Firebird ODBC/JDBC Driver 2.0 Manual

Alexander Potapchenko, Vladimir Tsvigun, Pavel Cisar, Jim Starkey, others

Version 1.0.3, 30 August 2020

Chapter 1. ODBC/JDBC Driver for Firebird Client Applications

This manual documents the official driver for connecting ODBC-aware client applications with a Firebird database, implementing the combined capabilities of dedicated wrappers for the Firebird C/C++ API functions with an ODBC-to-JDBC bridge to enable cross-platform connections in a Java VM environment.

1.1. About the Firebird ODBC driver

The Firebird ODBC driver supports client applications connecting to Firebird databases from Windows, FreeBSD, Solaris, and Linux. Separate kits are available for both Windows and the POSIX platforms, for use with 32-bit or 64-bit clients. On Windows, the respective dynamic `OdbcFb.dll` and the static `OdbcFb.lib` libraries are packaged in both .zip archives and executable installers. The POSIX packages come as either the binaries for x86 and amd64, respectively, both named `libOdbcFb.so`, or as a source code tarball. This help file is also included in the installation kits.

1.1.1. Features Supported

- Compiling for both 32-bit and 64-bit Windows clients on the Microsoft SDK base
- Unicode
- Thread-safe querying and other processing
- Creating databases by means of functions `SQLConfigDataSource`, `SQLDriverConnect`, `SQLExecDirect`.
- Multiple simultaneous transactions per connection, with varying transaction attributes if need be. For example, one read-only transaction, one or more simultaneous read/write transactions.
- Transparent connection pooling via transaction settings
- Firebird database events returned by triggers and stored procedures
- Use of Microsoft ODBC cursors (`odbccr32.dll`, `odbccu32.dll`)
- Firebird Services API (backup & restore, statistics, repair) by way of the function `SQLConfigDataSource`
- The schemas `SCHEMA` or `OWNER` for cases where a schema is required for cross-DBMS compatibility in SQL queries
- Fully functioning SQL syntax support for Services transactions via Firebird's *gpre* pre-compiler language ("EmbedSQL")
- Use of the COM interface for Microsoft Distributed Transaction Coordinator (DTC)

Chapter 2. Installing the Driver

The kit that you install will depend on what you plan to use it for. Regardless of whether you intend to connect to a 64-bit or a 32-bit Firebird server, you must install the driver and the Firebird client (`fbclient.dll` on Windows, `libfbclient.so` on Linux) that matches the "bitness" of your client application.

Installation is similar for both options. You can install both the 32-bit and the 64-bit driver on the same machine if the user is going to access Firebird from multiple applications of mixed bitness. Care will be needed to ensure that each application will connect using the correct DSN for the required driver.

Note for the Less Technically Versed

...because we have been asked: if you want to connect your Windows application — Excel or LibreCalc, for example — to your database running on a Linux or other POSIX server, you want the Windows driver, not the POSIX one. See also the note below about the Firebird client library.

2.1. Downloading the Driver

The Downloads section at https://www.firebirdsql.org/en/odbc-driver/ clearly identifies the bitness of the various kits available, with the latest release at the top of the page. For example, the 32-bit installer kit for Windows, at the time this document was prepared, was named `Firebird_ODBC_2.0.5.156_Win32.exe`, indicating that it is the executable installer for the 32-bit version. The following table should help to indicate what you will need. The "N.n.n.xxx" infix used here indicates "Major1.Major2.Minor.Subrelease". The "Subrelease" part changes the most frequently.

Table 1. Firebird ODBC/JDBC Driver Kits

Kit Name	Purpose
`OdbcJdbc-src-N.n.n.xxx.tar.gz`	Source code, which is bitness-independent. Recommended for POSIX installs with unusual rules about the location of libraries — instructions below.
`Firebird_ODBC_N.n.n.xxx_Win32.exe`	Executable installer for use with 32-bit client applications. Use this for an initial installation.
`Firebird_ODBC_N.n.n.xxx_x64.exe`	Executable installer for use with 64-bit client applications. Use this for an initial installation.

Kit Name	Purpose
`OdbcFb_DLL_N.n.n.xxx_Win32.zip`	Zip kit containing just the dynamic and static 32-bit libraries and documentation. This can be used to update the library of an existing installation when the driver is not active. On a 64-bit machine, the older version can be found in the folder `c:\Windows\SySWOW64` and Administrator privileges will be required to overwrite it.
`OdbcFb_DLL_N.n.n.xxx_x64.zip`	Zip kit containing just the dynamic and static 64-bit libraries and documentation. This can be used to update the library of an existing installation when the driver is not active. On a 64-bit machine, the older version can be found in the folder `c:\Windows\system32` and Administrator privileges will be required to overwrite it. It will not work on a 32-bit machine.
`OdbcFb-LIB-N.n.n.xxx.i686.gz`	32-bit binary for a POSIX client, gzipped
`OdbcFb-LIB-N.n.n.xxx.amd64.gz`	64-bit binary for a POSIX client, gzipped

2.2. Getting the Right Firebird Client Library

All Firebird RDBMS kits contain at least one version of the Firebird client library. If there is only one, then it will have the same “bitness” as the server installation kit itself.

Make sure you get the fbclient library that has the same major version number as the server it is going to connect with.

- On a 32-bit Windows installation, `fbclient.dll` is in Firebird’s `bin` folder in Firebird versions lower than version 3.0. For version 3.0 and above, it is in Firebird’s root folder, e.g. `C:\Program Files (x86)\Firebird\Firebird_3_0`, or wherever Firebird was installed.
- On a 64-bit Windows installation, the version of `fbclient.dll` in Firebird’s `bin` folder (or Firebird’s root folder for version 3.0 and higher) is the 64-bit one. In some builds, the 32-bit client is located in a folder, named either WOW64 or system32, that is beneath Firebird’s root.

  If your ODBC DSN setup is going to need the 32-bit `fbclient.dll` and it is not there, you will need to download the 32-bit Windows `.zip` kit from the main Firebird download page, extract the 32-bit client from it and place it in the same folder as your application. An alternative is to download the 32-bit installer instead and perform a client-only install, configuring the installer to place it where you want it to be.

- The POSIX server kits always come with only the matching `libfbclient.so`. You will need to extract it from an .i686 kit if your POSIX client application is 32-bit.

Have the client library in its proper place **before** installing the driver and configuring the DSN.

Compatibility of the Driver with Firebird Versions

The most current version of the ODBC/JDBC driver is expected to be compatible with any supported version of Firebird.

2.3. Installing the Driver on Windows

If you are doing a first-time install of the driver, or if you have uninstalled an older version, it is recommended that you use the executable installer. These instructions will assume that you are installing the 32-bit driver, but the procedure is the same for installing the 64-bit one. Under the hood, the 32-bit driver library will be installed into `\windows\sysWOW64` on a 64-bit Windows. Any other install will place the driver in `windows\system32`.

Download or move the executable installer kit to the desktop. Right-click on it and select **Run as Administrator**.

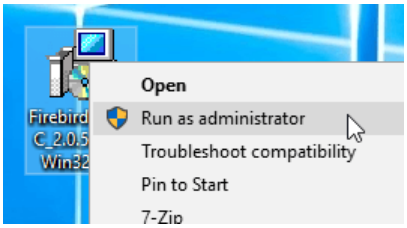

Figure 1. ODBC driver installer on the desktop

Click your way through the screens until you reach the one in which you configure your preferences for the installation:

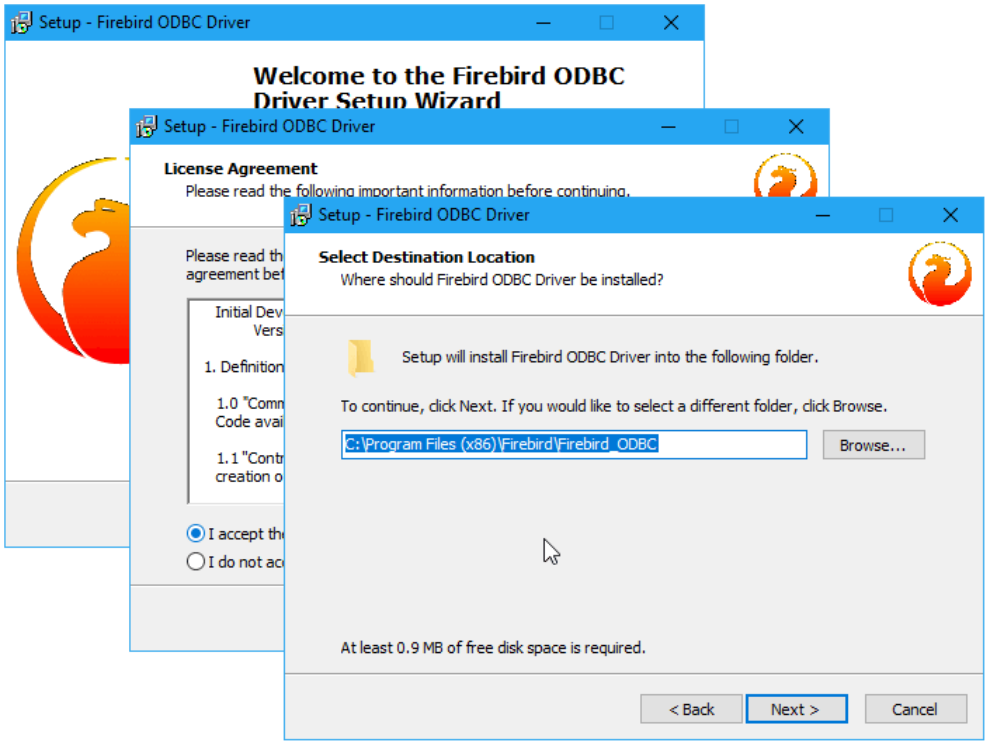

Figure 2. ODBC driver installer screens

If you want or need to, you can have the driver installed in some other location than the one offered by the installer as the default. Use the **[Browse]** button to find the location where you want

to have the driver installed.

The installer will create the `\Firebird_ODBC` subfolder if it does not exist already.

Lastly, the installer will display the configuration you have chosen. If you happy with it, just click **[Install]** and it is done.

