MWA

*MWA Software*

# IBX for Lazarus User Guide

Issue 1.6,
15 March 2018

McCallum Whyman Associates Ltd

EMail: info@ mccallumwhyman.com, http://www.mccallumwhyman.com

**CONTENTS**                                                                                               **Page**

# 1

# **Introduction**

The *IBX for Lazarus* Guide is a guide to the IBX fork created by MWA Software for Lazarus.

*IBX for Lazarus* is derived from the Open Source edition of IBX published by Borland/Inprise in 2000 under the InterBase Public License. In 2011, the Open Source edition of IBX was brought up-to-date by MWA Software (http://www.mwasoftware.co.uk) and focused on the Firebird Database API for both Linux and Windows platforms (32 and 64-bit), and has since been further developed. It is released under the InterBase Public License for the original code and under the compatible Initial Developers Public License for new software. The Firebird Relational Database Management System can be downloaded from http://www.firebirdsql.org.

While the core of the product remains the original IBX software, this version includes a completely new set of property editors supporting SQL generation and testing using the Firebird Database engine direct from the IDE. These are intended to be a significant improvement on the Delphi Property Editors. IBSQLMonitor has also been re-organised in order to isolate the platform dependent aspects, allowing for the use of SV5 IPC for the Linux environment. The original Windows IPC is retained for the Windows environment. IBEvents has also been updated to ensure compatibility with Firebird Events.

Support for generators has also been added compatible with the generator support added to IBX after the Open Source edition was published, supporting both "On New Record" and "On Post" generators. There are also many new data aware controls distributed as part of the package, plus a scripting engine. TIBExtract has also been brought up-to-date.

From version 2 onwards, IBX uses the *fbintf* package to use either the new Firebird 3 API or the legacy Firebird API. The *fbintf* package is partly derived from IBX and automatically loads the Firebird 3 API, if available, the legacy API if not. *fbintf* is distributed with IBX. The Firebird Pascal API Guide provided with the *fbintf* package provides important information on the installation for the Firebird Server for development system and guidelines for deployment.

See also the Firebird Pascal API Guide for information on:

- Using the API interfaces exposed by IBX

- Character sets and their relation to AnsiString Code Pages

- Deployment of applications using the Firebird Client library.

This Guide assumes that the reader has a basic knowledge of the Lazarus Integrated Development Environment (IDE). Some knowledge of Firebird and database concepts is desireable. However, a primer on the subject is provided (see chapter 3).

## 1.1      References

1. InterBase 6 API Guide (http://www.ibphoenix.com/files/60ApiGuide.zip)

2. Firebird 2.5 Language Reference (http://firebirdsql.org/file/documentation/reference_manuals/fblangref25-en/html/fblangref25.html)

3. InterBase 6 Data Definition Guide (http://www.ibphoenix.com/files/60DataDef.zip)

4. Firebird 3.0.1 Release Notes (http://www.firebirdsql.org/file/documentation/release_notes/html/en/3_0/rlsnotes30.html

5. IBX for Lazarus (MWA Software – http://www.mwasoftware.co.uk/ibx)

6. Firebird Pascal API Guide – MWA Software, 2016

## 1.2      Change History

### 1.2.1    Version 1.1

This version has been updated to include:

- Extended TIBExtract functionality for output of data, including the simple XML formats for binary blobs and arrays, and privileges (grants) given to Triggers and Stored Procedures.

- Extended TIBXScript functionality in order to process XML format data exported by TIBExtract and embedded in INSERT Statements.

- Documentation of Data Output Formatters (see 7.2).

- Minor Typos and corrections.

### 1.2.2    Version 1.2

- Minor typos and corrections

### 1.2.3    Version 1.3

- Introduces TIBUpdate

### 1.2.4   Version 1.4

- Removal of ReadOnly as a common property of IBX TDatasets. This was never true.

### 1.2.5   Version 1.5

- Support for Insert and Update query RETURNING clauses added (see 6.6.1.4)

- Support for Delete query RETURING clauses added (see 6.6.1.5).

- Support for Firebird 3 Identity Columns added (see 6.6.8).

- A new section on IBX TField subclasses is provided as section 6.8.

- The description of TIBStoredProc has been updated to include support for Firebird 3 Packages (see 6.4).

- A new section on the TIBSQL Property Editor has been added (see 5.4.3).

- A new section on row refresh has been added (see 6.6.9).

- The cached updates section has been improved and now describes the use of OnUpdateRecord and OnUpdateError event handlers (see 6.6.7).

- Missing migration issue added to "Upgrading From Earlier Versions" on additional TIBSQL error checks (See 2.6)

- Text reviewed and corrected for typos and other minor errors.

- Removal of TIBTable FieldDefs as a published property.

- Expected_db parameter added to Services API support (see 10.4).

- Additional Database Information properties and function in support of isc_info_active_tran_count, isc_info_creation_date and fb_info_page_contents request items (see 7.5).

- TIBSecurityService support of Admin Role from Firebird 2.5 onwards (see 10.11).

- TIBBackupService and TIBRestoreService now support additional statistics requests in verbose output mode (See 10.5 and 10.6).

- TIBOnlineValidationService added to Firebird Admin Palette (see 10.14).

- PagesFree and PagesUsed properties added to TIBDatabaseInfo (see 7.5)

- A SetNoLinger method added to the Database Configuration Service (see 10.7).

- ImageIndexField and SelectedImageIndexField properties added to TIBTreeView (see 12.3.1).

### 1.2.6     Version 1.6

- Chapter 10 updated for new and re-architectured Services API components. The previous text of chapter 10 is still available as Appendix A.

- New section added on IBX Lazarus Packages (see 4.3).

- TIBDataSet, TIBQuery and TIBTable can now all return performance statistics (see 6.9).

- AlwaysQuoteIdentifiers property added to TIBExtract (see 7.6.1)

- Clarification at end of introduction to chapter 12 as regards the correct place to set query parameter values when using the IBX Dynamic Controls (see also 12.1.5, 12.3.4 and 12.4.4)

- New property MasterDetailDelay added to TIBTable (see 6.3.1) and TIBQuery (see 6.5.1) and TIBDataset (TIBQuery documentation applies to IBDataset).

- Text added describing IBX use in multi-threaded applications (see 2.4).

# 2

# Installation and Preparation for Use

IBX for Lazarus is distributed in a single archive (zip or tar.gz format) and includes the *fbintf* package. You can obtain an up-to-date version from http://www.mwasoftware.co.uk/ibx.

The archive should be expanded into some permanent location on your development system alongside the Lazarus IDE. One possible location is to add a directory called "otherComponents" to your Lazarus installation directory and expand the IBX archive into that directory. IBX will then be located under:

"<Lazarus installation directory>/otherComponents/ibx".

## 2.1    Minimum Requirements

IBX 2.0.0 requires at least Lazarus version 1.6.0 and version 3.0.0 of the Free Pascal Compiler.

All versions of the Firebird Server are supported including version 3.

The Firebird client library must also be installed. If this client library supports the new Firebird 3 client API then this is used, otherwise IBX uses the older Firebird 2 API.

## 2.2    Installation under Lazarus

The Firebird Client Library should be installed on the system prior to installing into the Lazarus IDE. The Firebird Pascal API Guide provides guidelines for installing Firebird.

Installation into the Lazarus IDE is the same under both Linux and Windows. Unpack the source code archive into some suitable permanent location, as described above, and open the "dclibx.lpk" package description file using the "Package->Open Package File" menu item to open the file.

When the Package Editor opens, click on "Use->Install". Lazarus will now recompile itself and restart. THREE new tabs should now be present on the Component Palette: "Firebird", "Firebird

Admin", and "Firebird Data Controls". Respectively, these contain the IBX Database Access and Service API components. A third tab on the palette will contain the "Firebird Data Controls".

If no IBX components are visible, then the most likely reason is that the Firebird Client Library has not been installed and/or cannot be located. See the Firebird Pascal API Guide for information on how *fbintf* and hence IBX finds the Firebird Client Library.

## 2.3 Console Mode IBX

IBX can be used as visual components under Lazarus or in console mode programs. Non-LCL programs should use the ibnongui package, while LCL programs should use the ibexpress package. The ibexpress package uses ibnongui and adds the LCL dependencies (built in login dialog and hourglass cursor on DB operations).

In previous releases, a separate package was provided for console mode programs, excluding any LCL dependencies, called "ibexpressconsolemode". This is still available as a "stub" package that loads the ibnongui and iblocalnongui packages.

If you need to continue to use the separate console mode package in the IDE is to select "Packages->Open Package File" and open ibexpressconsolemode.lpk which you can find in the ibx root directory. You should then close it again immediately afterwards. There is no need to install or compile it. Opening the package is sufficient for Lazarus to remember it.

An example of console mode use is provided in ibx/examples/fbsql.

## 2.4 Multi-threading and Run Time Error 232

IBX can be used in a multi-threading application. However, the underlying Firebird API is not thread safe. Either only a single thread should use IBX or you should ensure that only a single thread at any one time is using an IBX provided function.

Internally, IBX makes use of separate threads for:

- SQL Monitoring (see 7.4)

- Waiting for Firebird Asynchronous Events (see 5.3)

- Implementation of the Master/Detail delay time in TIBTable (see 6.3.2.1), TIBQuery and TIBDataset (see 6.5.1).

If your program uses any of the above then it must include multi-threading support. In Windows and OSX, this is always available. However, under Linux it is only available if you include the "cthreads" unit as the first unit in your program unit. When an application is created from a Lazarus template, this unit is included by adding "-dUseCThreads" to the project's Custom Options.

If multi-threading support is not compiled into your program, then an attempt to use any of the above will result in run-time error 232.

## 2.5 Installing Firebird

You need access to a minimum of the Firebird Client library in order to use the *fbintf* package. This applies to both development and deployment. Guidelines for deployment are give in chapter 13 of the Firebird Pascal API Guide.

On a development system, the recommended approach is to download a pre-compiled installation package from http://www.firebirdsql.org and install the full system including examples. This will ensure that the example "employee" database is both installed and available for use by the *fbintf* testsuite, and a local server is available for testing. Firebird installation packages are available for both Linux and Windows as will as OSX.

With Linux, it is also possible to use the packages provided with your distribution. However, these will not necessarily be up-to-date. Under Debian/Ubuntu the example database is also provided as a separate package and you will need to install this package as well as unpack the database from a gzip archive and set the access permissions correctly before running the test suite. Paradoxically, unless you are very familiar with Firebird and Linux, it is often easier to install the firebirdsql package than the one from your distro.

After installation, you should check that the "employee" is correctly listed in the "aliases.conf" (databases.conf for Firebird 3) file in the Firebird installation folder. For example, with 32-bit Firebird under Windows, the file

```
C:\Program Files (x86)\Firebird\Firebird_2_5\aliases.conf
```

should contain the line:

```
employee = C:\Program Files (x86)\Firebird\Firebird_2_5\examples\empbuild\employee.fdb
```

## 2.6    Upgrading from Earlier Versions

There are many differences between the IBX2 files and earlier versions and you should first either remove or rename the directory containing earlier versions of IBX, and then install the new version as described in the preceding section. Applications using IBX should be rebuilt rather than just recompiled (use Run->Clean up and Build from the Lazarus menu).

IBX2 represents a major change in the underlying IBX codebase. The low level "glue" that represented the language binding between the Firebird 'C' API and native Pascal has been moved into a new package "fbintf" and communication between IBX and this "glue" is now through a well defined Pascal interface. Two implementations of the interface have been produced. One for the legacy Firebird API and another for the new Firebird 3 API. By default, IBX will use the Firebird 3 API, if available, otherwise it uses the legacy Firebird API.

The core body of IBX has been modified to use this new interface. Full support for Firebird Arrays has also been introduced. However, the emphasis has been on maintaining backwards compatibility as far as possible, even though there have been significant changes in the code base.

When migrating an existing IBX 1.4.x application to IBX 2.0.0 most users will need only to recompile against the upgraded package. However, advanced users may need to make changes due to the following incompatibilities:

1.  The IBIntf, IBCodePage and IBXConst units have been removed from the package. Uses clauses that use IBIntf or IBXConst should be replaced with use of the "IB" unit. This is now part of the *fbintf* package and which provides the Firebird Pascal API including all constants and type definitions associated with it.

    IBCodePage was an internal unit providing the mapping between Firebird Character sets and code pages. Equivalent functionality is now provided by the Firebird Pascal API.

2. Any use of the IBHeader unit should be replaced with use of the IB unit. If there is a resulting compile time error after this has been done then, the reason is probably due to a dependency on the legacy Firebird API. IBHeader still exists but it contains the definition of the legacy API and any dependency on it implies a potential problem when IBX uses the new Firebird 3 API. Any such dependency should be identified and replaced with the equivalent functionality provided by the Firebird Pascal API defined in the IB unit.

3. The `TIBSQL` property `SQLType` has been renamed to SQLStatementType. The version roll has been taken advantage of to remove a potentially ambiguous property name. The property name is also used by the input and output metadata to define SQL data types.

4. In IBX2, Automatic transaction Start/Commit is no longer the default except at design time. This may affect some simple uses of IBX with a single dataset on a form and no explicit transaction management. A "transaction no active" error will result when a dataset is opened if your application previously relied on this feature.

   A new property `AllowAutoActivateTransaction` (see 5.2.4) has been added to TIBCustomDataset descendents. By default this is false. If set to true then the original behaviour is restored.

   The version roll has again been taken advantage of to remove a problematic feature. Autostart of transactions only ever worked properly with single dataset applications. With multiple datasets, the order in which the datasets were closed became important (reverse order to opening assumed). With multiple datasets, the transaction could easily remain open after the datasets were closed relying on the database close to correctly perform a transaction completion.

   There could also be problems when explicit transaction start is used ,as the programmer needs to make sure that the transaction was started before any datasets were accessed. Otherwise, unexpected results could ensue.

   On the other hand, it is a valuable feature at design time, allowing a dataset to be opened and its data displayed in the IDE.

   If your application relied upon automatic starting/completion of transactions, the simplest way to restore this behaviour is to set the AllowAutoActivateTransaction property to true.

   If your application has more than one dataset on the form then this property need only be set for the first one that is opened (active property set to true). This dataset should also be the last one closed (active set to false).

5. The UniqueParamNames property is now ignored and exists only for backwards compatibility. Parameter name uniqueness is now determined dynamically.

6. TIBSQL error checking is now more strict. In earlier versions, there were no checks for data validity when (e.g.) accessing query results. In IBX2:

   • An exception is raised if an attempt is made to access query results before the query has been executed, when the cursor is at BOF or EOF, or after the query has been closed.

   • An exception is raised if an attempt is made to set query parameters before a query has been prepared.

## 2.7    New Features with IBX2

- Firebird 3 API Support

- Access to the Firebird Pascal API for embedded SQL execution.

