Creation and connection with the database.

```
std::string db_name = "c:\defaultbase.fdb";
fb::CreateDatabase ( db_name );
fb::CFirebirdForge forge ( db_name ); // attach inside
```

3.2. Forge and Procedure

CFirebirdForge class enables to set connection with a database (*.fdb), and create two types of procedures to work with the data in this base:

- CFirebirdProcedure (CFirebirdForge::CreateProcedure) is to call the existent in database stored procedure.

The name of the procedure is passed to the constructor and also the pointer to the buffer where all obtained data will be saved.

If the stored procedure has input parameters then it is necessary to add them in the same sequence and with the same names as they are used in this procedure (functions of AddTTTIn type are used, where TTT is a type. An object will be created in the procedure to implement the structure and behavior of this type).

If the procedure has output parameters then it is necessary to define them by means of the functions AddOutTTT, where TTT is a type of the obtained data. It is necessary to pass the name which is used in the RETURNS block of the procedure as a parameter. Parameters are defined in the same sequence as they are mentioned in the function description.

Example. Creation and execution of the procedure.

```
fb::CFirebirdForge forge ( db_name );
std::auto_ptr< fb::CFirebirdProcedure > pProc (forge.CreateProcedure ( "TEST_{id}",
"INSERT", NULL) ); //this should be TEST_INSERT procedure

// add correct input arguments by type and sequence
pProc->AddStringIn( strVal );
pProc->AddWStringIn ( wstrVal );
pProc->AddDoubleIn ( dVal );
pProc->AddSingleIn ( fVal );
pProc->AddBinaryIn ( binVal );
pProc->AddInt8In ( int8Val );
pProc->AddInt16In ( int16Val );
pProc->AddInt32In( int32Val );
pProc->AddInt64In ( int64Val );
pProc->AddDateTimeIn ( timeVal );
pProc->AddGuidStringIn ( guidVal );
pProc->AddLongBinaryIn ( lbinValCpy );
pProc->AddLongWStringIn ( ltxtVal );

pProc->Execute();
```

- CFirebirdExecutable (CFirebirdForge::CreateQuery) is the procedure to execute SQL-query without parameters and returned values.

Example.

```
fb::CFirebirdForge forge ( db_name );
std::auto_ptr< fb::CFirebirdExecutable > pQueryTable ( forge.CreateQuery ("CREATE
TABLE TBLTESTTYPES( \
   STRING Varchar(255),}
```

```
_   WSTRING  Varchar(512),}

_   DOUBLE_T  Double  precision  NOT  NULL,}

_   FLOAT_T  Float  NOT  NULL,}

_   BINARY  Varchar(1024),}

_   INT8  Char(1)  NOT  NULL,}

_   INT16  Smallint  NOT  NULL,}

_   INT32  Integer  NOT  NULL,}

_   INT64  Numeric(10,0)  NOT  NULL,}

_   DATETIME  Timestamp  NOT  NULL,}

_   GUID  Varchar(36)  NOT  NULL,}

_   BLOBBIN  Blob  sub_type  0,\
   BLOBTEXT  Blob  sub_type  1\
);" ) );
pQueryTable->Execute  ();
```

If it is necessary to execute a SQL-query which returns some parameters, simply create a stored procedure for it using CreateQuery and call it by CreateProcedure.

3.3. Work with types

Te behavior of different data types which the wrapper works with is described in the CSQLData class. This is a template class, which is instantiated corresponding to the data:

- SMALLINT
- INTEGER
- INT64
- FLOAT
- DOUBLE PRECISION
- CHAR (n)
- VARYING
- TIMESTAMP
- DATE
- TIME
- BLOB

Depending on the C++ data types you need to store the different functions of FirebirdProcedure class are used. Functions of this class allocate the memory for variables and convert their values to the format required for work with Firebird SQL. To put these problems aside from the wrapper users the format of Firebird SQL fields where the information will be stored is predefined. You can see the correspondence in the table:

C++	FirebirdProcedure::Add	Firebird SQL
float	Single	Float
double	Double	Double precision
char	Int8	Char(1)
short	Int16	Smallint
int	Int32	Integer
__int64	Int64	Numeric(10,0)
std::string	GuidString	Varchar(36)
std::string	String	Varchar(n)
std::wstring	WString	Varchar(n)
std::wstring	LongWString	Blob syb_type 1
std::vector< char >	Binary	Varchar(n)
std::vector< char >	LongBinary	Blob sub_type 0
double	DateTime	Timestamp

For simple types (integer ones and types with a floating point) the general template format is used. For types which do not just allocate data or use difficult logic of converting there are specializations:

- time (TIMESTAMP)

In our realization the time is used as a double type and is a variant presentation of time (see functions of converting in fbutils.cpp - SystemTimeToTmTime / TmTimeToSystemTime). Functions for time converting are developed using the internal Firebird classes (*src**common**classes**timestamp.h .cpp*)

- string (VARYING and CHAR)

It is necessary to allocate memory dynamically in this type, and there is different volume for CHAR and VARYING. Also it is possible to add Unicode to them by std::wstring - it will be converted to the set of bytes. You just need only to watch that the size of such string is twice less than the maximal length of the field.

- arrays of data (BLOB)

Special group of functions is used to work with the BLOB type.

Firebird does not enable to write or read the arrays of data directly. To work with them you should first get a handle, by means of which you can write or read information. Functions isc_create_blob, isc_put_segement are used for writing and isc_open_blob, isc_get_segment are used for reading. A maximal size of segment in BLOB is `32768 byte`.

If you want to write NULL value in any field of any type then during the initialization of variables in procedure write -1in the sqlind field of the XSQLVAR structure. The same is right for the getting the information – if sqlind is equal to -1 it means that returned value is NULL.

We didn't implement such types as DATE, TIME in our code, but it's not hard to add them if you want to use just these types.

3.4. BufferReader and BufferWriter

All data obtained from the stored procedures are serialized in the buffer by the BufferWriter class. Correspondingly you should then get them from the buffer using BufferReader class.

The int32 field goes at the beginning of all information, the amount of the obtained records is written in it – this field is necessarily present in every resulting buffer. If 0 records were returned, then the buffer will contain only 0.

The special function of BufferReader class corresponds to every data type:

Single – read_single
Double – read_double
Int8 – read_int8
Int16 – read_int16
Int32 – read_int32
Int64 – read_int64
GuidString – read_
String – read_string_as_wstring
WString – read_wstring2
LongWString - read_wstring2
Binary – read_raw / read_raw_ref
LongBinary - read_raw / read_raw_ref
DateTime – read_double

Example. Obtaining results.

```
std::vector< char > arrOut;
std::auto_ptr< fb::CFirebirdProcedure> pProc ( pAdoForge->CreateProcedure ( "TEST_{id}", "SELECT", &arrOut ) );
... // add out parameters
pProc->Execute();

cmn::BufferReader br;
br.ConnectTo ( &arrOut.front(), arrOut.size() );

for ( int I = 0, count = br.read_int32(); I != count; ++I )
{
   // get the return values from buffer
  //br.read_string () / br.read_wstring () / br.read_double ();
}
```

4. Structure of project files

Src\FireBird.sln – solution file for Microsoft Visual Studio 2003
Src\fbembed_provider_include\ - headers, library and dll files of Firebird version 2.0.5, built for Win32 platform
Src\ fbembed_provider_lib - lib-library project with Firebird C++ Wrapper realization
Src\utils - project with utility functions and classes, used in the realization of Firebird C++ Wrapper

5. List of the recommended literature

All information about Firebird Sql can be found in the official sources (http://www.firebirdsql.org/index.php?op=doc).

Download Wrapper Source Files from our site - http://www.apriorit.com/our-articles/firebird-embeded-sql-wrapper.html .