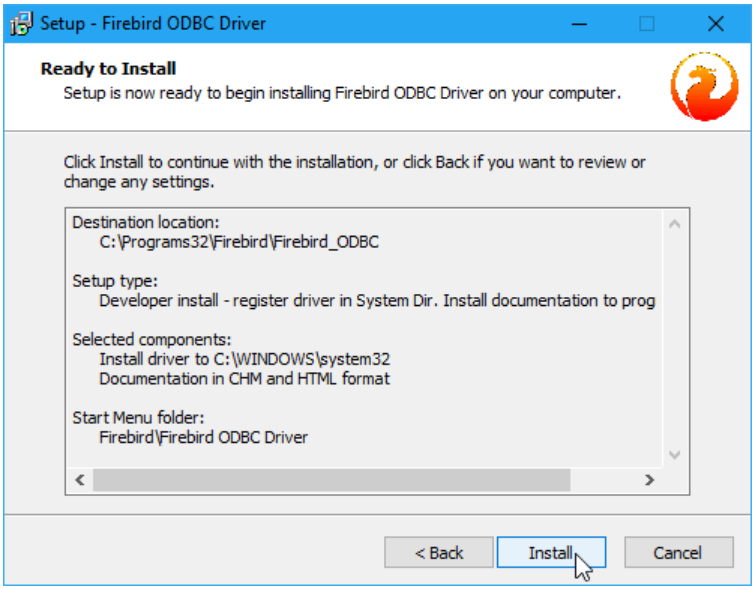

Figure 3. ODBC driver installer final screen

You might observe here that, on our system, we keep our own dedicated "Programs64" and "Programs32" folders under `C:\Windows`. That is simply preference as to how we organise our server and monitor the volume of stuff installed by Windows updates into its own program folders.

The `.chm` and `.html` documents noted on that screen are older evolutions of this document that are still built in with the kits at the point of this writing.

2.4. Installing the Driver on Linux

Pavel Cisar

There are two prerequisites for installing the ODBC/JDBC driver on Linux:

- The "unixODBC" package must be installed
- Firebird must be installed, initially at least, for testing the installation

2.4.1. Unpacking the Files

The ODBC/JDBC driver packages for Linux are gzipped `tar` files. After `gunzip` they should be processed by tar, or you can rename them to `.tar.gz` and use a tool such as Midnight Commander to unpack them.

2.4.2. Building from Sources

Building from source code (recommended), requires the development package for unixODBC. Proceed with the following steps:

1. Download and unpack the Firebird driver sources
2. Rename `makefile.linux` in `.source/Builds/Gcc.lin` to `makefile`
3. Set the evironment variables `FBINCDIR` (Firebird include directory) and `FBLIBDIR` (Firebird lib directory) if necessary.
4. Run `make` which will create the library `libOdbcFb.so` in a subdirectory
5. It is possible to copy the library to `/usr/local/lib64` or any preferred directory; or run `make install` to symlink the library from the `unixODBC` directory

2.4.3. Installing the Binary Package

To install from the binary package, copy `libOdbcFb.so` to `/usr/local/lib64`, `/usr/local/lib32` or any other desired destination directory, as appropriate.

Chapter 3. Firebird ODBC Configuration

The configuration settings you make in an ODBC data source description (“DSN”) define the attributes for connecting to a specific database. On Windows, a dialog box captures parameters that correspond to the connection attributes. On Linux, the parameters are configured manually in text (`.ini`) files.

3.1. Configuring a DSN on Windows

First, find the applets in the *Administrative Tools* section of the machine where you are going to set up a “channel” through which your application program is going to connect with a Firebird database, either on the same machine or elsewhere in the local or wide-area network.

On a 64-bit machine, you will find two such applets:

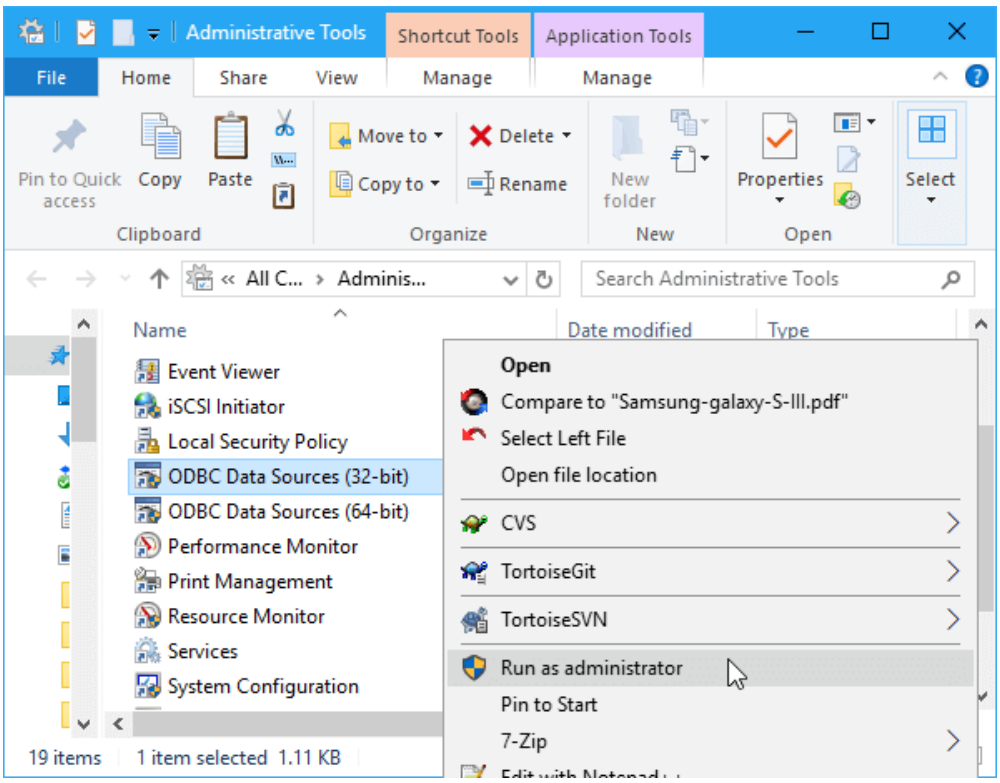

Figure 4. Selecting a DSN setup applet on Windows

For the purpose of our example, we want to pick the item *ODBC Data Sources (32-bit)*. Obviously, if we had installed the 64-bit driver with the intention of using it for a 64-bit application, we would pick the 64-bit item from this menu instead.

Run as Administrator!

Don’t left-click the item: right-click and, from the context menu, select **Run as Administrator**. This is necessary because you are about to set up a *System DSN*.

Click on the tab labelled **System DSN**, where you will begin setting up your DSN.

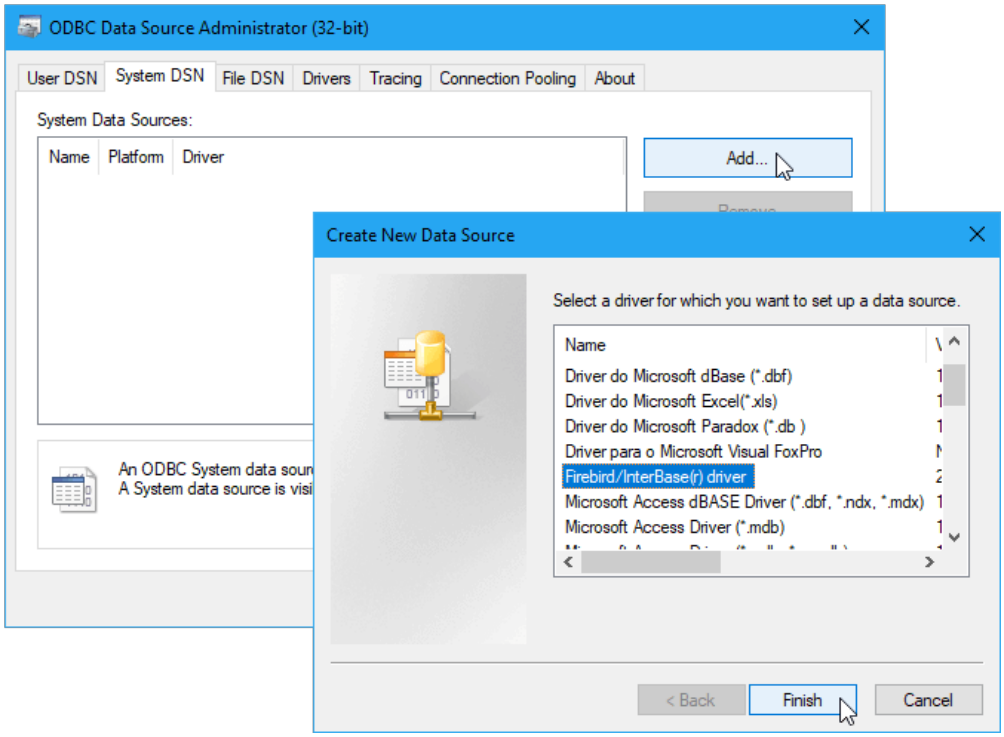

Figure 5. Selecting the Firebird driver for the DSN

Click **[Add...]** on the first screen to bring up the list of drivers on the next. Select the `Firebird/InterBase(r)` driver, then click **[Finish]**.

3.1.1. The DSN Settings

After clicking **[Finish]** on the previous screen, you are presented with a form into which you will enter the parameters for a connection and will be able to test that they all work.

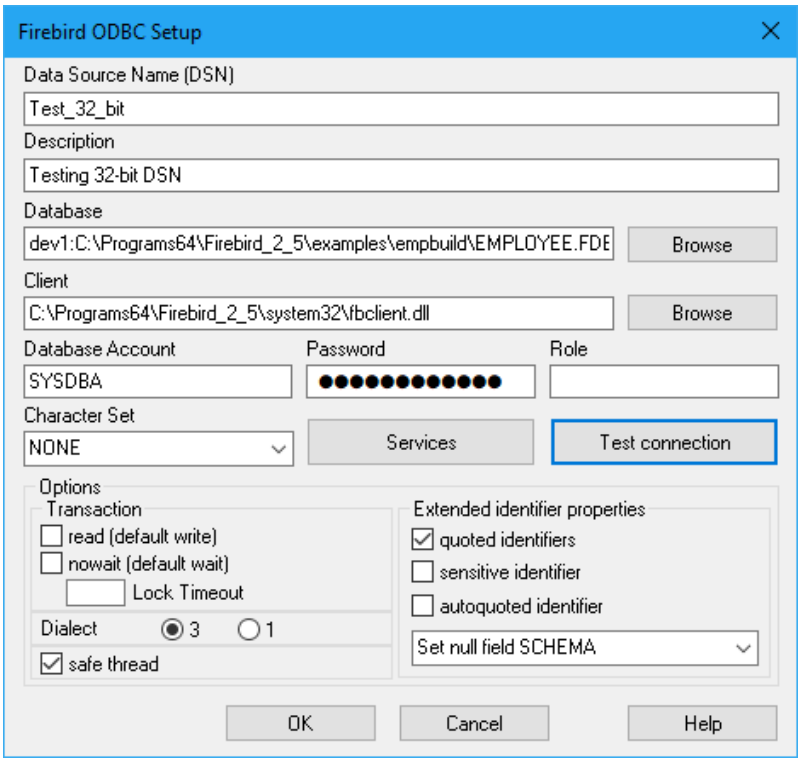

Figure 6. Setting up parameters for the DSN

Table 2. Parameters for the DSN Configuration

Parameter	Entry
Data Source Name (DSN)	REQUIRED. A unique, meaningful name indicating the type of connection or its use. Make it brief as you can expand the narrative elsewhere. Examples: “Connect from FbEmbed” or “ConnectFbServer”
Description	Optional. Can be used to provide more details about the data source.
Database	REQUIRED. Full address of the database, as required for an embedded or network connection. If the connection is remote, it can be in TCP/IP or WNET format. TCP/IP is recommended. Firebird database aliases are supported. Refer to Connection Examples.
Client	May be required. Local path to the Firebird client library. For embedded connections to a sub-V.3 Windows server, it can point to the copy of `fbembed.dll` in the application directory; otherwise, point it to where you have located the bitness-compatible Firebird remote client library unless you are certain the correct library will be found automatically in a system location.
Database Account	Optional, since login credentials can be captured during connection to a Firebird database. If it is not configured, the ODBC interface will prompt for a user ID (UID or USER) at connection time.
Password	Optional, since login credentials can be captured during connection to a Firebird database. If it is configured, it should be the password for the supplied User ID. Otherwise, the ODBC interface will prompt for a password (PWD or PASSWORD) at connection time. Any password configured is encrypted automatically and saved in `odbc.ini`. Storing the password here should not be a security risk.
Role	Optional. If it is defined and the login is by SYSDBA, role is ignored; otherwise, the login credentials, whether stored or captured at connection, must have been granted that role prior to the login attempt.
Character Set	May be blank. Sets the default character set of the client.
Options (set here in DSN or specify dynamically)	
Transaction parameters	
Read (default write)	Transactions are read/write by default. Check to make transactions read-only.
Nowait (default wait)	The transaction will wait if it encounters a lock conflict. Check to have the transaction return an error immediately upon encountering a lock conflict.