- IBDatabase: new property - `CreateIfNotExists`. If true and the database does not exist when an attempt is made to connect to it (run time only) then an attempt is made to create the database.

- IBDatabase: new event - `OnCreateDatabase`. This event is called after a database has been successfully created as a result of a call to `CreateDatabase` or when creating a database after it was found not to exist.

- Support for arrays has been added. This includes a new field class (`TIBArrayField`) – see chapter 9 - and a supported visual control derived from `TcustomStringGrid`.

## 2.8    Uninstalling IBX

To uninstall IBX, open the "dclibx.lpk" package description file using the "Package->Open Package File" menu item to open the file.

When the Package Editor opens, click on "Use->Uninstall". Lazarus will now recompile itself and restart without the IBX components in the palette. You may now delete the IBX source code.

# 3

# An Introduction to Databases, SQL and Firebird

This chapter is intended to provide a primer on Databases, SQL and Firebird for those not familiar with these subjects. Readers who are familiar with them are invited to proof read this chapter but otherwise, they may prefer to skip to the next chapter.

## 3.1    What is a Database?

The dictionary definition of a database is that a database is no more than a collection of data. It says nothing about how the data is organised or accessed. Some databases can be just a large amount of unstructured data, while others can be fully structured with strongly enforced rules. It is the latter case that we are interested here, and we will leave the former to Google.

The type of database that Firebird manages, and for which the Structured Query Language (SQL) was written, is structured with well defined rules so that they can be processed in a deterministic fashion with repeatable outcomes. This type of database is well suited to business applications, such as accounting and stock management, Personnel Management and Payroll.

### 3.1.1    In the Beginning

In the 1960s and through to the 1980s, Magnetic Tape was the dominate storage medium for big company databases (accounting, stock, etc.). Magnetic Tape is a linear medium accessed sequentially. The data is written to it as "records" and usually ordered using some common relation such as account number or a person's name. Each record contains the data for the account or some person's registration details.

Magnetic Tape databases had to be processed sequentially. It could take a long time to find the record you are interested in, as you had to start at the beginning and work forwards, reading through one or more tapes. Data update was equally laborious with the usual technique being to prepare an update tape with update actions in the same order as the database records and the

application of updates was essentially a data merge between the current set of tapes and the update tape resulting in a set of new master tapes.

### 3.1.2    The Arrival of Random Access Storage

Disk drives started becoming commonplace in the 1970s. Initially they were too expensive to hold complete databases and were used to cache data and to speed up operations. As they become bigger and cheaper, it was possible to start saving entire databases on to magnetic disks.

Magnetic disks can be randomly access. That is any sector on the disk can be accessed in about the same speed as any other. This opened up the possibility of having high speed access to database records and perhaps even *in place* updates. However, there was still the problem of how do you find the record you are interested in? If you still had to start at the beginning and read on until you found the desired record, access would still be slow and variable depending on how far down the data the record was located.

### 3.1.3    Indexes

The answer was to create indexes, where an index is a comparatively small lookup table or tables that may be randomly accessed and could quickly point you at the record you were interested in.

An index is intended for use with a selected key into the data (such as an account number). In principle an index could be just a table of account numbers (the key into the record) and the sector address (on the disk) where the record is held. The index table could then be searched much more quickly than going through the entire database and give much faster access to the data.

For small databases, a simple lookup table is sufficient. However, for large databases, the overhead of searching an index table is still significant and something better is needed. As a result, Indexes became better structured. Perhaps one table for the first part of an account number and then separate tables for the second part, and selected by lookup of the first part of the account number.

Another approach was to generate a hash value from a record key (e.g. the account number) and use that as a numeric table index to where the (e.g.) account number/record address was located. The development of efficient indexes became an important line of research.

### 3.1.4    Multiple Indexes and Datasets

Of course, there was no reason why only a single index was the limit. A database could have many indexes on the same data, one for each access key that you could define. Attention also moved to the structure of the database. The terminology started firming up with the database being broken up into smaller datasets, each with their own index; the sum total of datasets and indexes becoming the database. Given that, in this case, the dataset was a list of identically structured records, they could be modelled as "tables", with each record being a table row and the fields of each record forming the columns.

The original Magnetic Tape databases often contained data duplicated across different databases, if only because it was too difficult to organise the simultaneous processing of multiple tapes. When disks became common, there was value in removing duplication between datasets. This both avoided the risk of differences between data describing the same thing and minimised the use of still expensive disk storage.

However, this did mean the creation of many more smaller datasets and their own indexes and the need for the programs that accessed them to have simultaneous access to many datasets and to "join" the data together.

### 3.1.5    The Need for Middleware

When applications start having to solve common problems there is always an opening for a common middleware solution, and database access was no exception.

Soon many middleware solutions started appearing. Their role was to manage all the different datasets and their indexes and to provide standard ways of joining the datasets and updating the datasets. They freed the client applications from the need to open lots of separate files and instead became a single point of access – the database provider.

### 3.1.6    Enter the RDBMS

The middleware solutions soon evolved into the kind of Relational Database Management Systems (RDBMSs) that we know today. Although the various products have their differences, they can be said to:

- Manage a database comprising many datasets, where each dataset is viewed as a table of data accessed using one or more indexes.

- Maintain metadata (data about data) that describes each table in the database and each index.

- Provide a single point of access to the database to client applications.

- Allow the data to be accessed by table or by joining tables together, using common keys, to create larger virtual datasets (often called views).

- To provide a means to refine the views by limiting both the number of rows returned and the columns in each row. Thereby improving both efficiency and security (by limiting access to data).

- To provide a means to update rows, insert new rows and delete existing rows, including whole table operations.

Throw in performance optimisation, backup and restore, data redundancy (e.g. shadow databases) and you are starting to get something like the modern RDBMS. Some RDBMS servers still maintain each dataset as a separate file (e.g. some versions of MySQL), while others place the whole database in a single file and organise its contents into many files.

### 3.1.7    Multi-user Access

The old Magnetic Tape databases were, by their nature, single user access. However, a modern RDBMS can support a large number of concurrent users all reading data and often updating different parts of the database. In turn, this introduces the risk of conflict between different users concurrently reading and writing to the same record.

Simple table or record "locks" are one way around this problem. A user that wants to update a table or an individual record, first locks it, then updates the data and finally unlocks it. If only one user at a time can create a lock this can ensure that a user can lock all related records they need to

update, update them and finally release them, ensuring that data consistency is maintained. Other users can be prevented from updating those records while a lock is placed on them and can even be prevented from reading them. Users can also be made to wait for a lock to be released.

While a basic record lock mechanism can be viewed as essential for concurrent database update, modern RDBMSs usually go a step further and introduce the idea of a transaction.

Under this model of use, each database client connects to be database via a "Connection" (which can be local or remote), and each connection can have multiple transactions active at any one time, where a transaction:

- Has a well defined start and end, while existing for as long as the client needs it.

- Is the context under which all data is accessed and updated.

- "Owns" any necessary table and record locks, dataset cursors and any other resources used by a client.

- Provides "isolation" between concurrent users, in the sense controlling how much they see of changes performed by other transactions.

- When the transaction ends, all changes made during the transaction can either be committed – that is become changes to the database visible to everyone – or rolled back to their state when the transaction started.

Transactions allow each client to have a consistent view of the database, and a means of preventing data inconsistency resulting from conflicting changes.

## 3.2 The Structured Query Language (SQL)

As discussed above, a basic function of an RDBMS is to provide a means to define and maintain the metadata: the table and index definitions. There is also a need to describe how tables are joined and filtered to create views, both permanent and transient, and for commands to update the database.

This requirement can be satisfied in many different ways. However, SQL has become the *de facto* standard for these tasks. SQL dates back to IBM in the 1970s and provides a means to achieve the above using an English like syntax. It was standardised by ANSI in 1986 and became an international standard in 1987. There was a major update in 1992 (SQL-92) and further minor revisions have taken place since then. While an international standard, each RDBMS has implemented its own variations and hence has its own SQL dialect.

SQL can be split up into:

- The Data Definition Language (DDL), which is used to describe tables, indexes and views i.e. to maintain the database metadata.

- The Data Manipulation Language (DML), which is used to get (select) data from the database, as well as to insert, update and delete data.

- Transaction Management

- • Procedure and Trigger Language (PSQL), which is used to define operations on the database that can be requested by a client or which take place automatically when data is changed. In the latter case, this is often used to validate changes.

The following is an example of the DDL, and is an example of defining a database table:

```
CREATE TABLE EMPLOYEE
(
  EMP_NO smallint NOT NULL,
  FIRST_NAME varchar(15) NOT NULL,
  LAST_NAME varchar(20) NOT NULL,
  PHONE_EXT varchar(4),
  HIRE_DATE timestamp DEFAULT CURRENT_TIMESTAMP NOT NULL,
  DEPT_NO char(3) NOT NULL,
  JOB_CODE varchar(5) NOT NULL,
  JOB_GRADE smallint NOT NULL,
  JOB_COUNTRY varchar(15) NOT NULL,
  SALARY numeric(10,2) DEFAULT 0 NOT NULL,
  PRIMARY KEY (EMP_NO)
);
```

The above defines the table as consisting of ten columns. Each column is given a name and a data type. The table's primary key is the employee number (EMP_NO) and this provides a unique identifier for each row.

Note that SQL statements are always case insensitive including column names (although later extensions have allowed for column names that are case sensitive and which may include special characters by enclosing them in double quotes).

The table definition also includes an important concept that has not been discussed so far and that is the concept of the "NULL" value. Unless constrained to be "NOT NULL" as illustrated above, the values in each row of a table column can be either a value in their declared type or have no value (i.e. null). NULL values can be searched for, can be used to select data, and data values can be set to NULL. The Firebird Null Guide provides more information on the use of Nulls.

An example of a Select Statement follows. This creates a temporary virtual dataset which can then be read by the requesting client.

```
Select FIRST_NAME,LAST_NAME,EMP_NO,HIRE_DATE FROM EMPLOYEES
  Where LAST_NAME LIKE 'P%';
```

The dataset returned by the above has only four columns and is filtered so that it comprises only the employees whose last name starts with the letter 'P'.

Note: in SQL comparisons, the '%' character means any string.

## 3.3    The Firebird RDBMS

Firebird is an example of a Relational Database Management System (RDBMS). It is an Open Source product with a permissive licence for use that includes use in commercial applications. It can also require very little, if any, input from a System Manager and hence is well targeted on SME applications. Although it generally scales well to larger applications, as well.

Firebird came about when Borland/Inprise released the InterBase 6.0 software under an Open Source Licence in 2000. InterBase already had a long history (documented on

) and Firebird inherited a large user community from InterBase.

In its modern instantiation, Firebird:

- is a multi-user, transaction based RDBMS

- uses SQL for Data Definition, Data Manipulation, Transaction Management and Procedure and Trigger Definition.

- Deployed as either an embedded database engine (embedded server) or as a standalone server accessed using TCP/IP supporting both local and remote connections.

- Uses a single file per database (with the option of secondary files to allow for overflow to separate filesystems).

- Implementation packages are available for many platforms including Windows (32 and 64 bit), Linux (32 and 64 bit) and OSX.

Firebird also includes the concept of "Events". That is asynchronous alerts that are PSQL generated and which can be sent to an interested client. These are typically used from triggers to alert other users to changes in the data.

## 3.4    And then there was IBX

Firebird provides a client library (DLL under Windows, shared object (.so) under Linux) through which an Application Program Interface (API) is made available. This API is "low level" providing a basic set of functions with the data accessed through untyped pointers. This is a flexible approach, allowing use from many different programming environments, whilst requiring work from the client program to make sense of the data.

For 'C' programs, Firebird provides a pre-processor (gpre) that allows SQL statements to be embedded into the code and which then generates the 'C' code necessary to pass those statements to Firebird for execution and to pass input and receive output data to and from 'C' data structures. However, no such pre-processor is available for Pascal.

When Borland released Delphi in the mid-nineties, it was arguably a revolutionary development in visual programming. It also came with a model for database programming that included an abstract model of a dataset and "data aware" controls. That is a means to link the controls (or widgets) placed on forms with the fields in a dataset. This all works well as long as the abstract dataset can be somehow made "concrete" and linked to the datasets provided by the RDBMS that the developer wants to use.

The first versions of Delphi worked well with Paradox tables and included middleware known as the Borland Database Engine (BDE) to provide drivers for SQL databases, including InterBase (a trial version was shipped with Delphi). The BDE was probably not the most efficient solution for the problem.

One improvement on this was the Free IB Components, written by Gregory H. Deatz for the Hoagland, Longo, Moran, Dunst & Doukas Company. This was licensed by Borland and then provided with Delphi as InterBase Express (IBX).

IBX provided a direct implementation of the abstract dataset model for InterBase, using the API direct from a Delphi (Pascal) program. It allowed the programmer to define the datasets using SQL

and to update data using SQL, whilst keeping within the Delphi dataset model. By making direct use of the InterBase API it potentially gives the best performance possible to a Delphi program.

When Borland/Inprise released InterBase under an Open Source licence in 2000, it also released the IBX codebase under the same licence. In 2011, a fork of the IBX Open Source release was used by MWA Software to create *IBX for Lazarus*. This version of IBX was developed to work with the Lazarus LCL and to use the Firebird RDBMS in either embedded or standalone server mode.

Since its original release, *IBX for Lazarus* has been both maintained and extended with the introduction of additional components including an SQL Parser and script engine. In 2016, IBX2 included support for the new Firebird 3 API. This version includes a separate set of Pascal Language Bindings (the *fbintf* package) that provides the foundation for IBX2. The *fbintf* package can be used to effectively embed SQL statements within Pascal code.

# 4

# IBX Overview

The purpose of IBX is to provide an implementation of the TDataset model, and hence a data source for Data Aware components, and doing so by making direct use of the Firebird API. There is no middleware involved and the intent is to maximise performance. IBX is intended to provide the best performance possible when using Firebird from a Pascal program.

Firebird is an SQL database and a knowledge of SQL is generally necessary for all but basic use of IBX. IBX does not attempt to hide the SQL from the programmer[1]. Indeed, it gives the programmer full use of SQL.

The *IBX for Lazarus* components should behave identically to their Delphi equivalents and many online tutorials are available on how to use them. An introduction to their use is given below, and many example programs are also provided.

## 4.1    Conversion From Delphi IBX

You should be aware of the following issues:

1. The IBX components make use of the `TThread` class, and, as such require that multi-threading is enabled. Specifically, in the Linux environment, the "-dUseCThreads" option must be present in the "Compiler Options->Options->Custom Options" and set for every Lazarus project that uses them.

2. Prior to FPC 2.6.0, the `TIntegerField` type may cause problems when porting code from Delphi to Lazarus.

3. FMTBcd is not yet implemented by the Free Pascal Compiler. IBX for Lazarus thus uses the `TFloatField` type for extended floating point (64 bit) fields. This may cause problems where converting Delphi programs to Lazarus. The recommended approach is to change all

---

[1]The exception to this is the TIBTable and TIBStoredProc components. These can be used for simple database applications without requiring any SQL programming.