Parameter	Entry
Lock timeout	When a transaction is set for WAIT conflict resolution, express the length of time in seconds until the lock times out and a lock conflict error is returned (`isc_lock_timeout`).
Other optional parameters	
Dialect	SQL dialect for the client to use in accessing the database. The only valid options for Firebird are 1 or 3. Note, Dialect 1 is not compatible with quoted identifiers; Dialect 3 will not accept strings delimited by double quotes.
Quoted Identifier	Causes pairs of double quotes to be treated solely as delimiters of case-sensitive object identifiers. Attempts to pass double quotes as string delimiters will be treated as errors in both dialects. Note, double-quoted strings have always been illegal in Dialect 3.
Sensitive Identifier	This option affects the way the client treats the property `SQL_IDENTIFIER_CASE`. `SQL_IC_UPPER` (value=1) is the default, treating all identifiers as stored in upper case characters. Check to select `SQL_IC_SENSITIVE` (value=3) to have the interface treat all identifiers that are not in all upper case as though they were case-sensitive. This is not recommended! For an explanation, see Note (1) below.
Autoquoted Identifier	Default is `NO`. The effect of checking this is to change the setting to `YES`. In that case, every identifier in every statement will be double-quoted automatically. The need to set this on would be highly unusual and would need to be well understood to avoid non-stop errors.
SCHEMA options	Drop-down list offering three options for treatment of SQL schemas, which Firebird does not support. Normally, leave this at the default setting **Set null field SCHEMA**. For some details, see Note (2) below.

Note (1) regarding “Sensitive identifier”

If this setting is checked on, it would cause this statement

```
SELECT A.Test_Field FROM Mixed_Caps_Table A
ORDER BY A.Test_Field
```

to be converted to

```
SELECT A."Test_Field" FROM "Mixed_Caps_Table" A
ORDER BY A."Test_Field"
```

The following would result in a wrong conversion:

```
Select A.Test_Field From Mixed_Caps_Table A
Order By A.Test_Field
```

gets converted to

```
"Select" A."Test_Field" "From" "Mixed_Caps_Table" A
"Order" "By" A."Test_Field"
```

Note (2) regarding SCHEMA settings

Some applications generate SQL statements automatically, based on user inquiries, on the assumption that the target database supports namespaces and SQL SCHEMAs. For example,

```
select SYSDBA.COUNTRY,SYSDBA.CURRENCY from SYSDBA.COUNTRY
```

or

```
select * from SYSDBA.COUNTRY
```

This selection of schema settings attempts to prevent clashes with applications that do this kind of thing. The drop-down list offers the three variants:

1. Set null field SCHEMA
2. Remove SCHEMA from SQL query
3. Use full SCHEMA

Set null field SCHEMA is the default, causing the SCHEMA element to be set `NULL` whenever it is specified as part of a query. The result is a query that Firebird can process.

Remove SCHEMA from SQL query filters the namespace references from the statement whenever the `SQLExecDirect` command receives a request such as

```
select SYSDBA.COUNTRY,SYSDBA.CURRENCY from SYSDBA.COUNTRY
```

transforming it before passing it to the API as

```
select COUNTRY,CURRENCY from COUNTRY
```

Use full SCHEMA] is reserved for a future in which Firebird has the capability to process these concepts itself — perhaps in Firebird 4. In that event, the driver will have no need to screen out these constructions.

Click on the **[Test connection]** button to confirm that your configuration is good:

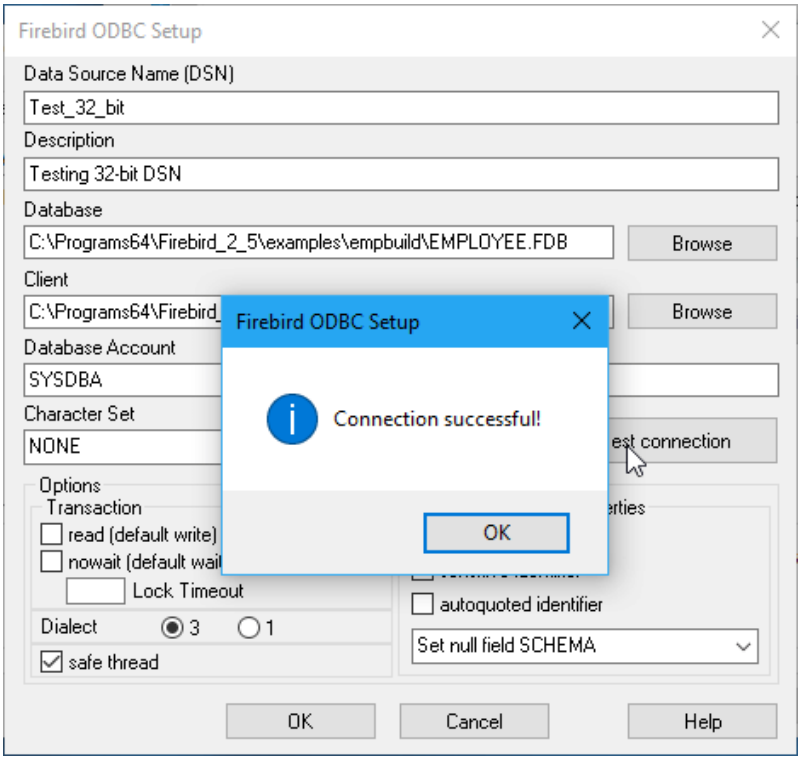

Figure 7. Testing the Configuration

If all is well, click **[OK]**, return to the main form and save the configuration by clicking **[OK]** there, too.

3.1.2. The Services Button

The Services button launches a number of server management utilities through a GUI management console. It is described later in The Services Interface.

3.2. Configuring a DSN on Linux

Pavel Cisar

Configuration depends on the Linux distribution but, somewhere in `/etc` or `/etc/unixODBC`, should be two files named `odbc.ini` and `odbcinst.ini`.

Add to odbcinst.ini:

```
[Firebird]
Description     = InterBase/Firebird ODBC Driver
Driver          = /usr/local/lib64/libOdbcFb.so
Setup           = /usr/local/lib64/libOdbcFb.so
Threading       = 1
FileUsage       = 1
CPTimeout       =
CPReuse         =
```

Add to odbc.ini:

```
[employee]
Description     = Firebird
Driver          = Firebird
Dbname          = localhost:/opt/firebird/examples/empbuild/employee.fdb
User            = SYSDBA
Password        = masterkey
Role            =
CharacterSet    =
ReadOnly        = No
NoWait          = No
```

3.2.1. Testing the Configuration

UnixODBC has a tool named `ISQL` (not to be confused with Firebird's tool of the same name!) that you can use to test the connection, as follows:

```
isql -v employee
```

If you have connection problems, make sure the directory where you placed the Firebird ODBC shared library, e.g. `/usr/local/lib64/libOdbcFb.so`, is on the system loadable library path. If not you could set:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/lib/odbc
```

or, more simply,

```
export LD_LIBRARY_PATH=/usr/lib/odbc
```

If you still have problems, the next thing is to try an `strace` to try to identify them:

```
strace -o output.txt isql -v employee
```

Chapter 4. Connecting to Firebird from Applications

The ODBC/JDBC driver attempts to connect a client to the Firebird server according to a set of attributes that default to those provided by the DSN definition. Those stored attributes can be — and usually are — overridden by parameters passed by the application or read from a file (FILEDSN) when it prepares to connect.

4.1. Connection Parameters

The connection parameters for the driver comprise a list of strings in the form `KEYWORD=value`, separated by semicolons (';'). The following table enumerates the keywords with their verbose meanings and, where it is not obvious, their possible values.

Table 3. Keywords for Connection Attributes

Keyword	Description	More Information
`UID`	Database account, i.e. username	
`USER`		
`PWD`	Password	
`PASSWORD`		
`ROLE`	Role	
`DSN`	Data source name	
`DRIVER`	Driver name	e.g., the string `Firebird/InterBase(r) driver`. Defaults to the driver defined in the DSN.
`DBNAME`	Database	Full path to the database **as seen by the server**, including IP address | server name[/port] for a remote connection. Defaults to the database defined in the DSN.
`DATABASE`		
`CLIENT`	Local path to the required `fbclient` library	May be needed if the connection is to be via an embedded server library located in an application folder.
`CHARSET`	Client-side default character set	Should be the same as the default character set of the database, if possible; or one that is known to be codepage-compatible.
`CHARACTERSET`		
`READONLY`	Read-only	Set transactions in this connection to be read-only. The default is read/write.
`NOWAIT`	No wait	Set transactions in this connection to have NO WAIT lock resolution. The default is WAIT.
`LOCKTIMEOUT`	Set the lock timeout on WAIT transaction	Pass the number of seconds to elapse after encountering a lock conflict until a transaction is to return an error. Not valid if the transaction is set for NO WAIT resolution.

Keyword	Description	More Information
DIALECT	Set SQL dialect	Only 1 or 3 is valid. Normally this would have been set in the DSN. It must match the dialect of the database.
QUOTED	Set on quoted identifiers	If set in the DSN, the setting should be correct, i.e. already ON or OFF.
SENSITIVE	Set on case-sensitive identifiers	If set in the DSN, the setting should be correct, i.e. already ON or OFF.
AUTOQUOTED	Set on auto-quoting identifiers	If set in the DSN, the setting should be correct, i.e. already ON or OFF.
USESCHEMA	Set on “use schema”	If set in the DSN, the setting should be correct.
SAFETHREAD	Safe threading	
FILEDSN	File DSN	Path to a file where the attribute strings from a previous connection are stored. If this string is present, the contents of the file will take priority over the main DSN.
SAVEDSN	Save DSN	Path to a file where the attribute strings from this connection, if successful, are to be stored. The password will be saved in encrypted format.

4.1.1. Read Sequence of the Keys

The ODBC function SQLDriverConnect gives priority to the attributes defined in the connection string, only fetching those stored in the DSN, or in a cited FILEDSN, to fill in any gaps.

4.1.2. Connection Examples

Some examples of connection strings for applications that use the ODBC function SQLDriverConnect:

```
Open("DSN=myDb;")
```

Here, the function is expected to read everything it needs from the DSN. User name and password are not supplied in the string. If they are not present in the DSN, either

1. it will use the environment variables ISC_PASSWORD and ISC_USER if they are are set; otherwise
2. it will prompt the user for the login credentials

```
Open("DSN=myDb; UID=MCSSITE; PWD=mcssite;")
```

The function should have what it needs to make this connection, provided the user name and password are authenticated by the server.

```
Open("DSN=myDb; UID=MCSSITE; PWD=mcssite; DBNAME=172.17.2.10:/usr/local/db/myDb.fdb;")

Open("DSN=myDb; UID=MCSSITE; PWD=mcssite; DBNAME=myserver:/usr/local/db/myDb.fdb;")
```

The `DBNAME` key points to the server IP address in the first example, with the path to the database file in the POSIX format. The second example is making the same connection, using the server's host name instead of the IP address.

Three examples including the `DRIVER` attribute in the string:

```
Open("DRIVER=Firebird/InterBase(r) driver;
DBNAME=172.17.2.10:/usr/local/db/myDb.fdb;")

Open("DRIVER=Firebird/InterBase(r) driver; UID=MCSSITE; PWD=mcssite;
DBNAME=172.17.2.10:/usr/local/db/myDb.fdb;")

Open("DRIVER=Firebird/InterBase(r) driver; UID=MCSSITE; PWD=mcssite; DBNAME=dummy;")
```

In the last example, a local connection using a database alias in place of the database file path. Of course, the alias must be present in `aliases.conf` in the root directory of the Firebird server (or, for Firebird 3 and up, in `databases.conf`).

Using the server IP address and specifying an alternative port, with the target database on a POSIX server; and the same using the server's host name instead:

```
172.17.2.10/3051:/usr/local/db/myDb.fdb

myserver/3051:/usr/local/db/myDb.fdb
```

Using the server IP address, with the target database on a Windows server; and the same using the server's host name instead:

```
172.17.2.10:c:\db\myDb.fdb

myserver:c:\db\myDb.fdb
```

Using the server IP address and specifying an alternative port, with the target database on a Windows server; and the same using the server's host name instead:

```
172.17.2.10/3051:c:\db\myDb.fdb

myserver/3051:c:\db\myDb.fdb
```

Using TCP/IP local loopback, using the local loopback IP address on a POSIX server; and the same

using the local loopback host name `localhost`:

```
127.0.0.1:/usr/local/db/myDb.fdb

localhost:/usr/local/db/myDb.fdb
```

The same things on a Windows server:

```
127.0.0.1:c:\db\myDb.fdb

localhost:c:\db\myDb.fdb
```

DBNAME for Embedded Connections

The `DBNAME` value for embedded connections and for the "Windows Local" (XNET) style of connection uses just the file path or alias, without host name, IP address or any port number.

From Firebird 3 on, the way we conceptualise non-network connections on all platforms is more unified than for the earlier versions. However, from the point of view of the ODBC/JDBC driver, the expression of the `DBNAME` value has not changed, regardless of the platform on which we are making our embedded connection.

Local connection on a Windows server using first the file path and next an alias:

```
DBNAME=C:\db\myDb.fdb

DBNAME=C:dummy
```

On a POSIX server:

```
DBNAME=/usr/local/db/myDb.fdb

DBNAME=dummy
```

DBNAME Using Aliases

It is strongly recommended to define and use aliases to simplify life for you and your users. It makes your `DBNAME` values completely neutral to the filesystem and so much less cumbersome. In our last pairs of examples, the same alias was used on both Windows and POSIX. The one on the Windows server would be defined thus:

```
dummy = C:\db\myDb.fdb
```

while, on the Linux server, it would be defined thus:

```
dummy = /usr/local/db/myDb.fdb
```

Chapter 5. Developing with the Firebird ODBC/JDBC Driver

The Firebird ODBC driver supports multiple simultaneous connections to different databases and different servers, each connection operating independently of any others.

5.1. Multithreading

Thread protection can be specified at two levels:

1. sharing an environment handle
2. sharing a connection handle

By default, the driver is built using the following define:

```
#define DRIVER_LOCKED_LEVEL     DRIVER_LOCKED_LEVEL_CONNECT
```

which enables a single connection to share multiple local threads.

The default setting is reflected in the initial setup of the DSN on Windows: `SAFETHREAD=Y`.

If the driver is built using the following define:

```
#define DRIVER_LOCKED_LEVEL     DRIVER_LOCKED_LEVEL_NONE
```

then the driver is built without multi-threading support and responsibility for threading control is transferred to the Firebird client library. This provides for fastest performance.

If you have a build that was made with this define, you should make it the default thread behaviour for the DSN by configuring `SAFETHREAD=N` in its interface.

If the driver is built using the following define:

```
#define DRIVER_LOCKED_LEVEL     DRIVER_LOCKED_LEVEL_ENV
```

then a single environment handle can be shared by multiple local threads.

You may save a specific set of connection conditions or overrides in a `FILEDSN`.

5.2. Transactions

Firebird supports three transaction isolation levels:

- READ COMMITTED

- SNAPSHOT (“concurrency” or “repeatable read”)
- SNAPSHOT TABLE STABILITY “consistency”)

The default isolation level of the ODBC/JDBC driver is READ COMMITTED, which maps with read committed in other database systems. Firebird’s other isolation levels do not map so easily. In the ODBC/JDBC driver, SNAPSHOT maps to REPEATABLE READ and SNAPSHOT TABLE STABILITY maps to SERIALIZABLE, with some tweaks.

Since version 2.0, the driver has been able to support every transaction configuration that Firebird can support, including table reservation (“table blocking”). That was achieved by incorporating the so-called “EmbeddedSQL” syntax that is native to the old pre-compiler, *gpre*, to prepare calls to the ODBC API by the function `SQLExecDirect`.

5.2.1. Locking

Firebird implements optimistic row-level locking under all conditions. A transaction does not attempt to lock a record until it is ready to post an update operation affecting that record. It can happen, though rarely, for an update to fail because another client has a lock on the record, even if the transaction that fails started before the one which secured the lock.

Firebird’s record versioning engine is able to achieve a granularity finer than that provided by traditional row-level locking. Versioning allows any number of transactions to read a consistent copy of any given record, even if other transactions are updating the same row simultaneously. Readers and writers never block one another and Firebird’s maintenance of record versions is totally transparent to the user.

5.2.2. Transaction Request Syntax

The syntax for an ODBC-friendly transaction request follows.

```
SET | DECLARE TRANSACTION [LOCAL] [NAME transaction-name [USING namedUniqueWorkspace]]
[READ WRITE | READ ONLY]
[WAIT | NO WAIT]
[AUTOCOMMIT]
[NO_AUTO_UNDO]
[[ISOLATION LEVEL] {SNAPSHOT [TABLE STABILITY]
| REPEATABLE READ
| SERIALIZABLE
| READ COMMITTED [[NO] RECORD_VERSION]}]
[RESERVING table-name-1 [, table-name-2[, ...table-name-n] ]
[FOR [SHARED | PROTECTED] {READ | WRITE}] [, ]
```

What the Options Mean

`DECLARE TRANSACTION⋯` declares the described transaction, without activating it. `SET TRANSACTION⋯`, on the other hand, activates the transaction, temporarily switching the `SQL_ATTR_AUTOCOMMIT` global attribute of the ODBC API to `SQL_AUTOCOMMIT_OFF`. The transaction will have to be finished explicitly; when it ends, the abiding rule of the API resumes.

`LOCAL` limits a transaction to acting only within the context of the current connection.

`NAME transaction-name` is a uniquely-named transaction, prepared for use by any connections in the global environment.

`USING namedUniqueWorkspace` is a uniquely-named transaction workspace in which `NAME transaction-name` can be set to run by any connections in the global environment. Identically named transactions with differing parameters can run in the same named workspace.

Named Transactions and Transaction Workspaces

The construct `DECLARE TRANSACTION ⋯ NAME transaction-name [USING namedUniqueWorkspace]` allows explicit transactions to be configured and saved into the global environment in preparation for repeated use for any connection request or by any active connection. An instance of the saved transaction can be called into action by a specific form of the `SET TRANSACTION` command:

For a connection request:

```
SET TRANSACTION NAME MyReadTransaction
```

or

```
SET TRANSACTION NAME MyReadTransaction USING MyDsnDb1
```

for separate requests within a single active connection:

```
SET TRANSACTION LOCAL NAME MyReadTransaction
```

or

```
SET TRANSACTION LOCAL NAME MyReadTransaction USING MyDsnDb1
```

and, in this connection, for another request:

```
SET TRANSACTION LOCAL NAME MyWriteTransaction
```

or

```
SET TRANSACTION LOCAL NAME MyWriteTransaction USING MyDsnDb1
```

The form `SET TRANSACTION ⋯ NAME transaction-name [USING namedUniqueWorkspace]` differs from earlier implementations whereby the configuration set by the `SET` command would be repeated for the next transaction. The inclusion of the `NAME` and/or `USING` clauses makes the configuration repeatable on demand by use of the name.

A return to the usual mode of operation requires a detach/connect cycle.

Ending Explicit Transactions

In SQL, a transaction is completed by a `COMMIT` or `ROLLBACK` request. ODBC has methods that do one or the other, such as `SQLEndTran`. Some programs are able to invoke `SQLExecDirect` but cannot call `SQLEndTran`. For those programs it is necessary to call an explicit

```
SQLExecDirect( hStmt, "COMMIT" )
```

to ensure that the interface will call

```
SQLEndTran( SQL_HANDLE_DBC, hConnection, SQL_COMMIT );
```

If a transaction is initiated locally, the driver will execute `SQLEndTran` for the local `hStmt`.

5.2.3. Two Phase Commit Transactions

The ODBC/JDBC driver supports two-phase commit transactions, that is, a single transaction across different Firebird databases. Up to 16 databases can be accessed simultaneously in one such transaction — that is an absolute limit.

The call to start a two-phase commit transaction is:

```
SQLSetConnectAttr (connection, 4000, (void*) TRUE, 0);
```

To cancel the common connection:

```
SQLSetConnectAttr (connection, 4000, (void*) FALSE, 0);
```

5.2.4. More Transactions

Firebird ODBC by default uses one transaction per connection. Programmatically you can use a more flexible transaction structure. For example, you can use multiple transactions within one connection, whereby a single connection can be using a number of read/write transactions simultaneously.

An Example