`TFmtBcdField` types to `TIBBcdFields`. This will allow Delphi forms to be converted to Lazarus. However, some of the conversions will not give the correct results. Typically, this will result in field values that appear to be of the order of several billion when the program is run. To resolve the problem, delete the field in the IDE Fields editor and then re-create it. The correct field type will then be used.

Alternatively, all `TFmtBcdField` fields should be deleted prior to conversion and then recreated in the IDE.

## 4.2    IBX in Context

The following diagram attempts to position IBX with respect to other packages. As illustrated the *fbintf* package is the provider of the Firebird API and may be both directly access by the user, while also providing the Firebird API to IBX. IBX is also accessed directly by the user, but also uses the FCL which is where the `TDataset` abstract class is located. IBX can make use of the LCL but only does so when not in console mode and this is only to provide the built-in Login Dialog.
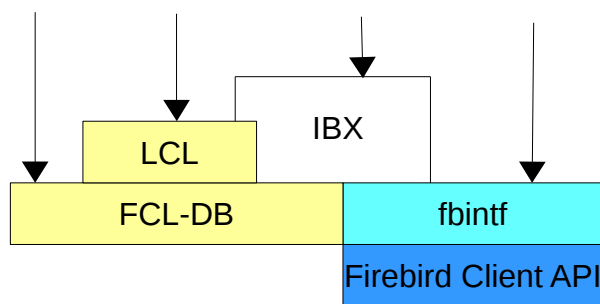


**Illustration 1: How IBX Relates to other Packages**

## 4.3    IBX Lazarus Packages

IBX is deployed as a set of Lazarus packages, organised as follows:

- ibnongui            This is the base set of IBX components and comprises all common IBX components that do not require the LCL. Use this package in console mode applications.

- ibexpress          This packages depends on ibnongui and adds the LCL dependent units. These provide the built in login dialogs and the "hour glass" cursor support for time consuming database operations. Use this package for LCL based applications.

- ibLegacyServices   This package depends on ibnongui and adds the legacy services API components contained in the IBServices unit (see Appendix A.). This may be used in both CLC console mode applications.

- iblocalnongui      This package depends on ibnongui and adds the Personal Database support components (see chapter 11) for console mode applications.

- iblocal      This package depends on ibexpress and iblocalnongui and adds the Personal Database support components for LCL applications.

- ibcontrols      This package depends on ibexpress and adds the IBX Controls components (See chapter 12).

- dclibx      This is the design time package used to install IBX into the Lazarus IDE.

- ibexpressconsolemode   In previous versions, this package was used by console mode applications using IBX. It is still available for legacy use and depends upon both ibnongui and iblocalnongui.

In the most cases, it is not necessary to explicitly add an IBX package to your project as the Lazarus IDE does this for you when you "drop" an IBX component on to a form in your project. Only when developing console mode applications will you need to explicitly add one or more of the non GUI packages to your project dependencies.

## 4.4    Component Overview

The following components are installed on the Firebird tab:

**TIBDatabase**      Every project that uses IBX must have at least one `TIBDatabase` component. This is usually placed on a data module or the main form, and represents the connection to the database. Its properties identify the server on which the database is located, its name or pathname on that server, the login credentials, and the local character set when transliteration is required. It can also generate a login prompt for the user name and password, or support a user provided login form. See 5.1.

**TIBDatabaseInfo**      This component supports a `TIBDatabase` and provides read only access to a database's properties and statistics. See 7.5.

**TIBTransaction**      Every project that uses IBX must have at least one `TIBTransaction` component. Firebird is a transaction oriented database and all operations must take place in the context of a transaction. Its properties determine the transaction isolation (see Firebird documentation). A `TIBTransaction` is typically provided with the `TIBDatabase` and linked to it by the `TIBDatabase DefaultTransaction` property. See 5.2.

**TIBQuery**      This component is a descendent of TDataset and generates the dataset from the results of an SQL query (Select statement or a

Stored Procedure that returns a results set). The SQL query used is given by its SQL property. This can be parameterised with the values of the parameters set before the query is executed. When the "active" property is set to true then the query is executed and the results set returned. When "active" is set to false, the results set is discarded. The dataset is read only unless its "UpdateObject" property references a TIBUpdateSQL or a TIBUpdate object.

The TIBQuery's properties must identify the database and the transaction used for executing the query. See 6.5.

**TIBUpdateSQL**

This component may be referenced from a TIBQuery component and is used to support updateable queries. It provides SQL statements to:

- Delete the current row in the results set

- Refresh (from the database) the current row in the results set

- Update the current row in the database to match the (modified) values in the database

- Insert a new row into the database.

See 6.6.

**TIBUpdate**

This component may be referenced from a TIBQuery component and is a more general way to support updateable queries than that provided by TIBUpdateSQL. While TIBUpdateSQL supports single SQL statement for Delete, Update or Insert, TIBUpdate provides an event handler for Update, Insert or Delete together with an ISQLParams interface providing access to all current and "old" field values. This gives the programmer complete freedom as to how the Update, Insert or Delete is performed.

**TIBDataSet**

This is also a `TDataset` descendent and combines the functionality of `TIBQuery` and `TIBUpdateSQL` into a single component. See 6.7.

Its properties must identify the database and the transaction used for executing the query.

You will normally want to use `TIBDataset` instead of a TIBQuery and TIBUpdateSQL pair. Alternatively, the latter combination may be used, for example, when a form uses a TIBQuery to provide a read only dataset, and a subclassed (inherited) form needs to update the dataset. The IBUpdateSQL can be added to the subclassed form to provide the update capability.

**TIBStoredProc**

This component is used to execute a stored procedure (on the Database Server), and one that does not generate a results set. Its properties must identify the database and the transaction used for

executing the query. See 6.4.

**TIBSQL**

This component is the basic SQL engine of IBX and is used internally by `TIBQuery`, `TIBDataset` and `TIBStoredProc` to perform SQL queries. It can be used directly by the programmer to effectively implement embedded SQL statements.

Its properties must identify the database and the transaction used for executing the query.

`TIBSQL` is essentially an object oriented encapsulation of the Firebird DSQL API. See 6.5.

**TIBEvents**

One very useful feature of the Firebird Database is its ability to generate asynchronous "events" from a Stored Procedure or Trigger and which can then be acted upon by any active client that is listening on the event. Database clients can thus act immediately on changes made by another client without needing to regularly query the database.

The `TIBEvents` component is used to register for and receive Firebird Events. Up to 16 events can be waited upon simultaneously. The name of each event to be listened to is set in the component's Events property. If you need to wait on more than 16 events, then additional `TIBEvents` components can be used.

The event notification is asynchronous and takes place at the end of the transaction in which the event was generated. A separate thread is used by `TIBEvents` to wait on the event notification. When the event occurs it calls the "`OnEventAlert`" event handler to report which event has been received. Note that the event handler is run in the context of the main thread and hence there is no need to worry about thread synchronisation. See 5.3.

**TIBSQLMonitor**

This component supports debugging and performance tuning by allowing one process to monitor the SQL function calls in the same or another process (on the same system).

The `TIBDatabase` trace flags determine which function calls can be traced with respect to its database connection. However, a process only starts to broadcast its function calls after an explicit call to the `IBSQLMonitor.EnableMonitoring` procedure, and stops after a call to `IBSQLMonitor.DisableMonitoring`.

To receive SQL Function call traces, you need to place a `TIBSQLMonitor` component on your form. The properties of this component can be set to filter the SQL function calls to what you are interested in. The OnSQL event handler is used to receive and process SQL function call trace events. See 7.4.

> Note that the Windows implementation allows any process to monitor the SQL Trace events broadcast by another. The Linux implementation restricts monitoring to processes owned by the same user.

| | |
|---|---|
| **TIBTable** | This component provides a simple `TDataset` descendent where the contents of the dataset are the same as a named Database Table. This component is useful for very simple applications, but `TIBDataset` should normally be preferred for most applications. It also supports Master/Detail relationships between linked tables. See 6.3. |
| **TIBExtract** | This component allows the extract of database metadata. The component is intended to be compliant with all Firebird extensions to the DDL up to and including Firebird 3. See 7.6. |
| **TIBBatchMove** | This component supports a table to table copy from a source IBX dataset to a `TIBTable`. |
| **TIBXScript** | This component is used to run an SQL script in the specified file or stream. The text is parsed into SQL statements which are executed in turn. The intention is to be ISQL compatible but with extensions. See 7.1. |

The Firebird Admin tab provides the Service API components. These support various server side functions including user password maintenance and database backup/restore (see 10.2).

## 4.5    Databases and Transactions

IBX always access a database through a "connection" whether it is a local database or a remote one. Data is the read and written in the context of a transaction. Transactions exist:

- to isolate users from each other,
- to allow a user to see a consistent view of the data independent of what other users are doing,
- to allow users to lock data for change and to control whether other users wait on such locked data, and
- to provide a well defined transaction start and end where, at a transaction end, any changes made to the data are either committed and made available for other users to see, or are abandoned and the data rolled back to where it was before the transaction started.

Firebird is a transaction oriented database and all interactions with the database have to take place in the context of a transaction. Firebird allows multiple independent transactions to take place simultaneously and provides several possible isolation strategies in order to avoid the transactions interfering with each other.

At least one `TIBDatabase` (see 5.1) and one `TIBTransaction` (see 5.2) component are required for an application that uses IBX. These are normally placed on the project's main form or on a data module. The `TIBDatabase` component represents the connection to a database, and it's rare than a project needs more than one such component (e.g. if you need to support simultaneous connections to two or more databases).

The `TIBDatabase` component's properties identify the location of the database and provide the logon parameters. It can also use a built in dialog to prompt for a user name/password or use an application provided dialog. A default transaction is also identified in the `TIBDatabase` properties. Datasets linked to this database automatically use the default transaction unless this is explicitly overridden in the dataset's properties.

`TIBTransaction` represents a transaction, and, you can have as many `TIBTransaction` components as necessary. A `TIBTransaction` is normally linked to single database (`TIBDatabase`) but can be linked to multiple databases in order to synchronise updates to them.

## 4.6    Datasets

The TDataset model is integral to the managing the relationship between Data Aware controls and database tables in both Lazarus and Delphi. A `TDataset` derived component is used, at its simplest, to represent the data in a single database table. Data aware controls, such as `TDBEdit` can then be linked to a single (text) column in the dataset and allow the data in the current row of that column to be both displayed and edited. A control such as TDBGrid allows multiple rows to be shown together and as a table.

An example of this is shown in Illustration 2. This is a snapshot from the example application (see the "ibx/examples/employee" directory) and illustrates the use of `TDBGrid` to show a table of database data.

### 4.6.1    Datasets and Transactions

Each dataset is linked to a single transaction and all data reads and writes using that dataset take place in the context of this transaction.

In a simple application, you only need to include a single `TIBTransaction` component with your application.

- In the most basic case, the first dataset to be activated implicitly starts the transaction (as long as its `AutoActivateTransaction` property is set) and, if, when it is deactivated, no other datasets are active, it will automatically commit the transaction. When the database is closed, the default action of the `TIBTransaction` is to commit all changes.

- In a more advanced application (such as the example application), you will want more control over committing or rolling back the transaction and `TIBTransaction` provides methods to start a transaction and to commit it or roll it back. In this case, it is usually advisable to explicitly start each transaction rather than relying on implicit starts as this avoids a dataset unexpectedly committing a transaction when it is closed. Commit and Rollback are then always under programmatic control.

Note that when a transaction ends, all datasets referencing the transaction are automatically deactivated. A new transaction has to be started and those datasets reactivated if they are to continue to be populated.

**Illustration 2: Employee List from the examples/employee application**

However, it is possible to save changes (commit the transaction) and to not close the datasets. This is by using the `TIBTransaction.CommitRetaining` method. This commits the transaction whilst retaining the transaction context. The datasets can thus remain open. The downside of using this function is that in a multi-user database, the datasets only pick up changes made by other users when they are closed and re-opened.

### 4.6.2 Single Table Datasets

While a single table may be used for simple applications, the dataset is more normally the result of a database query where the result is one or more rows. In some cases, the dataset may just consist of a single row where it is supporting a form that edit's that row's contents. This avoids the overhead of reading many rows from a database when only a single one is needed.

In IBX, the `TIBTable` component is a `TDataset` descendent that allows a single table to be named and, on your behalf, it does all that is necessary to read the rows in the table and update them as required. It does this by automatically generating the SQL Statements needed to access and update the database. However, this is a limited approach, and does not allow the full power of SQL to be exploited.

### 4.6.3 SQL Defined Datasets

IBX also provides `TIBQuery`. This is also a `TDataset` descendent but reads a dataset that is the result of an SQL Select Query specified by the programmer either a design to runtime. The query can join multiple tables and be limited to just the rows required. It is much more powerful than just accessing a single table.

If you need also to update the database after editing a `TIBQuery` results set, another component (`TIBUpdateSQL`) can be linked to the `TIBQuery` to provide the SQL Statement necessary to Update, Insert or Delete rows, or to refresh rows that may have changed.

More generally, `TIBDataset` is a component that allows you to specify the Select, Update, Insert, Delete and Refresh queries in one component.

## 4.7    Examples

The ibx/example/employee example application provides an example of the use of the `TIBDatabase, TIBTransaction, TIBQuery` and `TIBDataset` components supporting data aware components for viewing and editing data from a database.

# 5

# The Database Access Components

The IBX components are presented here in three main groups. This chapter is concerned with the "Database Access" set of components. Chapter 6 is concerned with the dataset components, while chapter 7 presents support components.

The IBX Database Access components are simply those components that are little more than "wrappers for interfaces exported by the *fbintf* package, and are:

- TIBDatabase – encapsulates the IAttachment interface
- TIBTransaction – encapsulates the ITransaction interface
- TIBEvent – encapsulates the IEvents interface
- TIBSQL – encapsulates the IStatement interface.

The Firebird Pascal API describes each of these interfaces in detail and how to use them. However, when using, you should use the methods and properties provided by the component in preference to similar methods and properties provided by the underlying interface, except as described below.

## 5.1    TIBDatabase

This is a non-visual component and a `TCustomConnection` descendent. It provides the link between the database connection as viewed by the `TDataset` model and the Firebird database connection.