```
HSTMT stmtRd;
HSTMT stmtWr;
SQLAllocHandle( SQL_HANDLE_STMT, connection, &stmtRd );
SQLAllocHandle( SQL_HANDLE_STMT, connection, &stmtWr );
SQLExecDirect( stmtRd, (UCHAR*)
  "SET TRANSACTION LOCAL\n"
  "READ ONLY\n"
  "ISOLATION LEVEL\n"
  "READ COMMITTED NO RECORD_VERSION WAIT\n",
  SQL_NTS );
SQLExecDirect( stmtWr, (UCHAR*)
  "SET TRANSACTION LOCAL\n"
  "READ WRITE\n"
  "ISOLATION LEVEL\n"
  "READ COMMITTED NO RECORD_VERSION WAIT\n",
  SQL_NTS );
SQLExecDirect( stmtRd,(UCHAR*)
  "SELECT CURRENCY FROM COUNTRY"
  "   WHERE country = 'Canada'"
  "   FOR UPDATE OF CURRENCY",
  SQL_NTS );
SQLFetch( stmtRd );
SQLPrepare( stmtWr, (UCHAR*)
  "update COUNTRY\n"
  "set    CURRENCY = 'CndDlr'\n"
  "where  COUNTRY = 'Canada'\n",
  SQL_NTS );
SQLExecute( stmtWr );
SQLExecDirect( stmtWr, (UCHAR*)"COMMIT", SQL_NTS );
```

5.2.5. MS DTC Transactions

The Microsoft Distributed Transaction Coordinator (MS DTC) service is a Windows component that is responsible for coordinating transactions that span multiple resource managers, such as database systems, message queues, and file systems. It can perform global, single-phase or two-phase commit transactions involving Microsoft SQL Server, Sybase and other servers that are able to work with it. Our ODBC/JDBC driver provides that capability for Firebird servers.

An Example Using MS DTC

```
// Include MS DTC specific header files.
//--------------------------------------------------------------------------
#define INITGUID
#include "txdtc.h"
#include "xolehlp.h"

ITransactionDispenser *pTransactionDispenser;
ITransaction *pTransaction;
// Obtain the ITransactionDispenser Interface pointer
// by calling DtcGetTransactionManager()
DtcGetTransactionManager( NULL,// [in] LPTSTR pszHost,
     NULL,// [in] LPTSTR pszTmName,
     IID_ITransactionDispenser,// [in] REFIID rid,
     0,// [in] DWORDdwReserved1,
     0, // [in] WORDwcbReserved2,
     NULL,// [in] void FAR * pvReserved2,
     (void **)&pTransactionDispenser // [out] void** ppvObject
     );
// Establish connection to database on server#1
LogonToDB( &gSrv1 );
// Establish connection to database on server#2
LogonToDB( &gSrv2 );
// Initiate an MS DTC transaction
pTransactionDispenser->BeginTransaction(
     NULL,// [in] IUnknown __RPC_FAR *punkOuter,
     ISOLATIONLEVEL_ISOLATED,// [in] ISOLEVEL isoLevel,
     ISOFLAG_RETAIN_DONTCARE,// [in] ULONG isoFlags,
     NULL,// [in] ITransactionOptions *pOptions
     &pTransaction// [out] ITransaction **ppTransaction
     );
// Enlist each of the data sources in the transaction
SQLSetConnectOption( gSrv1->hdbc, SQL_COPT_SS_ENLIST_IN_DTC, (UDWORD)pTransaction );
SQLSetConnectOption( gSrv2->hdbc, SQL_COPT_SS_ENLIST_IN_DTC, (UDWORD)pTransaction );
// Generate the SQL statement to execute on each of the databases
sprintf( SqlStatement,
  "update authors set address = '%s_%d' where au_id = '%s'",
   gNewAddress, i, gAuthorID );
// Perform updates on both of the DBs participating in the transaction
ExecuteStatement( &gSrv1, SqlStatement );
ExecuteStatement( &gSrv2, SqlStatement );
// Commit the transaction
hr = pTransaction->Commit( 0, 0, 0 );
// or roll back the transaction
//hr = pTransaction->Abort( 0, 0, 0 );
```

5.3. Password Security

When a DSN is created with the username and password in place, the database password is encrypted and is saved in `odbc.ini`. Alternatively, the login credentials can be entered during the database connection phase or can be passed using the connection string.

5.4. Cursors

In the current Firebird ODBC/JDBC driver, the Dynamic and Keyset cursors are modified to use the Static cursor, through which it is not possible to update sets.

For best performance, use the cursor `ForwardOnly`.

The read operators `SQLFetch`, `SQLExtendedFetch` and `SQLScrollFetch` use `SQL_ROWSET_SIZE` and `SQL_ATTR_ROW_ARRAY_SIZE`.

For best performance using BLOB fields, use the operator `SQLBindParameter`, regardless of the size of the BLOB field, as this will work much faster than using `SQLPutData/SQLGetData`.

To use the Firebird driver's cursors, call the following statements:

```
// Specify that the Firebird ODBC Cursor is always used, then connect.
SQLSetConnectAttr( hdbc, SQL_ATTR_ODBC_CURSORS, (SQLPOINTER)SQL_CUR_USE_DRIVER, 0 );
SQLConnect( hdbc, (UCHAR*)connectString, SQL_NTS, NULL, 0, NULL, 0 );
```

5.4.1. ODBC Cursor Library

This topic is well documented in MSDN. However, we must stress the absolute requirement to use these statements before connecting:

```
// Specify that the ODBC Cursor Library is always used, then connect.
SQLSetConnectAttr( hdbc, SQL_ATTR_ODBC_CURSORS, (SQLPOINTER)SQL_CUR_USE_ODBC, 0 );
SQLConnect( hdbc, (UCHAR*)connectString, SQL_NTS, NULL, 0, NULL, 0 );
```

That data sets keys (?) in the rowset buffers. Updating the buffers requires this statement:

```
SQLFetchScroll( hstmtSel, SQL_FETCH_RELATIVE, 0 );
```

5.5. Stored Procedures

In Firebird, we can have two types of stored procedures, known as *executable* and *selectable*. Both types can take input parameters and return output, but they differ both in the way they are written and in the mechanism for calling them.

- Output from an executable procedure is optional and any output returned is a set of not more

than one “row” of values. If output is defined and none is produced, the output is null.

Returning data is not the primary goal of an executable procedure. Its purpose is to perform data operations that are invisible to the user.

The mechanism for calling an executable procedure is the SQL statement `EXECUTE PROCEDURE`. For example,

```
execute procedure MyProc(?,?)
```

- A selectable procedure is written with the objective of returning a set of zero, one or many rows of data. It can be used to change data, but it should not be written to do that. The PSQL statement `SUSPEND` is used in this style of procedure to pass a row of output that has been collected inside an iteration of a `FOR SELECT..` loop out to a buffer.

  The mechanism for calling a selectable procedure is the SQL statement `SELECT`.

  In this example we have a selectable procedure from which we expect to receive a set of zero or more rows based on the input parameters:

  ```
  select * from MyProc(?,?)
  ```

Microsoft Excel and some other applications use this statement to call a stored procedure:

```
{[? =] Call MyProc (?,?)}.
```

The Firebird ODBC/JDBC driver determines what call to use when executing a stored procedure, from the metadata obtained from the Firebird engine. Firebird flags a procedure as ‘executable’ or ‘selectable’ according to count of `SUSPEND` statements in the assembled (BLR) code of its definition. For a trivial example:

```
create procedure TEST
  as
    begin
    end
```

Because the procedure has no `SUSPEND` statements, the ODBC driver knows to pass the call as `execute procedure TEST`.

For this procedure:

```
create procedure "ALL_LANGS"
   returns ("CODE" varchar(5),
         "GRADE" varchar(5),
         "COUNTRY" varchar(15),
         "LANG" varchar(15))
   as
   BEGIN
     "LANG" = null;
     FOR SELECT job_code, job_grade, job_country FROM job
     INTO :code, :grade, :country
     DO
       BEGIN
         FOR SELECT languages FROM show_langs(:code, :grade, :country)
         INTO :lang
           DO
             SUSPEND;
             /* Put nice separators between rows */
             code = '=====';
             grade = '=====';
             country = '===============';
             lang = '==============';
             SUSPEND;
       END
     END
```

the BLR code for the stored procedure contains more than zero `SUSPEND` statements, so the ODBC Driver will use `select * from "ALL_LANGS"`.

5.6. ARRAY Data Type

To modify single dimension array data type fields, you need to conform to the following rules:

- Specify simple types (`INTEGER`, etc.) as {1, 2, 3}
- Specify string types (`CHAR`, etc.) as {'1', '2', '3'}

TRAPS!

If you edit an element of the array e.g. element 1, 2 and 5, and do not specify the other elements of the array, e.g. 3 and 4, then the other elements of the array will be zeroed (integer), or blank (string).

With some programs where columns are dependent on array data, it is possible to enter array data into a currently `NULL` array column without a validity check being made on the various array elements. Under these circumstances it is essential to enter the array elements before entering the column data.

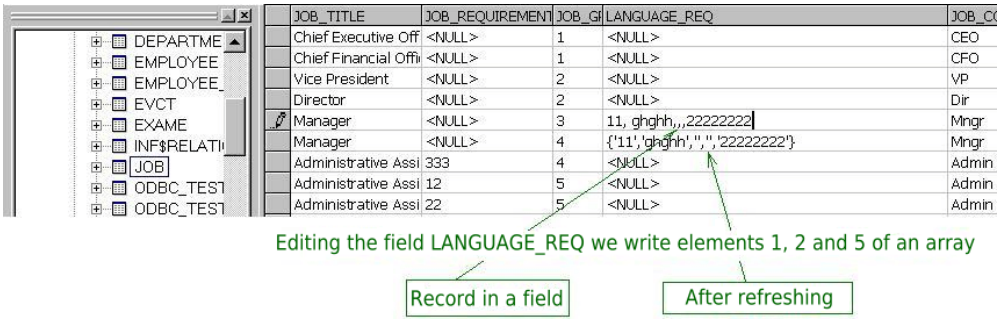

Figure 8. Data loss when updating an ARRAY field (1)

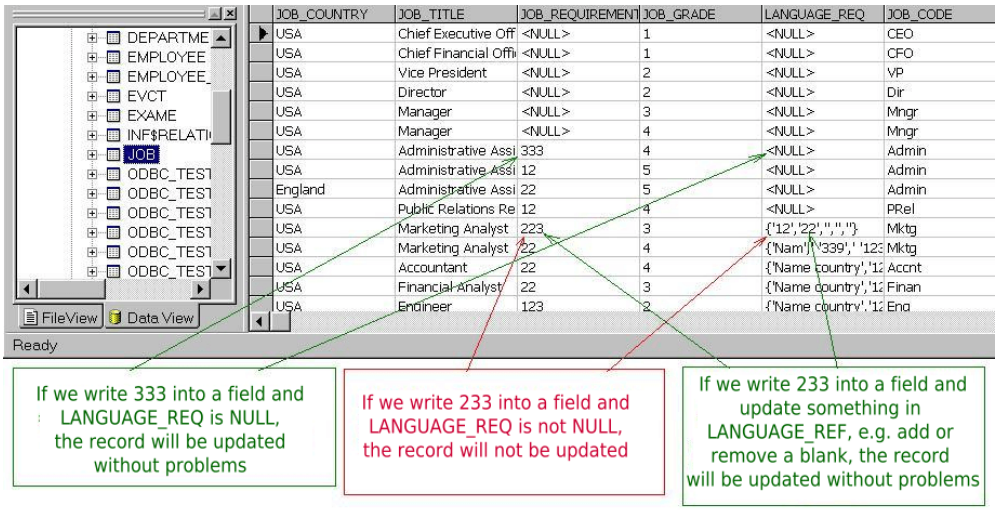

Figure 9. Data loss when updating an ARRAY field (2)

5.7. Usage with Clarion

Jorge Brugger; Vernon Godwin; Vladimir Tsvigun

Clarion users can work with mixed-case object names in Firebird.

1. Create your database in Firebird. You can use table names like `"Pending_Invoices"` and fields like `"Order_Number"`.
2. Create the DSN for the Database, making sure to check all options in “Extended Identifier Properties”
3. Open your dictionary, and import multiple tables as normal from the odbc source. It will work, but do not try to browse or use the files in an application yet.
4. For every field, type in the “External Name” the name of the field surrounded by quotes (for example, type `"Order_Number"` in the external name).

That’s it! Now use your dictionary with mixed case identifiers, without problems. But remember — you must use double quotes around object names in all SQL statements from inside Clarion.

Chapter 6. Firebird Events

To illustrate the use of Firebird events with the ODBC/JDBC driver, we use the example database, `employee.fdb` and work with the `SALES` table. This table has an `AFTER INSERT` trigger `POST_NEW_ORDER` that contains the statement `POST_EVENT 'new_order';`. Its effect will be to signal a listener on the client side when a new record is committed into `SALES`.

Let us suppose that the table has also a `BEFORE UPDATE` trigger that posts an event `'change_order'` in subsequent operations when the field `ORDER_STATUS` is changed.

The trigger `BEFORE UPDATE` does not exist: this scenario is just for illustration purposes, but you could create it if you like:

```
CREATE OR ALTER TRIGGER BI_SALES FOR SALES
ACTIVE BEFORE UPDATE
AS BEGIN
  IF (NEW.ORDER_STATUS = 'new') THEN
  BEGIN
     NEW.ORDER_STATUS = 'open';
     POST_EVENT 'change_order';
  END
END
```

For our demo, we need to insert a new record into `SALES`. The field `ORDER_STATUS` on the newly-inserted record contains the default value `'new'`. After it commits, posting the event `'new_order'`, we want to go back and change something in the new record. When we do so, our `BEFORE UPDATE` trigger, `BI_SALES` will check whether the value of `ORDER_STATUS` is still `'new'` and, if so, it will change it to `'open'` and post the event `'change_order'`.

We are not really interested in how inserting and changing the record affects the database state. The idea here is to show how to prime the driver to manage listening for multiple events.

6.1. Priming the Driver to Listen for Events

The first piece of setting up the driver to listen for events is to connect to an ODBC interface file that describes Firebird events processing:

```
#include "OdbcUserEvents.h"
```

Next, in the table `eventInfo`, we specify the events that we are interested in. For our example, the event `'new_order'` is the only one we are interested in at this stage. The event `'change_order'` is in the picture only to demonstrate the driver's ability to manage multiple events.

```
ODBC_EVENT_INFO eventInfo[] =
{
 INIT_ODBC_EVENT("new_order"),
 INIT_ODBC_EVENT("change_order")
};
```

Now, we need to create a structure — which we will name `MyUniqueData` — to store the data tasks involved in our operation. In our example, a field `event_flag` will signal an event delivered from the server. Our job starts from there.

```
struct MyUniqueData
{
 int event_flag;
 //... other define for use into astRoutine
};
```

We need to create a callback function, `astRoutine`, which will be activated when events defined in the `eventInfo` table are flagged:

```
void astRoutine( void *userEventsInterfase, short length, char * updated )
{
    PODBC_USER_EVENTS_INTERFASE userInterface =
(PODBC_USER_EVENTS_INTERFASE)userEventsInterfase;
    SQLSetConnectAttr( userInterfase->hdbc, SQL_FB_UPDATECOUNT_EVENTS,
(SQLPOINTER)updated, SQL_LEN_BINARY_ATTR( length ) );
    MyUniqueData &myData = *(MyUniqueData*)userInterfase->userData;
    myData.event_flag++;
    printf( "ast routine was called\n" );
}
```

The function needs to have a call:

```
SQLSetConnectAttr( userInterfase->hdbc,
                   SQL_FB_UPDATECOUNT_EVENTS,
                   (SQLPOINTER)updated,
                   SQL_LEN_BINARY_ATTR( length ) );
```

This call is needed for updating the state of events in our structure `eventInfo`. That structure has a field `countEvents` that maintains a total of event operations and a Boolean field `changed` that is set True when the 'before' and 'after' values of `countEvents` are different.

When we want to flag an event that we are interested in, we issue the command:

```
myData.event_flag++;
```

It provides a fairly primitive mechanism for synchronizing workflow, but it is sufficient for our needs. Its setup is as follows:

- At connection time or when the DSN is being constructed, the `NOWAIT` option must be set to `OFF`
- The following statements need to be issued:

```
// Specify that the Firebird ODBC Cursor is always used, then connect.
SQLSetConnectAttr( hdbc, SQL_ATTR_ODBC_CURSORS, (SQLPOINTER)SQL_CUR_USE_DRIVER, 0
);
SQLConnect( hdbc, (UCHAR*)connectString, SQL_NTS, NULL, 0, NULL, 0 );
```

- For the purpose of our demonstration we need to prepare an SQL cursor request. Your own, real-life scenario would be less trivial, of course.

```
SQLPrepare( stmtSel, (UCHAR*)
  "SELECT po_number"
  " FROM sales"
  " WHERE order_status = 'new'"
  " FOR UPDATE",
  SQL_NTS );
```

- We'll construct the cursor query for our demo, naming it 'C':

```
char *cursor = "C";
SQLSetCursorName( stmtSel, (UCHAR*)cursor, sizeof( cursor ) );

SQLPrepare( stmtUpd, (UCHAR*)
  "UPDATE sales"
  " SET order_status = 'open'"
  " WHERE CURRENT OF C",
        SQL_NTS );
```

- Initialize the structure `ODBC_EVENTS_BLOCK_INFO` as the events interface that is passed to the driver:

```
myData.event_flag = 0;
ODBC_EVENTS_BLOCK_INFO eventsBlockInfo = INIT_EVENTS_BLOCK_INFO(
  hdbc, eventInfo, astRoutine, &myData );
SQLSetConnectAttr(
  hdbc, SQL_FB_INIT_EVENTS,
  (SQLPOINTER)&eventsBlockInfo,
  SQL_LEN_BINARY_ATTR((int)sizeof( eventsBlockInfo )) );
- to inform connection, that we are ready to accept events.
SQLSetConnectAttr( hdbc, SQL_FB_REQUEUE_EVENTS, (SQLPOINTER)NULL, 0 );
```

- Events begin ...

```
while ( !iret )
{
  // If the event was triggered, reset the buffer and re-queue
  if ( myData.event_flag )
  {
    myData.event_flag = 0;
    // Check for first ast_call.  isc_que_events fires
    // each event to get processing started
    if ( first )
      first = 0;
    else
    {
      // Select query to look at triggered events
      ret = SQLExecute( stmtSel );
      for (;;)
      {
        ret = SQLFetch( stmtSel );
        if ( ret == SQL_NO_DATA_FOUND )
          break;
        ret = SQLExecute( stmtUpd );
      }
    }
    /* Re-queue for the next event */
    SQLSetConnectAttr( hdbc, SQL_FB_REQUEUE_EVENTS, (SQLPOINTER)NULL, 0 );
    /* This does not block, but as a sample program there is nothing
     * else for us to do, so we will take a nap
     */
    Sleep(1000);
  }
}
```

Chapter 7. The Services Interface

From the configuration page for your Firebird DSN on Windows you have access to a useful graphical management console that is built across the ODBC API and Firebird's Services API. It gives a database administrator on Windows a user-friendly way to run service utilities that would otherwise be run from a command-line tool. We are using it to introduce this chapter because the source code could be a useful resource for developers looking for ideas about including Services functions in their applications.

7.1. Exploring the ODBC Services Console

To use the console, open that configuration page and click the button in the centre, labelled **[Services]**:

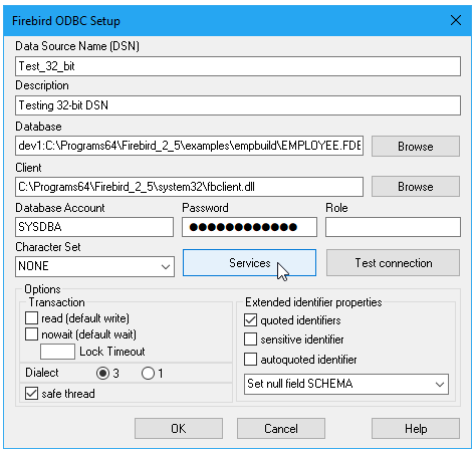

Figure 10. Launching the Services UI on Windows

The console is a tabbed display providing access to many of the Services API functions, with the **Backup** tab on top.

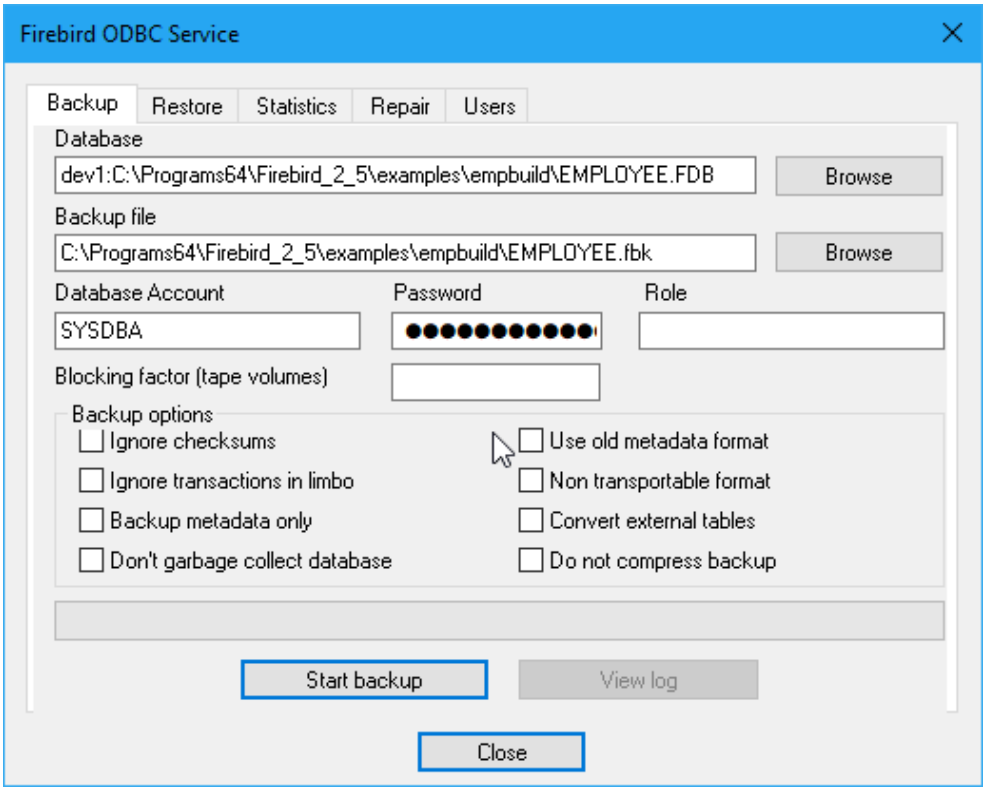

Figure 11. Firebird ODBC Services console — Backup tab

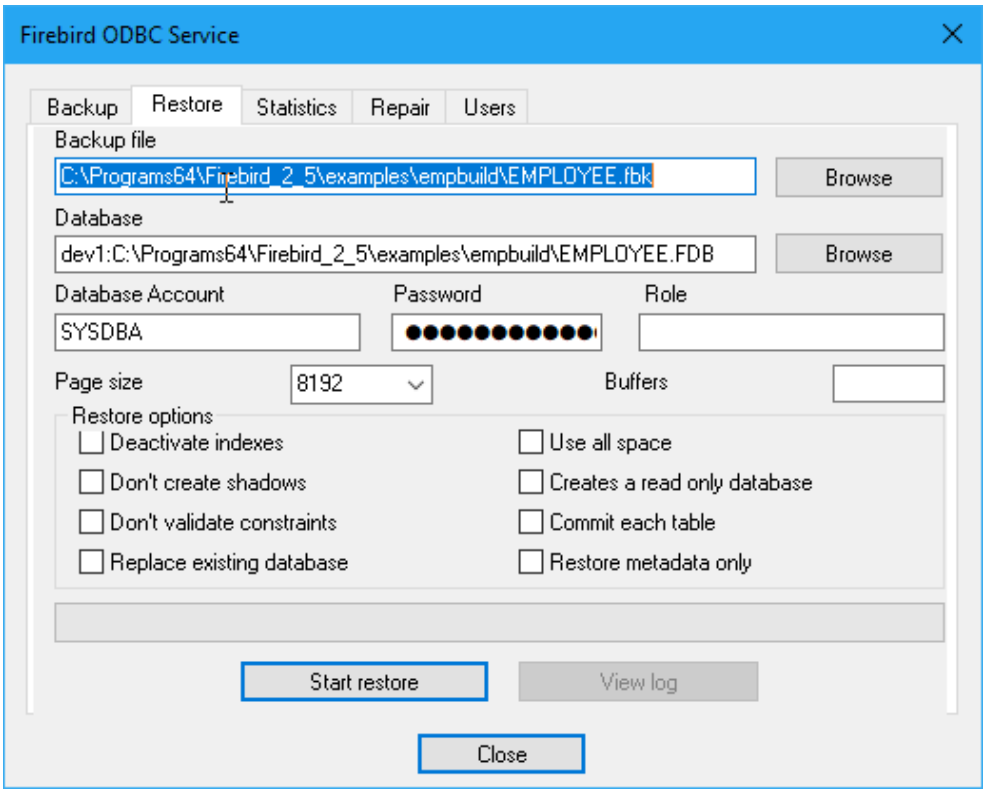

Figure 12. Restore tab

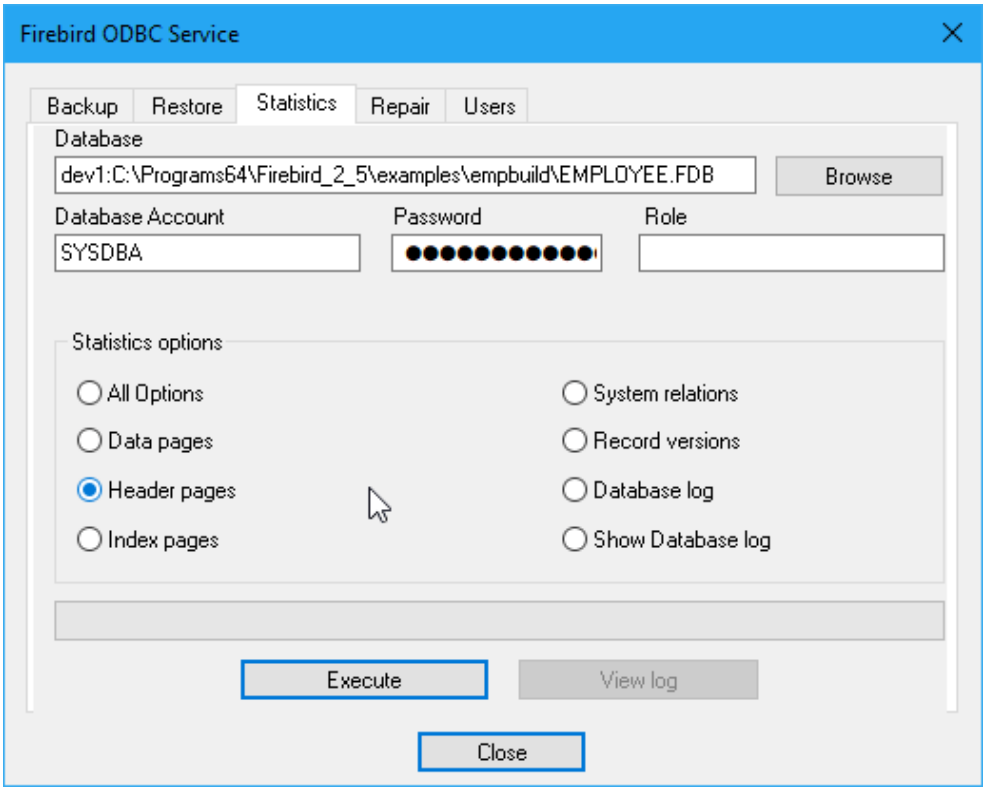

Figure 13. Statistics tab

We selected **Header pages**, which produced the `gstat -h` report for our database. Clicking on the **[View Log]** button delivers the output to the browser:

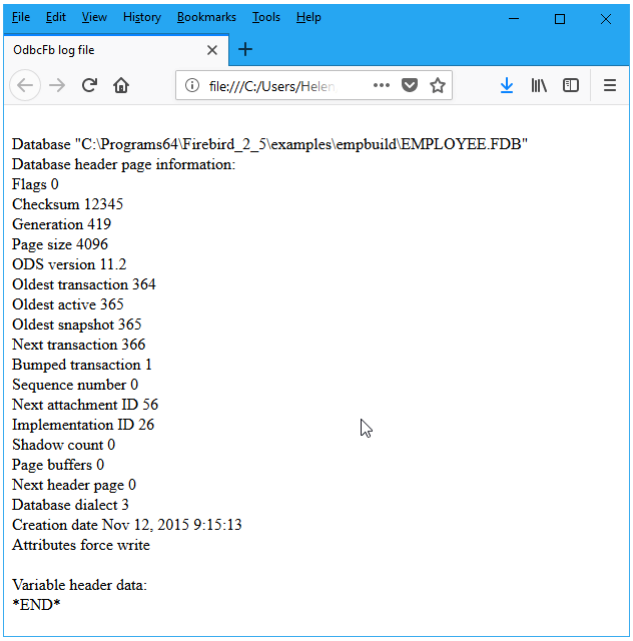

Figure 14. Statistics log

Of course, you can have any statistics report, the Firebird log, metadata reports and more.

The **Repair** tab gives easy access to most of the *gfix* housekeeping functions:

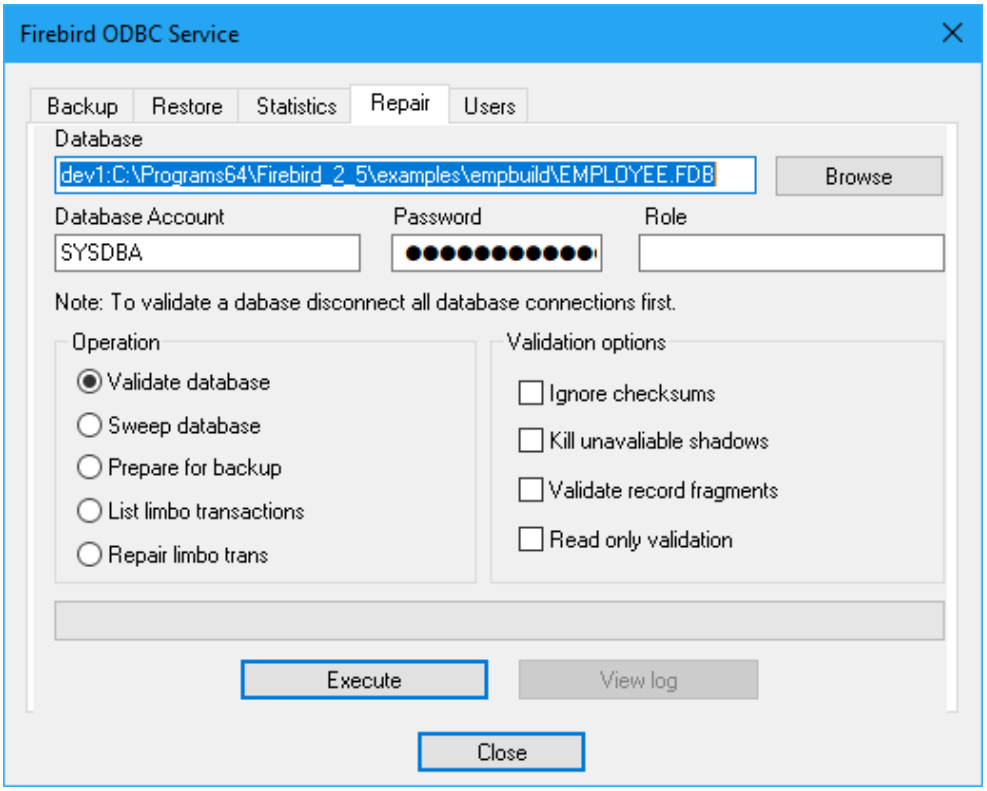

Figure 15. Repair tab

The **Users** tab could be used to maintain accounts in the security database of any version of Firebird prior to version 3.0, although the Services API method was discouraged from V.2.5 onward. The Services API method is still available to maintain users in Firebird 3 databases if they were defined using `Legacy_Auth` authentication management. It will not work with users defined with the default `SRP` authentication manager.

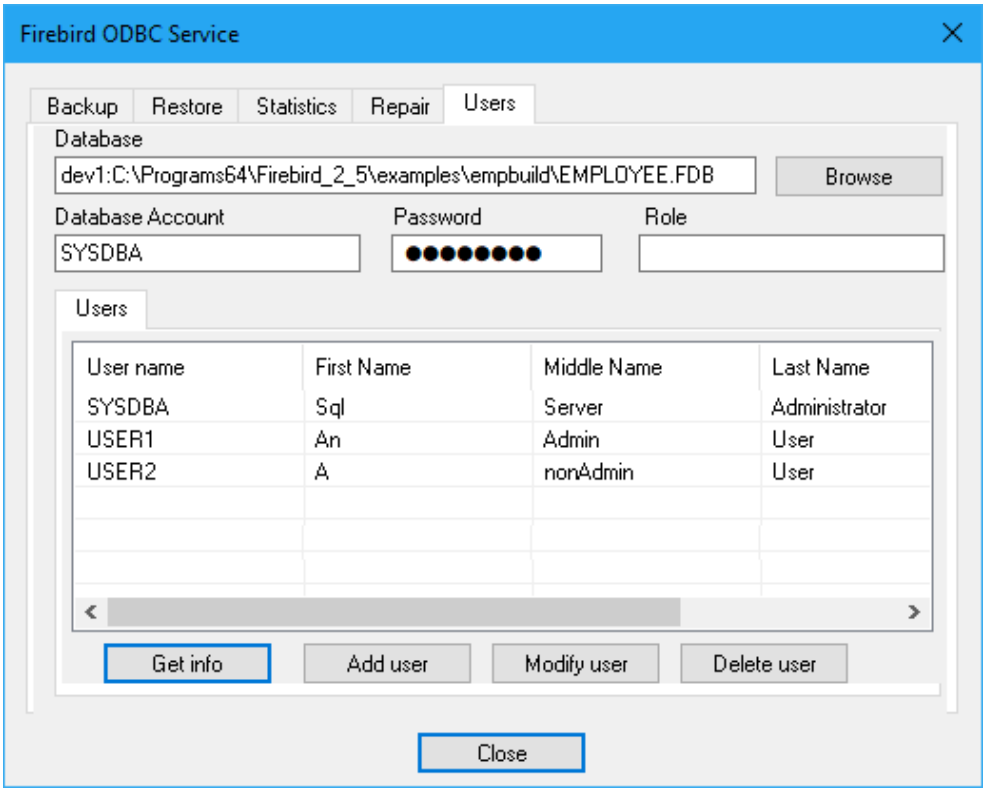

Figure 16. Users tab

Click on the appropriate button to add, modify or delete a user. Remember, the user performing

these tasks must be SYSDBA or a user with elevated server privileges. The role `RDB$ADMIN` is not sufficiently elevated.

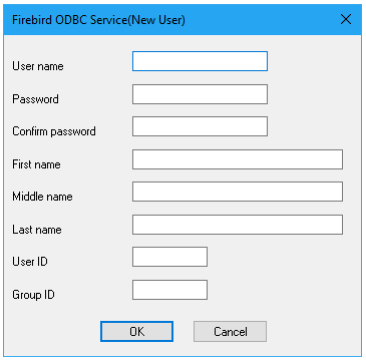

Figure 17. Add user

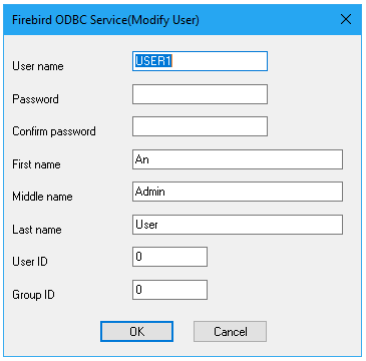

Figure 18. Modify user

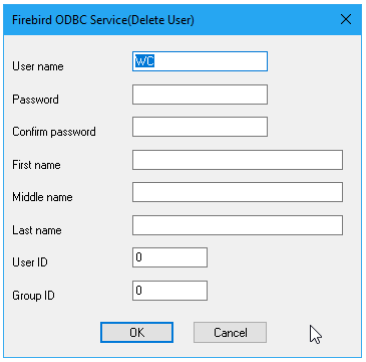

Figure 19. Delete user

7.1.1. Showing Logs from the Interface

If a log file is available from the execution of a Service API function, the **[View Log]** button will become active. The UI provides it on demand in HTML format and opens it in your default browser. If you wonder how to go about coding this into your own ODBC application, the source code is a resource that is freely available to you.

7.2. Using the Services API

The ODBC/JDBC driver wraps a great many of the Services API functions. The management console built into the Windows DSN interface provides examples of most of them. One thing you cannot do via the console is create databases. Fear not! the driver has it wrapped!

In the Connection chapter is a table of the keywords available to signify the values for attachments

via Firebird's "regular" API. The table below provides the keywords for the `KEYWORD=value` parameters for connecting to the server and launching a service request. These are additional to the relevant connection parameters. For some cases, the default settings from the DSN, if used, will be correct for Service requests.

Table 4. Keywords for Service Request Attributes

Keyword	Description	More Information
`BACKUPFILE`	Backup file	This is a filesystem path and file name. Unlike a database, a backup path can be a network storage address.
`LOGFILE`	Path and name of the log file for the service	Optional; valid for any service that provides a log file option. The same filesystem rules apply as for backup files.
`CREATE_DB`	Create database	See the examples below for usage
`BACKUP_DB`	Backup database	The path and name of the database backup file, for backups and restores.
`RESTORE_DB`	The network path and name of the database to which a backup is to be restored. This cannot be a network storage address. The file name part can be an alias, provided the alias exists.	
`REPAIR_DB`	Repair database	Local path to the database to be repaired or validated. Remote access is invalid.
`COMPACT_DB`	Compact database	Not currently applicable to Firebird databases.
`DROP_DB`	Drop database	Not currently applicable to Firebird databases.

7.2.1. Examples of Services Use

The following samples show how to configure the various service requests.

Creating a Database