### 5.1.1    Highlighted Properties

| | |
|---|---|
| `Connected` | Set this property to true to connect to a database and to false to disconnect. |
| `AllowStreamConnected` | This property exists to avoid a conflict between connecting to a database at design time and automatically connecting at run |

| | |
|---|---|
| | time. (see 5.1.10). |
| `CreateIfNotExists` | New in IBX2: If true, and an attempt is made to connect to a database that does not exist, then the database is created, if possible. See also the `OnCreateDatabase` event. |
| `DatabaseName` | The pathname to the database includes the server name. The format varies depending on the connection protocol used and the way pathnames are expressed on the target system. See the Firebird documentation for more information. Note: from IBX 2.2 onwards, local database names can include simple macros. See 5.1.2. |
| `DefaultTransaction` | Reference to the default transaction for the database. This is purely a convention. When a dataset is linked to a `TIBDatabase`, it's default transaction is assigned as the dataset's transaction if none is already specified. |
| `IdleTimer` | If non-zero, this is the time in milliseconds between successive polls for database activity. If no activity has been detected between two successive polls then the database is automatically disconnected. Can be used to timeout idle connections if needed. |
| `LoginPrompt` | When true, the a login dialog is shown to the user to confirm the database user name (as given in the `Params`) and to enter a password. If the `OnLogin` event handler is defined then this is called and is expected to generate the login dialog. Otherwise, the built-in login dialog is used. |
| `Params` | This is a list of parameter values to be used in the Database Parameter Block (DPB) at connect time. These are in "keyword=" format. See below. |
| `SQLDialect` | The default SQL dialect to be used for when creating a database (3 is recommended). See Firebird Documentation. |
| `SQLHourGlass` | If true, then the cursor is changed to an Hour Glass (or equivalent) during calls to the database server. |
| `UseSystemDefaultCodePage` | If true, then the system default code page is used as the connection default character set. Not recommended for Lazarus programs where UTF8 is assumed by many LCL functions. |

### 5.1.2 DatabaseName Macros

In order to aid the development of cross platform applications that use local databases in well known locations, from IBX 2.2 onwards, the DatabaseName property value can include simple macros used as prefixes for database names.

The following prefixes are available:

- $TEMP$

  Before connecting to a database, this is replaced with the local system's temp directory (including trailing delimiter).

- $DATADIR$

  Before connecting to a database, this is replaced with a prescribed data directory (including trailing delimiter).

Under Unix systems the data directory is a hidden directory in the user's home directory. The hidden directory name is either the string returned from the SysUtils "VendorName" or "IBX" if empty. In either case prefixed by a '.'.

Under Windows, the directory is either the string returned from the SysUtils "VendorName" or "IBX" if empty. In either case prefixed by the User's application data path.

For example: "$TEMP$1DTest.fdb" is expanded on Linux to "/tmp/1DTest.fdb".

Note: These macros are used in the IBX examples to avoid platform dependencies.

### 5.1.3 Parameter Keywords

The parameters are best edited at design time using the database dialog editor (double click on the `TIBDatabase` component icon, once it has been placed on your form). The parameter names available include:

| user_name | Login user name |
|-----------|-----------------|
| password  | Login password (use of this parameter to save a password at dedign tiem is not recommended). |
| lc_ctype  | Name of the connection default character set (e.g. UTF8). UTF8 is recommended for Lazarus programs. |

### 5.1.4 Highlighted Events

TIBDatabase events are typically used to react to changes in the connection state. These include:

| AfterConnect | Called after a connection has been successfully established. A good place to start the first transaction and open datasets. |
|--------------|---------------------------------------------------------------------------------------------------------------------------|

| AfterDisconnect | Called after a connection has been disconnected. |
| --- | --- |
| BeforeConnect | Called before an attempt is made to connect to a database. Could be used to update the connection parameters, database name, etc. |
| BeforeDisconnect | Called before an attempt is made to disconnect from a database. |
| OnCreateDatabase | Called after a database has been successfully created. Could be used to run a DDL script (e.g. using TIBXScript – see 7.1) to initialise the database. |
| OnIdleTimer | Called after an idle timer has disconnected the database. |
| OnLogin | Called if LoginPrompt is true and maybe used to complete the login parameters by prompting the user. If not set and LoginPrompt is true then the built in login dialog is used. |

## 5.1.5  Connecting to a Database

The recommended approach is:

- To set AllowStreamedConnect to false
- Do not include a password in the parameters
- Use the built-in or a user defined login prompt to confirm the user name and enter the password.

The following code is recommended for setting the connected property to true. This may be part of an OnShow handler for the main form or when otherwise required:

```
repeat
  try
    IBDatabase1.Connected := true;
  except
   on E:EIBClientError do
    begin
      Close;
      Exit
    end;
   On E:Exception do
    MessageDlg(E.Message,mtError,[mbOK],0);
   end;
until IBDatabase1.Connected;
```

An example of this code in use may be found in "ibx/example/employee". The purpose of the above is to trap and report errors, such as mis-typed passwords whilst allowing the user to click on cancel and exit through a "client error".

Note: you will need to add the "IB" unit to your units list in order for the above to compile.

### 5.1.6    Database Disconnect

You can disconnect from a database at any time by setting the Connected property to false. However, it is not necessary to explicitly disconnect before a program terminates as this is performed automatically when the `TIBDatabase` component is destroyed.

### 5.1.7    Creating a new Database

The `CreateDatabase` method can be used to explicitly create a new database, alternatively, if the `CreateIfNotExists` property is set to true then a new database is automatically created if the connection attempt fails with a database not found error.

The `DatabaseName` property is used to determine the location of the database and the `Params` property provides the create parameters, including the database owner (user_name), and the database default character set (lc_ctype). The `SQLDialect` property determines the database SQL Dialect.

Alternatively, the `CreateDatabase` method may be called with a "CREATE DATABASE" sql statement. In this case, the sql statement is the sole source of the create parameters.

Once the database has been created, the `OnCreateDatabase` event is called. An SQL script may now be run to initialise the database, (e.g.) by using TIBXScript (see 7.1).

### 5.1.8    Dropping a Database

Once a connection has been established, it is possible to drop (disconnect and delete) a database using the `DropDatabase` method. This will delete the original database file, provided that the logged in user has sufficient privilege to do this operation.

### 5.1.9    Using the Attachment Interface

The `TIBDatabase` component also exposes the underlying `IAttachment` interface as the public property: `Attachment`. This can be used for embedded SQL. For example,

```
var employees: integer;
begin
  employees := IBDatabase1.Attachment.OpenCursorAtStart(
                          'Select count(*) From Employees').AsInteger;
```

gets the current number of rows in the employee table. Use of the `IAttachment` interface is described in the Firebird Pascal API Guide.

### 5.1.10   Using the AllowStreamConnected Property

In order to get the lists of tables and fields and to test the SQL statements, property editors (see 6.5.3) need a connection to a development version of your database. They use the dataset's `TIBDatabase` to do this and may set its connected property to true in order to achieve this. However, when the form is saved, this can also result in the connected property being saved in the form as "true".

The result is that when the compiled program is run, the `TIBDatabase` component is loaded with a connected property set to true and is then in its "streamed connected" state. This is it connects to the database as soon as the component finishes loading. This may be what you want – which is fine. But often this is not desirable.

When a `TIBDatabase` connects to the database as soon as the component finishes loading, it causes problems with error handling. That is exceptions cannot be caught and a simple error, such as mistyping a password can cause an ungraceful exit; as far as the user is concerned the program has crashed.

To avoid this, it is better to set the connected property explicitly, perhaps once the main form has finished loading and in its `OnShow` event handler (see 5.1.5). It is then possible to wrap the "connected := true" with an exception handler.

This can be ensured by setting the `TIBDatabase AllowStreamedConnected` property to false. This prevents the component entering the "streamed connected" state even if the connected property is set to true when the form was saved.

## 5.2     TIBTransaction

This non-visual component is a wrapper for the `ITransaction` interface. Its role is to:

- Provide a component that represents a transaction
- Provide a means to specify transaction parameters at design time.
- Allow design time references to be established between datasets, transactions and databases.
- Provide a centralised point for event handling when transactions are started or closed, or to react to state changes in datasets linked to transactions (e.g. to indicate data changes).

The simplest way to set transaction parameters at design time is to use the transaction editor. This is accessed by double clicking on the `TIBTransaction` component icon, once it has been placed on your form.

### 5.2.1    Highlighted Properties

| | |
|---|---|
| `Active` | Set to true to start a transaction.<br><br>Returns true when a transaction is active. Set to false to close a transaction and rollback. When set at design time will cause the transaction to be started as soon as it has loaded. |
| `DefaultAction` | Set to taCommit or taRollback. Determines the default closure of the transaction if implicitly closed. |
| `DefaultDatabase` | A reference to the default `TIBDatabase` for the transaction. |
| `IdleTimer` | If non-zero, this is the time in milliseconds between successive polls for transaction activity. If no activity has been detected between two successive polls then the transaction is automatically closed. Can be used to timeout idle transactions if needed. |
| `Params` | This is a list of parameter values to be used in the Transaction Parameter Block (TPB) at connect time. These are in "keyword=" format. See below |

### 5.2.2 Events

| AfterDelete | Called after a record has been deleted in a dataset linked to the transaction. |
|---|---|
| AfterEdit | Called after a dataset linked to the transaction has entered the edit state. |
| AfterExecQuery | Called after a TIBSQL component linked to the transaction has executed a query. |
| AfterInsert | Called after a dataset linked to the transaction has entered the insert state. |
| AfterPost | Called after a dataset linked to the transaction has posted changes. |
| AfterTransactionEnd | Called after the transaction has been closed. |
| BeforeTransactionEnd | Called before the transaction is closed. |
| OnIdleTimer | Called after the transaction has been closed due to lack of activity. |
| OnStartTransaction | Called after a transaction has been started. |

### 5.2.3 Transactions and Databases

In most cases, a transaction is linked to only a single database (the `DefaultDatabase`) and the transaction takes place in the context of the associated database connection. However, a transaction can be linked to more than one database.

Additional databases must be added at run time using the `AddDatabase` method and before the transaction is started. They can also be removed using the `RemoveDatabase` method. The public property Databases can be used to inspect the current set of databases linked to the transaction.

When more than one database has been added to a transaction, the transaction becomes a mult-database transaction co-ordinating updates over the set of databases.

### 5.2.4 Starting a Transaction

A transaction can either be started implicitly when (e.g.) a dataset is opened or explicitly by setting the `Active` property to true, or by calling the `StartTransaction` method.

Note: From IBX2 onwards, at run time, implicit start has to be enabled using the dataset's AllowAutoActivateTransaction property.

Implicit start occurs when a dataset is opened (active property set to true) and the linked transaction has not been started. In this case, the transaction is automatically started when the dataset is opened. It can also occur when the transaction's `Active` property is set at design time

and immediately after the form on which the transaction component has been placed, completes loading.

An explicit transaction start can be performed at any time while the linked database is connected. However, an exception is raised if the transaction is already started. The recommended code is:

```
IBTransaction1.Active := true;
```

Explicit start of transactions is recommended for all but the most simple applications as this allows better control over error handling and is part of a properly thought out use of transactions.

### 5.2.5    Transaction Parameters

The following keywords may be given as transaction parameters. Except as where indicated, the parameters have no value associated with them and hence are not followed by an "=" sign. Transaction parameters can also be set using the Transaction Editor (see 5.2.6).

| `consistency` | Table-locking transaction model |
|---|---|
| `concurrency` | High throughput, high concurrency transaction with acceptable consistency; use of this parameter takes full advantage of the Firebird multi-generational transaction model [Default] |
| `shared` | Concurrent, shared access of a specified table among all transactions; use in conjunction with _lock_read and lock_write to establish the lock option [Default] |
| `protected` | Concurrent, restricted access of a specified table; use in conjunction with lock_read and lock_write to establish the lock option |
| `exclusive` | Same as "protected" |
| `wait` | Lock resolution specifies that the transaction is to wait until locked resources are released before retrying an operation [Default] |
| `nowait` | Lock resolution specifies that the transaction is not to wait for locks to be released, but instead, a lock conflict error should be returned immediately |
| `read` | Read-only access mode that allows a transaction only to select data from tables |
| `write` | Read-write access mode of that allows a transaction to select, insert, update, and delete table data [Default] |
| `lock_read` | Read-only access of a specified table. Use in conjunction with shared, protected, and exclusive to establish the lock option.<br><br>Table name given as parameter value. |

| | |
|---|---|
| `lock_write` | Read-write access of a specified table. Use in conjunction with shared, protected, and exclusive to establish the lock option [Default].<br><br>Table name given as parameter value. |
| `verb_time` | *This is poorly documented and its use uncertain. Do not use unless you know how it works.* |
| `commit_time` | *This is poorly documented and its use uncertain. Do not use unless you know how it works.* |
| `ignore_limbo` | Do not wait for Limbo Transactions. Used for a garbage collector thread. |
| `read_committed` | High throughput, high concurrency transaction that can read changes committed by other concurrent transactions. Use of this parameter takes full advantage of the Firebird multi-generational transaction model. |
| `autocommit` | Server performed auto-commit of each change. |
| `rec_version` | Enables a "read_committed" transaction to read the most recently committed version of a record even if other, uncommitted versions are pending. |
| `no_rec_version` | Enables a "read_committed" transaction to read only the latest committed version of a record. If an uncommitted version of a record is pending and "wait" is also specified, then the transaction waits for the pending record to be committed or rolled back before proceeding. Otherwise, a lock conflict error is reported at once. |
| `restart_requests` | *This is poorly documented and its use uncertain. Do not use unless you know how it works.* |
| `no_auto_undo` | With no auto undo, the transaction refrains from keeping the log that is normally used to undo changes in the event of a rollback. Should the transaction be rolled back after all, other transactions will pick up the garbage (eventually). This option can be useful for massive insertions that don't need to be rolled back. For transactions that don't perform any mutations, no auto undo makes no difference at all. |
| `lock_timeout` | This takes a non-negative integer as the parameter value, prescribing the maximum number of seconds that the transaction should wait when a lock conflict occurs. If the the waiting time has passed and the lock has still not been released, an error is generated. |

The following is a suggested parameter list for a typical read/write transaction:

read_committed
rec_version
nowait

## 5.2.6    The Transaction Editor

The transaction editor is opened at design time by double-clicking on the `TIBTransaction` component once it has been placed on a form.
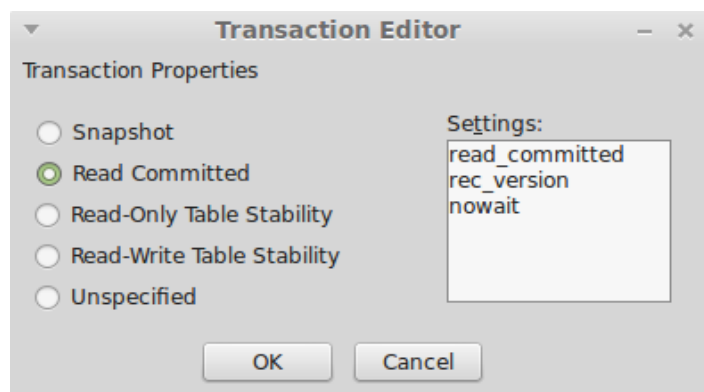


**Illustration 3: Transaction Editor**

This editor allows the transaction parameters to be set using one of the predefined options listed on the left. The result of selecting "Read Committed" is to set the `Params` property to that given in the Settings listbox.