```
SQLConfigDataSource( NULL,
                     ODBC_ADD_DSN,
                     "Firebird/InterBase(r) driver",
                     "ODBC\0"
                     "CREATE_DB = D:\\TestService\\test.fdb\0"
                     "DESCRIPTION = My Firebird database\0"
                     "UID         = SYSDBA\0"
                     "PWD         = masterkey\0"
                     "CHARSET     = NONE\0"
                     "PAGESIZE    = 8192\0"
                     "DIALECT     = 3\0" );
```

More alternative examples for creating databases are at the end of this chapter.

Backing Up a Database

```
SQLConfigDataSource( NULL,
                     ODBC_ADD_DSN,
                     "Firebird/InterBase(r) driver",
                     "ODBC\0"
                     "BACKUP_DB = D:\\TestService\\test.fdb\0"
                     "BACKUPFILE = D:\\TestService\\test.fbk\0"
                     "UID         = SYSDBA\0"
                     "PWD         = masterkey\0" );
```

Restoring a Database

```
SQLConfigDataSource( NULL,
                     ODBC_ADD_DSN,
                     "Firebird/InterBase(r) driver",
                     "ODBC\0"
                     "RESTORE_DB = D:\\TestService\\testNew.fdb\0"
                     "BACKUPFILE = D:\\TestService\\test.fbk\0"
                     "LOGFILE = D:\\TestService\\test.log\0"
                     "UID         = SYSDBA\0"
                     "PWD         = masterkey\0" );
```

Repairing a Database

```
SQLConfigDataSource( NULL,
                     ODBC_ADD_DSN,
                     "Firebird/InterBase(r) driver",
                     "ODBC\0"
                     "REPAIR_DB = D:\\TestService\\test.fdb\0"
                     "UID         = SYSDBA\0"
                     "PWD         = masterkey\0" );
```

More Ways to Create a Database

Create a database using the ODBC API function `SQLConfigDataSource`. A convenient method for creating a database that is going to be managed by someone else.

```
SQLConfigDataSource( NULL,
                     ODBC_ADD_DSN,
                     "Firebird/InterBase(r) driver",
                     "ODBC\0"
                     "CREATE_DB = D:\\TestService\\test.fdb\0"
                     "DESCRIPTION = My Firebird database\0"
                     "UID         = SYSDBA\0"
                     "PWD         = masterkey\0"
                     "CHARSET     = NONE\0"
                     "PAGESIZE    = 8192\0"
                     "DIALECT     = 3\0" );
```

Create a database using the ODBC API function `SQLDriverConnect`. Convenient when the job is going to be performed from a user application. The driver will handle errors and continue attempting to create the database until it eventually succeeds in connecting to it. Access is passed to the client upon success.

```
UCHAR buffer[1024];
SWORD bufferLength;
SQLDriverConnect( connection, hWnd,
                  (UCHAR*)"DRIVER=Firebird/InterBase(r) driver;"
                  "UID=SYSDBA;"
                  "PWD=masterkey;"
                  "PAGESIZE=8192;"
                  "DBNAMEALWAYS=C:\\Temp\\NewDB.fdb", SQL_NTS,
                  buffer, sizeof (buffer), &bufferLength,
                  SQL_DRIVER_NOPROMPT );
```

Create a database using the ODBC API function `SQLExecDirect`. This scenario is interesting in that the database is created within the context of an existing client connection. It is not necessary therefore to include `"DRIVER=Firebird/InterBase (r) driver;"` in the call, since it will be taken from the current connection.

As with the first method that used `SQLConfigDataSource`, the current user does not get management rights on the database created. For that requirement, `SQLDriverConnect` should be used instead.

```
SQLExecDirect( hStmt,
               "CREATE DATABASE \'C:/TEMP/NEWDB00.FDB\'"
               "   PAGE_SIZE 8192"
               "   SET NAMES \'NONE\'"
               "   USER \'SYSDBA\'"
               "   PASSWORD \'masterkey\';",
               SQL_NTS );
```

Chapter 8. Examples

This page is optimistically left (almost) blank.

```
This space is left for your contributed example.
```

If you have something to offer, please feel free to zip it up and drop it into the Tracker, as an improvement, in either the ODBC or the DOC section.

We would welcome a short description saying what your example demonstrates, in what programming or scripting language and on what OS platform you tested it.

Immortality comes in so many guises.

Appendix A: Licence Notices

Documentation Licence

The Original Documentation is titled *Firebird ODBC/JDBC Driver Manual*.

The Initial Writers of the Original Documentation are Alexander Potapchenko, Vladimir Tsvigun, James Starkey and others.

Software Licence

Appendix B: Document History

The exact file history is recorded in the firebird-documentation git repository; see https://github.com/FirebirdSQL/firebird-documentation

Revision History

1.0.3	MR	27 Aug 2020	Conversion to AsciiDoc, minor copy-editing
1.0.x	??	??	??
0.2	27 Nov 2017	H.E.M.B.	Picked up missing info about multi-threading at the beginning of Ch. 5
0.1	25 Nov 2017	H.E.M.B.	Tidied up old .chm help file and converted to Firebird documentation format.