## 5.2.7    Closing a Transaction

A transaction can either be closed implicitly with the default closure (commit or rollback) or explicitly.

Implicit closure occurs:

- When the database connection is closed.

- When the transaction idle timer expires.

- When a dataset that implicitly started a transaction closes and there are no other open datasets that are linked to the transaction.

- When the transaction's active property is set to false.

Explicit closure uses the `Commit` or `Rollback` methods and may be called at any time on an active transaction.

When a transaction is closed, all linked datasets that are still opened are closed implicitly and their data discarded.

Note: it is important to consider how your datasets react to transaction closure. See 6.6.5.

### 5.2.8    Retaining Transaction State after Closure

When explicitly closed, it is possible to retain the transaction state after a transaction has been closed by using the CommitRetaining or RollbackRetaining methods.

In either case, the effect is to immediately restart the transaction after the closure. As the database engine retains transaction state, there is no need for implicit closure of linked datasets. They thus remain open and there is no need to refresh their data.

The downside of using CommitRetaining or RollbackRetaining is that in a multi-user database, the datasets only pick up changes made by other users when they are closed and re-opened.

## 5.3    TIBEvent

This non-visual component is a wrapper for the `IEvents` interface. Its role is to:

- Provide a component that represents an event handler
- Provide a means to specify the events on which an application is waiting for at design time.

### 5.3.1    Highlighted Properties

| | |
|---|---|
| Database | A reference to the `TIBDatabase` for which this component is the event handler |
| Events | A string list containing the names of the current set of events for which the application wants to be alerted. |
| Registered | When true, the event handler is active and waiting for events. When set at design time, the event handler becomes active as soon as the linked database is connected. |

### 5.3.2    Events

| | |
|---|---|
| OnEventAlert | This event handler is called whenever an event alert is received from the database for one of the monitor events. |

### 5.3.3    Using Events

Events are raised by the database engine when a POST_EVENT statement is executed by a stored procedure or Trigger. The POST_EVENT statement also names the event. For example:

```
POST_EVENT 'MYEVENT';
```

Once the transaction, in which the trigger or stored procedure runs under, has been committed, the event is sent to all connected clients that have registered to receive the event. These clients may then react to the event. For example, by refreshing affected datasets. This is particularly useful in

multi-user databases as it provides a mechanism by which changes made by one user can then be seen by other users looking at the same data and as soon as the change is committed.

The names of the events an application is waiting for are typically defined at design time and the Registered property set to true. Then, whenever the event is signalled the `OnEventAlert` event handler is called. It is up to the application to then decide what to do.

Note: the event handler is called asynchronously and may run at any time. However, it will run in the context of the application main thread and thread synchonisation is not required.

The event handler is called once for each named event and reports the number of times the event has been signalled. This can be used to avoid reacting to duplicate events.

## 5.4     TIBSQL

This is a non-visual component and is a wrapper for the `IStatement` interface. It is used by the dataset components to execute SQL statements and to access the results of select queries. It can also be used directly by applications in which case, its role is to:

-   Provide a component that represents an SQL Statement
-   Provide a means to specify statement SQL at design time
-   Allow design time references to be established between an SQL Statement, the transaction used to execute the statement and the database connection.

IBX2 supports embedded SQL through the `TIBDatabase.Attachment` interface (see 5.1.9). However, you may still prefer to specify a statement at design time and as a component by using TIBSQL.

### 5.4.1    Highlighted Published Properties

| Database | A reference to the `TIBDatabase` to which the SQL Statement applies |
|---|---|
| GenerateParamNames | If true then positional parameters are replaced by parameter names in the format "IBXParam*nnn*" where *nnn* represents the zero based position number. Most users may safely ignore this parameter. |
| UniqueParamNames | Ignored in IBX2. Uniqueness of parameter names is determined automatically. |
| GoToFirstRecordOnExecute | When true and the SQL Statement is a select query, the cursor is positioned at the first record, if any, after the query is executed. |
| ParamCheck | Default true. The SQL Statement is parsed for named parameters. If no parameters or positional parameters are used then this may be set to false and the parsing overhead avoided. |
| SQL | The SQL Statement as a multi-line string list. |
| Transaction | A Reference to the transaction used to execute the SQL |

| | Statement. Must be active before the query is executed otherwise an exception is raised. |
|---|---|

## 5.4.2 Using TIBSQL

TIBSQL can be used to specify an SQL Statement at design time and to link it to the transaction used to execute it and the database connection to which it applies. However, the SQL Statement still has to be executed and used programmatically. `TIBSQL` cannot be used as a data source for data aware controls.

### 5.4.2.1 Executing a Stored Procedure

For example, if a `TIBSQL` component (e.g. named IBSQL1) contains the SQL Statement:

```
  Execute Procedure MyProc :Param1;
```

Then this could be executed using:

```
with IBSQL1 do
begin
  Transaction.Active := true; {make sure transaction active}
  ParamByName('MyProc').AsInteger := 1; {assume integer parameter}
  ExecQuery;
end;
```

In the above, the query is first prepared and then the parameter accessed by name and set to the value of one. The query is then executed.

Note that parameters can always be accessed by position using the Params public property, even when they also have a name. If more than one parameter has the same name, then setting a parameter by name applies the same value to all parameters with the same name. The syntax for parameter names is the same as used for the Firebird Pascal API IStatement interface (see the Firebird Pascal API Guide).

Parameter names are parsed and applied by the *fbinf* package (see the Firebird Pascal API section 6.1.1 for further details).

### 5.4.2.2 A Stored Procedure that returns Output

For example, if the store procedure is defined as:

```
  Create Procedure MyProc(aParameter Integer)
    Returns (SomeText VarChar(64))
  As ...
```

Then executing the query should return results. In the above, these are simply accessed by name using the FieldByName method. Alternatively, they can be accessed by position using the Fields property. For example:

```
with IBSQL1 do
begin
  Transaction.Active := true;
  ParamByName('MyProc').AsInteger := 1; {assume integer parameter}
  ExecQuery;
  writeln('Some Text = ',FieldByName('SomeText').AsString);
```

```
end;
```

Field Names are generally the same as the column or alias names given in the SQL Statement. However, there are exceptions. See the Firebird Pascal API Guide section 6.1.2 for further information.

Note: Firebird handles INSERT … RETURNING and UPDATE … RETURNING in the same way as stored procedures. Returned values are similarly accessed in the same way as  stored procedure output parameters i.e. using the FieldByName method.

### 5.4.2.3   Executing a Select Statement

If the TIBSQL statement is a Select SQL Statement then it may return many rows of output. It may optionally also have input parameters similar to any other SQL Statement.

When the statement is executed, a uni-directional cursor is returned to the results set. The fields in each row of the results set may be accessed using the `FieldByName` method. Alternatively, they can be accessed by position using the `Fields` property. You can scroll forwards to the next row by calling the Next method. Once all rows have been returned, the EOF property is set to true. The query should be explicit closed in order to free the results set.

Note: when a cursor is first opened, the cursor is placed on the first row unless the GoToFirstRecordOnExecute property is set to false. In which case, the Next method must be called prior to accessing the first row.

For example, if the database is the example "employee" database and the SQL Statement is:

```
  Select EMP_NO,FULL_NAME From EMPLOYEES Order by 2;
```

then the following will execute the query and write out the results, one line at a time.

```
with IBSQL1 do
begin
  Transaction.Active := true;
  ExecQuery;
  while not EOF do
  begin
    write('EMP_NO = ',Fields[0].AsString);
    writeln(': Full Name = ',Fields[1].AsString);
    Next;
  end;
  Close;
end;
```

## 5.4.3    The TIBSQL SQL Property Editor

The TIBSQL component has its own property editor for editing the SQL query property.
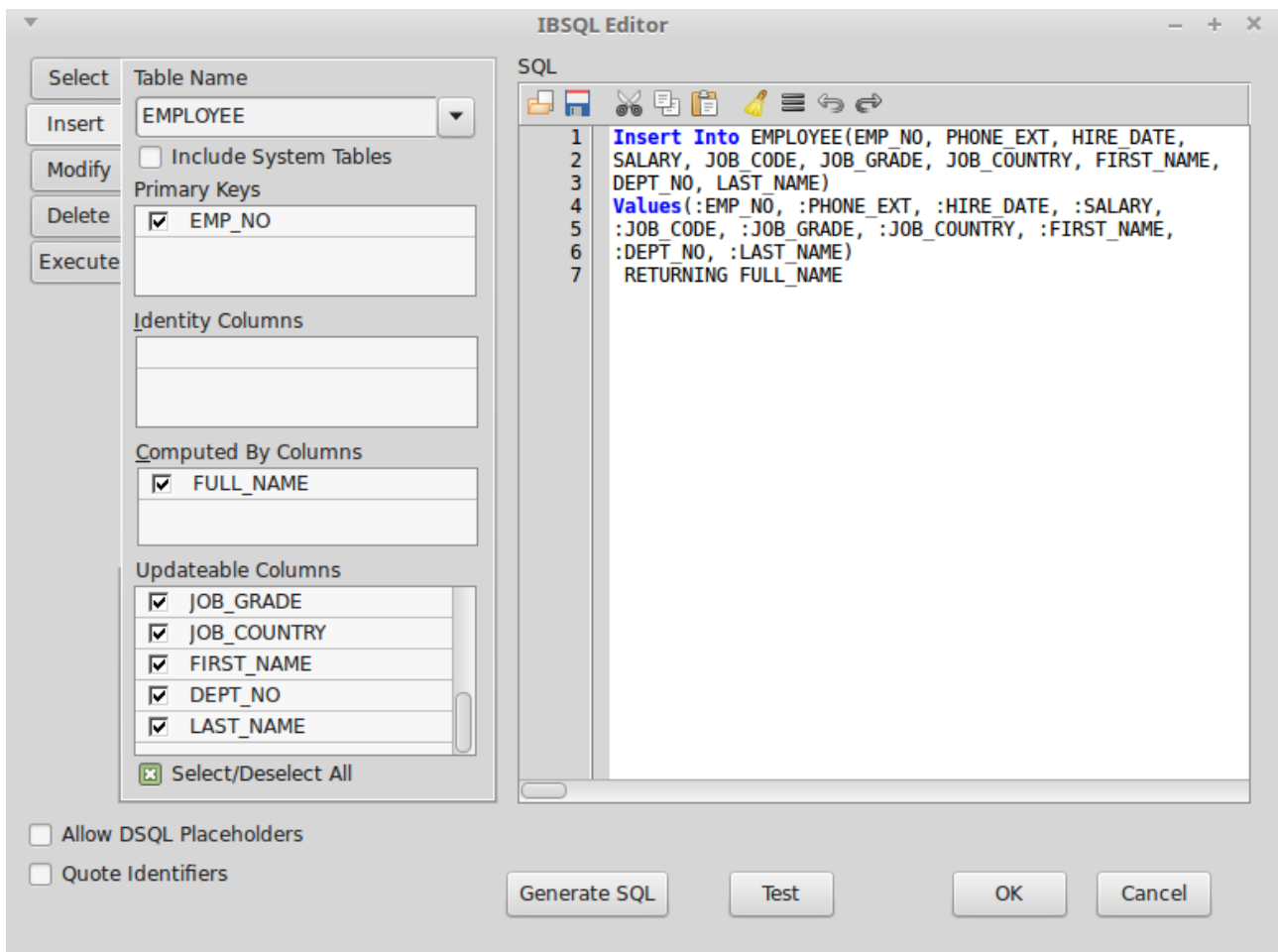
**Illustration 4: TIBSQL SQL Property Editor**

The property editor uses a SynEdit control to both display (using SQL syntax highlighting) and edit the SQL Statement. Any DDL or DML SQL statement may be given. The above example illustrates the use of an Insert statement.

Provided that the TIBSQL component is linked (at design time) to a valid database then:

- The SQL syntax can be tested using the "Test" button. When the "Test" button is clicked, the statement is passed to the Firebird Server for validation.

- The SQL Property Editor can also generate an initial SQL Statement which can then be edited by the user. The left hand tabs and panel are used to select the type and composition of the generated SQL statement.

In the above example, the Insert tab is selected implying that an Insert Statement is to be generated. The panel will then show an appropriate set of options, starting with the table name. The drop down list shows all tables in the database and the required table selected. The drop down list also uses auto-complete to simplify the selection with large numbers of tables in the database.

The available columns are then analysed into:

- Primary Keys (for the table)

- Identity Columns (Firebird 3 only – see 6.6.8)

- Computed By columns (i.e. columns computed by the server from other columns in the same row).

- Updateable Columns (i.e. all other columns).

By default all columns are selected. However, the mouse may be used to deselect the columns not required for the Insert Statement by clicking on the tick box.

When the "Generate SQL" button is clicked an Insert SQL Statement is generated:

- The selected primary keys are listed first except for those also selected as Identity Columns

- The selected Updateable columns then comprise the remainder of the insert statement.

- The selected Identity and Computed By columns are then listed in the RETURNING clause and, after the statement has been executed, their updated values can hence be retrieved from the TIBSQL.Fields property, or by using the FieldByName function (as discussed above for select queries).

Generation of Select, Update and Delete SQL statements follow similar rules, except that Identity columns are only relevant to Insert SQL Statements.

The generation of execute statements depends on the type of stored procedure (see Illustration 8).

From Firebird 3 onwards, SQL Property Editors now include a "Package Name" drop down box to allow selection of a Firebird 3 Package from which a stored procedure can be selected. With no package name selected, non-package stored procedures are listed, otherwise, the list is restricted to stored procedures in the selected package. In the illustration, the example package 'FB$OUT' is selected and then the PUT_LINE stored procedure from this package.

Clicking on the "Generate SQL" key will generate an Execute Procedure SQL statement with the input parameters specified for the procedure. Note that while these are shown in the panel, the input parameters are not selectable (a requirement of the SQL syntax).

Similarly with output parameters. These are not normally selectable and are not included in the generated SQL. However, they can be read after the statement has been executed; their values can be retrieved from the TIBSQL.Fields property, or by using the FieldByName function (as discussed above for select queries).

The exception is for stored procedures that return multiple rows. In this case, a select statement is generated and the output parameters are selectable (see 5.4.2.3).
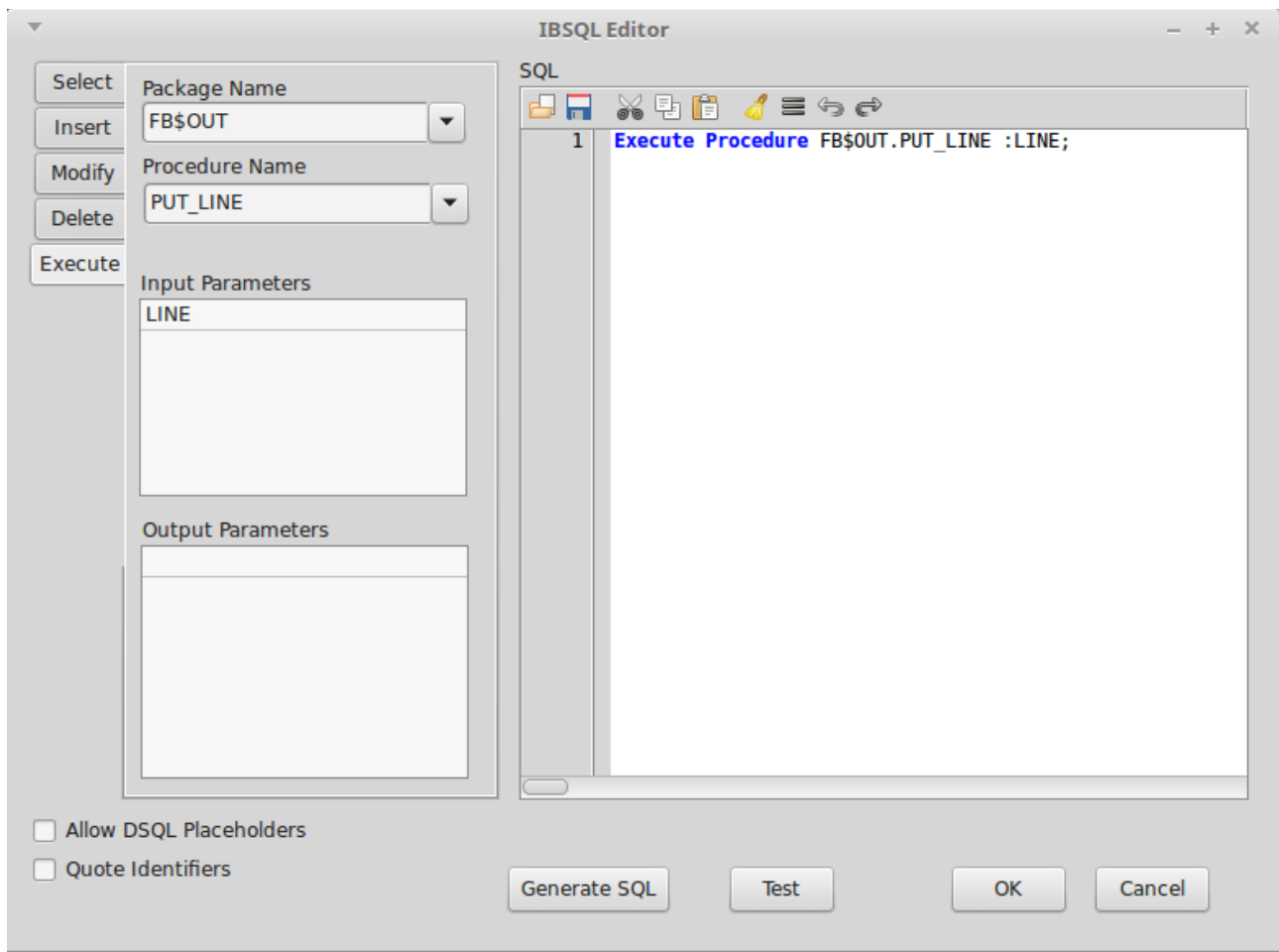
**Illustration 5: Using the TIBSQL Property Editor to Generate Execute Statements**

# 6

# The DataSet Components

The `TDataset` model introduced with Delphi and re-implemented by Free Pascal provides a common model for accessing datasets and is used by many Lazarus applications. Data aware controls can be placed on a Lazarus form and linked to a dataset via a `TDataSource`. In most cases, they also link, by name, to a single field[2] in the dataset. The focus can be placed on a single row in the dataset, and changes made to data in that row by editing the data present in one or more data aware controls. The changes are saved to the underlying database by "posting" the change (i.e. calling the `TDataset.Post` method). A `Cancel` method also exists to undo unposted changes.

`TDataset` is an abstract class which provides a common ancestor for many different types of dataset and databases. IBX provides a set of `TDataset` subclasses in order to provide efficient SQL based access to data held in Firebird databases. In addition to defining the common view of a dataset, `TDataset` also defines common methods for selecting (`locate`) a specific row in the dataset and moving backwards and forwards through the dataset (`First`, `Last`, `Next`, `Prior` and `MoveBy`).

IBX defines an internal class – `TIBCustomDataset` – which handles most of the details of providing access to a Firebird sourced dataset. However, this is not itself made directly visible and instead a set of derived classes are provided on the Firebird palette as the IBX Datasets.

## 6.1    IBX Datasets

The IBX Datasets available for use are:

- TIBTable
- TIBStoredProc
- TIBQuery
- TIBDataset

---

[2]A dataset is composed or one or more rows of data, organised into columns. A Field is said to be the intersection of a row and a column and is known by the column name. Its value is taken from the current row.

In support of TIBQuery, `TIBUpdateSQL` provides a means to add or override the SQL used to modify, insert, delete or refresh a dataset.

## 6.2     Common Concepts

### 6.2.1    Common Properties

All the IBX `TDataset` subclasses support the following properties:

| | |
|---|---|
| `Active` | At run time, set to true to open the dataset and to false in order to close the dataset. If set at design time then the dataset is opened as soon as the form it is on has completed loading. |
| `AllowAutoActivateTransaction` | If this property is set to true then activating the dataset (setting its active property to true) will automatically start the dataset's transaction. |
| `AutoCalcFields` | When true, calculated fields determined by table lookup are automatically re-calculated each time the current row changes. This is inherited behaviour from `TDataset`. |
| `AutoCommit` | May be set to "disabled" (default) or to "CommitRetaining". In the latter case whenever a row is deleted or posted, the dataset's transaction is closed with "CommitRetaining" thereby saving the change to the database whilst retaining the transaction context and allowing the dataset to stay open. |
| `BufferChunks` | **Important Performance Parameter**: This parameter determines the size by which the internal buffer allocation pool is increased every time it becomes fully used. The default is 1000 rows. <br><br> IBX will eventually cache the complete dataset in internal buffers. If the dataset is known to only ever have a few rows then BufferChunks can be set to a small number (e.g. 10 if the number of rows is typically less than 10) and the memory footprint is reduced. <br><br> On the other hand, if the number of rows is large (e.g. 100,000) then setting the BufferChunks to a larger figure (e.g. 25000) avoids a too frequent reallocation of the buffer pool as the dataset is read in. However, the figure should be chosen carefully to avoid a large number of unused buffers once the dataset has been read in. In some cases, it may even be appropriate to determine this figure at run time by first querying the database to return a count of the number of rows in the dataset and then setting BufferChunks just before the dataset is opened. |

| | |
|---|---|
| CachedUpdates | If true then updates are held locally instead of being written to the database, when changes are posted. Only when the ApplyUpdates method is called are the changes written to the database. The CancelUpdates method can be used to roll back all changes to the last time ApplyUpdates was called.<br><br>This performs a form of Commit/Rollback without having to write data to the database itself. |
| Database | A reference to the TIBDatabase managing the connection to the database used for this dataset. |
| DatasetCloseAction | Set to "DiscardChanges" (default) or "SaveChanges". This determines how the dataset handles a modified but un posted row when the dataset is closed. If "SaveChanges" is selected then the row is posted before closing the dataset. Otherwise, the changes are silently discarded. |
| Transaction | A reference to the transaction under which the dataset is read from the database and updates are applied. |
| Unidirectional | If true then the dataset can only be scrolled forwards. Avoids the need to create and maintain a large internal buffer pool (see BufferChunks above). |

### 6.2.2  Common Events

Event handlers may be defined for events trigger before and after the dataset is opened and closed, and before and after rows are inserted, deleted and posted. Similarly for entering the Edit state and cancelling changes, and before and after a transaction end.

| | |
|---|---|
| BeforeOpen | This is one of the most important event handlers and fires after the dataset Active property is set to true but before it is opened. If the select query that defines the dataset has parameters, this is a good place to set the parameter values.<br><br>This event handler can also be used to activate datasets that should be open before this one is opened[3]. |
| AfterOpen | This event handler can be used to locate the dataset on a specific record. It is also a good place to open datasets that depend on this dataset e.g. in a master/detail relationship. |

---

[3]When datasets are organised into a dependency hierarchy, if the event handers are set up correctly, only the top level dataset needs to be explicitly activated. The remainder are then opened in sequence as determined by the event handlers.

| BeforeClose | Typically used to ensure that datasets that must be closed before this one are closed. e.g. in a master/detail relationship. |
|---|---|
| AfterClose | Typically used to close datasets that no longer need to be open once this dataset has closed. |
| BeforeEdit | Raise an exception here if the row should not be edited. |
| AfterEdit | This handler can be used to set some visual indication that the row is now in the edit state. |
| BeforeInsert | Raise an exception here if the row should not be inserted at this time |
| AfterInsert | A good place to set initial values for a newly inserted row. |
| BeforeDelete | Raise an exception here if the row should not be deleted. |
| AfterDelete | Could be used to give some visual indication that there are pending changes to be committed (AutoCommit off). |
| BeforeCancel | Raise an exception here if the changes should not be cancelled. |
| AfterCancel | This handler can be used to reset the visual indication that the row is in the edit state. |
| BeforePost | Raise an exception here if the changes should not be posted, perhaps after validation. <br><br> May also be used to set field values when the field is not managed by a data aware control. |
| AfterPost | Could be used to give some visual indication that there are pending changes to be committed (AutoCommit off). |
| BeforeTransactionEnd | Raise an exception here if the transaction should not be completed at this time. |
| AfterTransactionEnd | Could be used to reset the visual indication that there are pending changes to be committed |
| OnCalcFields | A TDataset can include additional fields to those retrieved from the database and which are calculated on a per row basis. This event handler is called each time the dataset is scrolled and a new row becomes current. This is where the calculated fields should be refreshed. |

| | |
|---|---|
| `BeforeScroll` | Called before changing to a different row. Raise an exception here if the row should not be changed. |
| `AfterScroll` | Called after changing to a different row. Could be used to update controls from field values when the controls are not data aware. |

### 6.2.3 Exception Handling

An IBX dataset provides several error handling events including `OnDeleteError, OnEditError, OnPostError`. This may be used to handle database generated exceptions on a per dataset basis. However, it should be noted that all database engine exceptions are raised as `EIBInterBaseError`. A single centralised exception handler set up using `TApplication.AddOnExceptionHandler` and which reports `EIBInterBaseError` exceptions appropriately may be a more efficient approach.

### 6.2.4 Character Sets and Code Pages

Firebird text fields always belong to one of the supported character sets (e.g. UTF8). From FPC 3.0.0, Pascal AnsiStrings have a code page attribute that identifies the character set used for the string. IBX is designed to ensure that text field character sets and AnsiString code pages are always consistent and to transliterate if necessary when transferring data to and from text fields in a database. See Chapter 9 of the Firebird Pascal API Guide for more information.

## 6.3 TIBTable

`TIBTable` provides a simple means to access and update a Firebird database table or view without having to have knowledge of SQL. Only the table name needs to be provided and IBX does the rest. `TIBTable` datasets can also be arrange in Master/Detail relationships. See also the example in "ibx/examples/ibtable".

### 6.3.1 Highlighted Properties

| | |
|---|---|
| `GeneratorField` | Specifies the automatic setting of a field from a Firebird generator when a new row is appended to the table. See 6.6.3. |
| `IndexDefs` | Allows editing and hence modification of the index definitions for table. Use with the "StoreDefs" property. |
| `IndexFieldNames` | Used by the "Detail" table in a Master/Detail relation: identifies the fields (as a semi-colon separated list of field names) by which the detail table is joined to the master and also gives the sort order of the detail table. |
| `IndexName` | Identifies an index used to sort the table. This property may not be used in a "Detail" table. |
| `MasterFields` | Used by the "Detail" table in a Master/Detail relation: identifies the field names in the master table which correspond to the `IndexFieldNames`. |

| MasterSource | Used by the "Detail" table in a Master/Detail relation: Identifies the data source for the master table. |
|---|---|
| MasterDetailDelay | Used by the "Detail" table in a Master/Detail relation: when non-zero, the property value is interpreted as the number of milliseconds to wait between the master dataset being scrolled to a new row and the detailed dataset being closed and re-opened. If the master dataset is scrolled again before the timer expires then the timer is restarted.<br><br>Inserting such a delay can be a useful way of improving performance when the detail dataset is large or takes a long time to open.<br><br>Note: Requires FPC 3.0.2 or later. |
| StoreDefs | If true then the index defs are stored in the form's lfm file. |
| TableName | The name of the database table that is the source for the dataset. |
| TableTypes | Used to control the list of tables names returned from the server at design time. The default list is user tables only. System tables and views can be added to the list by selecting the appropriate options. |

### 6.3.2   Using TIBTable

A `TIBTable` is arguably the simplest IBX dataset to use – although also limited in capability. To use it simply drop the component on to a form and, in the Object Inspector, link it to a `TIBDatabase` and then select the required table from the drop down list for the `TableName` property. When the `Active` property is set, SQL is automatically generated to read the database table contents into the dataset.

The row order can be varied by selecting an (existing) index on which to sort the data, or giving a list of `IndexFileNames` in field sort order.

The table is also updateable unless its ReadOnly property is true. The SQL needed to update the database is also automatically generated. If `AllowAutoActivateTransaction` and `AutoCommit` are also true then transaction management is fully automated and all changes are automatically saved to the database. You should also select the `DatasetCloseAction` to `SaveChanges`. The example program operates in this way, with master data `AllowAutoActivateTransaction set to true`.

#### 6.3.2.1   Master/Detail Tables

The main limitation of TIBTable is that it is a single table view on the database. The dataset presented must correspond one-to-one with a table or view defined in the database. However, even within these restrictions, it is still possible to use the component to manage tables/views in master/detail relationships.

A Master/Detail relationship occurs when there are fields in common between two tables that allow the tables to be joined together in such a way that one row in the Master table relates to multiple rows in the detail table. For example, one table may be a list of departments, while the other lists employees and the department in which they work. If the table of departments is the master and the employees is the detail, then joining the two on the department name in a master detail

relationship means for each row selected in the department table, the employee table lists the employees in that department – whilst excluding the rest.

To set up two tables in a master/detail relationship:

1. Create the two `TIBTable` components and link them to the same database. In each case, set the required table name.

2. Drop a `TDataSource` component on to the form and link it to the master table.

3. Link the `MasterSource` property of the detail table to this data source.

4. Open the Detail Table's MasterFields property editor by clicking on the button next to this property in the Object Inspector (see Illustration 6).
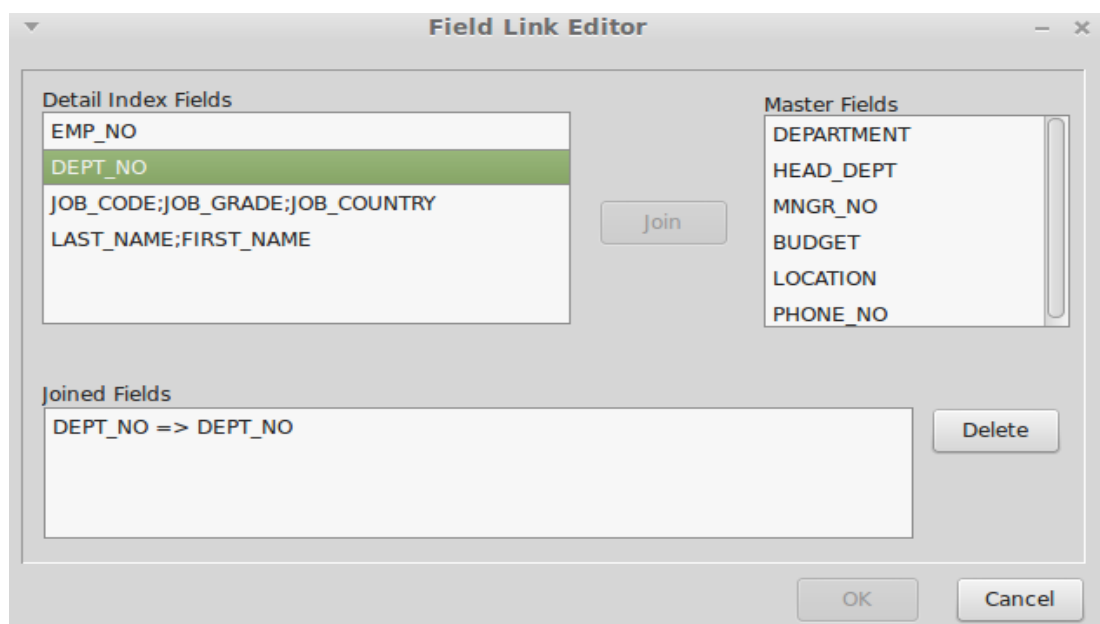


**Illustration 6: Master Fields Property Editor**

5. The "Detail Index Fields" presents a list of available index field combinations in the detail table (as defined by table indexes in the database). The appropriate index field list on which to make the join should be selected.

6. Now select the corresponding field name(s) in the Master Fields list and click on the "Join" button. The joined fields are now shown below.

7. Click on OK to complete the join.

At run time, you should first activate the Master table and then Detail table. Activating the detail table in the `AfterOpen` event handler for the Master table is one way of ensuring this. Likewise closing the detail table in the `BeforeClose` event handler for the Master table.

While the tables are open, selecting a row in the Master table will cause the detail table to be re-opened selecting the detail rows appropriate to the selected master row.

Note, in the ibx/examples/ibtable example program, the Master table is activated in the TIBDatabase AfterConnected event handler, and the Detail table activated in the Master table's AfterOpen event handler.

## 6.4 TIBStoredProc

`TIBStoredProc` is used to execute a stored procedure (on the Database Server), and one that does not generate a results set. It generated the SQL needed to execute the procedure, and hence avoids the programmer having to code this themselves. Advanced users may prefer to use embedded SQL (see 5.1.9) or TIBSQL (see 5.4).

### 6.4.1 Highlighted Properties

| | |
|---|---|
| `PackageName` | Firebird 3 onwards: used to specify the package in which the stored procedure is located. If empty then the stored procedure will be assumed to be defined outside of any package. |
| `StoredProcName` | This is the name of the stored procedure in the Firebird database. |
| | If the database server is Firebird 3 or later then if PackageName is not empty, the StoredProcName will be assumed to refer to a stored procedure in the specified package. |
| `Params` | The list of procedure input and output parameters, if any. Generated from the database. |

### 6.4.2 Using TIBStoredProc

At design time:

1. Drop a `TIBStoredProc` component on your form and select the database.

2. Firebird 3 or later: Set the PackageName property to the package in which the stored procedure is located, if any, by selecting the package name from the drop down list.

3. Set the `StoredProcName` property in the object inspector by selecting the procedure name from the drop down list. If PackageName is non-empty then the list is restricted to the stored procedures located in the package. Otherwise, the list comprises all stored procedure defined outside of any package.

At run time, and each time you execute the procedure, you may specify any required parameter values using the `Params` property or `ParamByName` method, and then execute the procedure using the `ExecProc` method.

For example:

```
with IBStoredProc1 do
begin
  Transaction.Active := true;
  ParamByName('EMP_NO').AsInteger := 8;
  ExecProc;
end;
```

In the above, it is assumed that the stored procedure has a single integer parameter with the name EMP_NO. Once this is set the procedure may be executed. Note that the parameter names are taken from the stored procedure definition in the database, e.g.

```
Create Procedure MyProc(EMP_NO Integer)
...
```

A stored procedure can also return values as output parameters. These are accessible only by interrogating the `Params` property to find the output parameter with the required name (again taken from the stored procedure definition in the database).

Note: TIBStoredProc may not be used to execute a stored procedure that returns multiple rows using the SUSPEND command. A TIBQuery or a TIBDataset should be used to execute this type of stored procedure using a select query.

## 6.5 TIBQuery

This component is a descendent of `TDataset` and creates the dataset from the results of an SQL query (Select statement or a Stored Procedure that returns a results set). The SQL query used is given by its SQL property. It is much more powerful than `TIBTable` as it can generate a dataset from any select SQL statement, joining or processing as many tables as needed. However, it does require some knowledge of SQL to use.

The dataset provided by `TIBQuery` is read only unless the component is supported by an Update Object.

### 6.5.1 Highlighted Properties

| | |
|---|---|
| DataSource | This is an optional link to another dataset that can be referenced for some or all of the values required by a parameterised query. Can be viewed as a more general version of the `MasterSource` property in a `TIBTable`. |
| MasterDetailDelay | Used by the "Detail" table in a Master/Detail relation (i.e. a TIBQuery with a non-empty DataSource property): when non-zero, the property value is interpreted as the number of milliseconds to wait between the master dataset being scrolled to a new row and the detailed dataset being closed and re-opened. If the master dataset is scrolled again before the timer expires then the timer is restarted. Inserting such a delay can be a useful way of improving performance when the detail dataset is large or takes a long time to open. Note: Requires FPC 3.0.2 or later. |
| ForcedRefresh | In an updateable query, this forces each row to be refreshed automatically from the database after it is inserted or updated. Useful for updating computed fields (from the database) and the results that may result from database triggers. Note: An Update/Insert query with a RETURNING clause may be more efficient as it avoids a full select query. Note: The AfterRefresh event is not trigged by a ForcedRefresh. Use the AfterPost handler instead. |
| GeneratorField | In an updateable query specifies the automatic setting of a field from a |

| | |
|---|---|
| | Firebird generator when a new row is appended to the dataset. See 6.6.3. |
| `Params` | The list of parameters extracted from a parameterised query. |
| `SQL` | The query SQL provided as a string list. |
| `UpdateObject` | A reference to an optional `TIBUpdateObject` (see 6.6) |

## 6.5.2    Using TIBQuery

At design time:

1.  Drop a `TIBQuery` component on your form and select the database.

2.  Specify the SQL Query by using the SQL property editor opened by clicking on the button next to the TIBQuery's SQL property in the Object Inspector (see 6.5.3).

If the SQL has no parameters then all that needs to be done at run time is to set the component's Active property to true; the query is executed and the results become available as the dataset.

•   If the database is not connected when the active property is set to true then it is implicitly opened.

•   If the transaction has not been started when the active property is set to true then an exception is raised unless the `AllowAutoActivateTransaction` property is also true.

The handling of parameterised queries is discussed below in 6.5.4.

Note: the dataset is often restricted to a single row in the select query. This is appropriate for a form used to display/edit a single row of a database table and is much more efficient that reading in an entire dataset just to edit a single row. An example of the use of multi-row datasets is when they are the source for the data aware component `TIBDynamicGrid` (see 12.1).

### 6.5.3    The Select SQL Property Editor

The Select SQL Property editor provides a means to enter and edit an SQL Query at design time. It also allows the query to be tested for correct syntax. Model queries can also be generated from information sourced from the database.
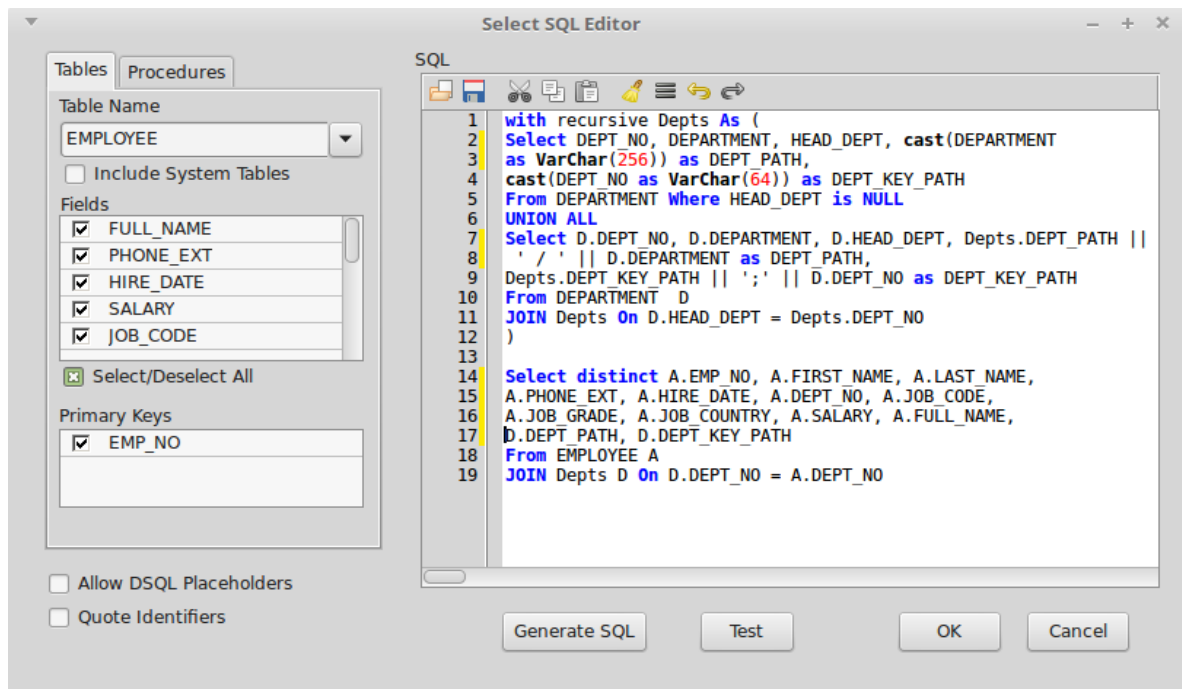


**Illustration 7: Select SQL Property Editor**

For the editor to function correctly, a `TIBDatabase` component must referenced from the `TIBQuery` and linked to a live database. A default transaction must be present and linked to the `TIBDatabase` and the Database property must be set for each data access component, and pointing to the `TIBDatabase` component.

The Select SQL Editor is shown in Illustration 7. The Select SQL Editor is a specific case. IBX SQL Editors also exist for Refresh, Update, Insert and Delete SQL.

All of IBX's SQL editors follow the same basic scheme with a combo box in the left hand frame listing available tables and listboxes showing the columns and Primary Keys respectively for the currently selected table. This can be used both for reference purposes and as a source for the Generate SQL function. The Insert and Modify property editors additionally show the "Computed By Columns" in a separate list, and the Insert Property editor has a further list for "Identity Columns" (see 6.6.8).

- A "Generate SQL" button causes a Select, Insert, Update or Delete Statement, as appropriate, to be generated for the currently selected table, and presented into the right hand editor window. If one or more columns are selected then the statement is restricted to those columns, otherwise all columns are included in the statement.

   Note that in the Modify SQL Property Editor, Update statements may be restricted to avoid updating the primary key values. If these are internal keys that are not visible to the user, then there is little

> value in including them in the update statement. Computed By columns are refreshed after the update using the UPDATE...RETURNING clause.
>
> Note: The Insert SQL Property Editor separately lists Identity Columns. When SQL is generated, these are not included in the list of fields to be inserted. However, a RETURNING clause is added to return the assigned values of these columns.

- Double-clicking on a table or column name will cause that name to be inserted in the SQL statement.

- The "Test" button can be used to validate the current SQL statement. The error generated by the database engine, if any, will be shown to the user.

In typical use, the "Generate SQL" function provides an initial set of SQL statements that can be edited to suit the actual purpose. The "Test" function can be used to check correct syntax and avoids having to compile and run the program in order to test SQL syntax correctness.

The "Quote Identifiers" option places double quotes around all Field and Table Names.

The "Allow DSQL Placeholders" option allows the use of the "?" placeholder in parameterised queries (see below).

SynEdit with SQL syntax highlighting is used to edit the statement. A toolbar and a right click popup menu are also included for common operations including manually initiated wrap on SQL token boundaries in order to display the entire text in the width of the current window.

Note: all SQL Property editors are sizeable using the mouse.

A Select SQL Statement is normally generated from tables and views. However, stored procedures that return a dataset are also a potential source. Hence the tab available for a list of suitable stored procedures (see Illustration 8).

In the Procedures tab, a drop down list of stored procedures is given. This list is restricted to stored procedures that use the "SUSPEND" command to return multiple rows and which have output parameters. The generated SQL is a select statement, but which may (as illustrated) have an input parameter to the stored procedure (see also 6.5.4).

From Firebird 3 onwards, SQL Property Editors now include a "Package Name" drop down box to allow selection of a Firebird 3 Package from which a stored procedure can be selected. With no package name selected, non-package stored procedures are listed, otherwise, the list is restricted to stored procedures in the selected package.
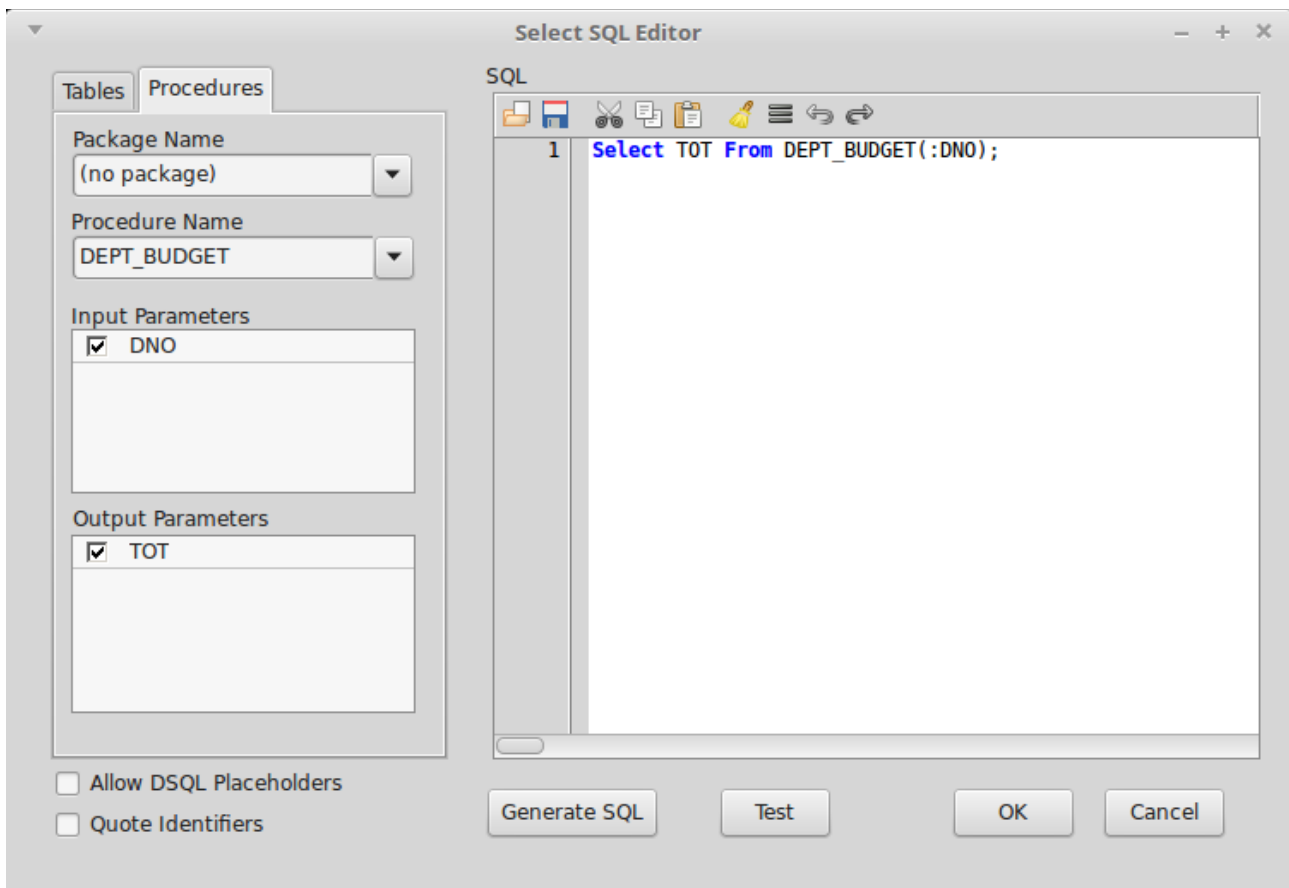
**Illustration 8: Select Query from a Stored Procedure**

### 6.5.4 Parameterised Queries

In this context, a parameterised query is an SQL statement given as the value of a `TIBQuery`'s SQL property and which has (named) parameter placeholders included in the statement.

The SQL statement syntax supported by IBX is the same as the Dynamic SQL syntax supported by Firebird. Both the Data Manipulation Language (DML) and the Data Definition Language (DDL) can be used. However, there is one important difference and that is in the handling of parameterised queries.

In normal Firebird Dynamic SQL, query parameters are represented by a '?' placeholder and are manipulated as positional parameters. IBX does allow this approach to be used but also borrows from the Firebird Procedure and Trigger Language and additionally allows the use of named parameters, where a parameter name starts with a colon character and otherwise conforms with the requirements for a database column name. For example:

```
Select A.EMP_NO, A.FIRST_NAME, A.LAST_NAME, From EMPLOYEE A
   Where A.EMP_NO = :EMP_NO;
```

is a parameterised select query where ":EMP_NO" is a parameter. This enhancement is essential for support of update objects (see 6.6). The actual implementation of parameterised queries is supported by the *fbintf* package. See section 6.1.1 of the Firebird Pascal API Guide.

For a select query, the parameter value must be specified before the query is activated. For example:

```
IBQuery1.ParamByName('EMP_NO').AsInteger := 1;
IBQuery1.Active := true;
```

If a `TIBQuery` with a parameterised Select query is activated without a parameter value assigned, and the query has a DataSource property set, the component will query the Dataset linked to `DataSource,` provided that it is already active. If it has a field with the same name as the parameter (leading ':' omitted) then the current value of the field is taken as the parameter value. This allows master/detail relationships to be simply established by careful choice of parameter names.

In practice, the `TIBQuery`'s `BeforeOpen` event handler is a good place to set any parameter values that are not automatically set from a `DataSource`. Placing them here ensures the parameter values are always set before the dataset is opened.

## 6.6     Update Objects

`TIBQuery` on its own only provides a read only dataset. If the dataset is to be updateable then a TIBUpdateSQL or TIBUpdate must also be placed on the form and set as the value of the TIBQuery's `UpdateObject` property.

### 6.6.1    TIBUpdateSQL

The `TIBUpdateSQL` provides the Modify, Insert, Delete and Refresh SQL statements needed to make the dataset updateable.

#### 6.6.1.1    Highlighted Properties

| | |
|---|---|
| `RefreshSQL` | A StringList that defines an SQL Statement used to refresh a single row in the dataset (see 6.6.9). |
| `ModifySQL` | A StringList that defines an SQL Statement used to update a single row in the database from updated field values in the current row of the dataset. (the UPDATE... RETURNING clause is supported see 6.6.1.4) |
| `InsertSQL` | A StringList that defines an SQL Statement used to insert a single row into the database from field values in the current row of the dataset (the INSERT... RETURNING clause is supported see 6.6.1.4) |
| `DeleteSQL` | A StringList that defines an SQL Statement used to delete a single row in the database corresponding to the deleted (current) row of the dataset. |

Property editors are available for each of the above and follow the same pattern as for the TIBQuery SQL property editor. The `TIBUpdateSQL` component editor provides access to all four of the above in one dialog.

#### 6.6.1.2    SQL Syntax for Update Object Queries

Note: from IBX 2.2.0 onwards, Insert and Update SQL may contain a RETURNING clause (see 6.6.1.4).

Each of the SQL Statements given in an UpdateObject are parameterised queries:

- A Refresh SQL query is a select SQL Query very similar to the `TIBQuery`'s SQL statement except that includes a "where" clause that restricts the query result to a single row: the row that corresponds to the current row of the dataset (typically using the Primary Key as the select criteria). e.g.

```
Select A.EMP_NO, A.FIRST_NAME, A.LAST_NAME, A.PHONE_EXT, A.HIRE_DATE,
    A.DEPT_NO, A.JOB_CODE, A.JOB_GRADE, A.JOB_COUNTRY, A.SALARY,
    A.FULL_NAME From EMPLOYEE A
    Where A.EMP_NO = :EMP_NO
```

- A Modify SQL query is an update SQL statement that updates a single row in the database using the current dataset row as its data source (note the use of the :OLD_ convention for a last parameter -see 6.6.1.3) e.g.

```
Update EMPLOYEE A Set
  A.EMP_NO = :EMP_NO,
  A.FIRST_NAME = :FIRST_NAME,
  A.LAST_NAME = :LAST_NAME,
  A.PHONE_EXT = :PHONE_EXT,
  A.HIRE_DATE = :HIRE_DATE,
  A.DEPT_NO = :DEPT_NO,
  A.JOB_CODE = :JOB_CODE,
  A.JOB_GRADE = :JOB_GRADE,
  A.JOB_COUNTRY = :JOB_COUNTRY,
  A.SALARY = :SALARY
Where A.EMP_NO = :OLD_EMP_NO
RETURNING FULL_NAME
```

- An Insert SQL query is an insert SQL statement that inserts a single row into the database using the current dataset row as its data source. e.g.

```
Insert Into EMPLOYEE(EMP_NO, FIRST_NAME, LAST_NAME, PHONE_EXT,
    HIRE_DATE, DEPT_NO, JOB_CODE, JOB_GRADE, JOB_COUNTRY, SALARY)
Values(:EMP_NO, :FIRST_NAME, :LAST_NAME, :PHONE_EXT, :HIRE_DATE,
    :DEPT_NO, :JOB_CODE, :JOB_GRADE, :JOB_COUNTRY, :SALARY)
RETURNING FULL_NAME
```

- A Delete SQL query is a delete SQL statement that deletes a single row from the database using the current dataset row as its data source. e.g.

```
Delete From EMPLOYEE A Where A.EMP_NO = :EMP_NO
```

In each of the above, named parameters are used with the convention that parameter names correspond to field names in the dataset. This convention is then interpreted to mean that when the query is executed, the parameter value is taken from the field in the current row with the same (alias) name.

- In the Refresh query, the "where" clause should always use one or more fields to select the row that have a combination of values that is unique to that row. Typically the fields that make up the primary key for the underlying table. If the query returns multiple rows, only the first is used to refresh the current row.

- In the update query, each field in the database is updated from its corresponding field in the current row. However, note the "OLD_" convention used in the "where" clause. This is

discussed below in 6.6.1.3. Otherwise, it should select the corresponding row in the database similar to the refresh statement.

- In the insert query, each field in the database is also updated from its corresponding field in the current row.

- In the delete query, the "where" clause selects the row in the database to be deleted and should be identical to the "where" clause in the refresh statement.

In both INSERT and UPDATE queries, the "COMPUTED BY" column FULL_NAME is returned by the query and updates the corresponding field. There is thus no need to refresh the dataset to get the up-to-date value of this field which changes when (e.g.) LAST_NAME changes (see Firebird's example employee database). See also 6.6.1.4.

### 6.6.1.3   OLD and NEW Parameters

These are typically used in Modify SQL statements. For example, when the primary key value is changed. In this case, the row to be updated needs to be selected by the original value of the primary key, while the field(s) that corresponds to the primary key needs to be set to the new value. In order to support this capability, IBX allows parameter names to be prefixed by "OLD_" and "NEW_" where the former references the fields value when it was last read from the dataset (i.e. before a call to `TDataset.Edit`), while the latter is the default and refers to the modified value set after a call to `TDataset.Edit`. For example, specify an Update SQL statement in the form:

```
UPDATE MYTABLE Set Key1 = :NEW_KEY1, COL2 = :COL2 Where Key1 = :OLD_KEY1;
```

to correctly handle database updates such as:

```
IBDataset1.Next;
IBDataset1.Edit;
IBDataset1.FieldByName('key1').AsInteger := <a new value>;
IBDataset1.Post;
```

Note that as NEW_ is the default, it does not need to be used unless the objective is to be explicitly clear.

The "OLD_" convention can also be used for setting field values in an update statement where there is a need to preserve the previous value (e.g. in a different field). For example:

```
        Update EMPLOYEE A Set
          A.EMP_NO = :EMP_NO,
          A.LAST_NAME = :LAST_NAME,
          A.PREVIOUS_NAME = :OLD_LAST_NAME,
        Where A.EMP_NO = :OLD_EMP_NO
```

The "OLD_" convention should not be used with `Refresh` as refresh cannot be used unless the dataset is in "browse" mode i.e. the current row is not being edited or inserted. Delete queries should normally use the current value for the key rather than the old value.

### 6.6.1.4   Insert and Update Returning Clauses

From Firebird 2.1 onwards, Firebird supports Insert and Update Queries with "RETURNING" clauses. After the query has been executed, this clause allows the query to return the current values for the named columns. From IBX 2.2.0 onwards, IBX updateable datasets support Insert and Update Queries with "RETURNING" clauses by:

- Once the query is executed a check is made for any returned values.

- For each returned value, the name of the returned value (its alias name) is matched against the list of alias names for the dataset columns.

- If a match is found then the returned value replaces the current value for that column in the current row.

Note: if, for some reason, the data type of the returned data is incompatible with the data type of the column, then an exception is raised.

### 6.6.1.5    Delete Returning Clauses

DELETE...RETURNING queries are also recognised. However, as they are called when the dataset row is being deleted, there is no value in updating the current record from the query result. Instead, an event handler `OnDeleteReturning` is provided. If a DELETE...RETURNING query is execute and an `OnDeleteReturning` event handler is provided then it is called with the IResults returned by the query. The event handler can then interrogate the query results and perform whatever action is necessary. For example to confirm, to the user, the deletion of a row with the returned values.

### 6.6.1.6    Using Stored Procedures for Insert, Update or Delete

It is possible to specify a Stored Procedure for an Insert, Update or Delete SQL if this results in the correct semantic i.e. row insertion, update or delete. In this case, the "Execute Procedure" syntax is used with the procedure input parameters expressed in the same (leading ':') syntax as used for parameterised queries. E.g.

```
Execute Procedure MyProc :Arg1, :Arg2
```

Stored procedures executed in this way can return a singleton row of output parameters (if the procedure definition includes output parameters). If available, these are always returned. The output parameters, if any, are treated in the same way as INSERT...RETURNING or UPDATE...RETURNING outputs. They are scanned after the stored procedure is executed and for each output parameter name that matches the aliasname of dataset field, the value of the output parameter is used to update the corresponding field in the current row.

## 6.6.2    TIBUpdate

This component can also be linked from a TIBQuery component and is a more general way to support updateable queries than that provided by TIBUpdateSQL. While TIBUpdateSQL supports single SQL statement for Delete, Update or Insert, TIBUpdate provides an event handler for Update, Insert or Delete together with an ISQLParams interface providing access to all current and "old" field values. This gives the programmer complete freedom as to how the Update, Insert or Delete is performed.

Examples of use include:

- Updating a dataset listing user privileges and roles, with Update, Insert and Delete translated into one or more Grant and Revoke statements.

- Implementing a writeable view programmatically rather than through triggers. This may be necessary when a schema update to add triggers to (e.g.) an existing view is not permitted, or when access to information not readily available at the trigger level is required.

- Filtering of dataset updates including silent discard of updates.

An example of the use of TIBUpdate to support Firebird 3 User Management (e.g. with multiple security databases) is available in examples/FB3UserManager.

### 6.6.2.1 Highlighted Properties

| | |
|---|---|
| RefreshSQL | A StringList that defines an SQL Statement used to refresh a single row in the dataset. |
| DataSet | A read only property referencing the dataset being updated. |
| OnApplyUpdates | This event is called in response to the dataset posting an Update, Insert or Delete action. The event handler is provided with the:<br><br>• Update action requested (Update, Insert or Delete), and<br><br>• An ISQLParams interface giving access by name to all field values: both current and "old" values. The "old" values (i.e. prior to the update) are accessed by prefixing the field name with "OLD_". All accesses by name are case insensitive. |

## 6.6.3 Generators

A Firebird Generator (known also as a Sequence) is used to generate a unique sequence number independent of transactions. A typical use is to generate a new primary key for a table which is guaranteed to not clash when another user also adding a record to the same table.

A new generated sequence number is often required when a new row is added to a dataset and the field used for the record key should be set to this sequence number. IBX supports this for updateable datasets when appending or inserting new rows, and through the dataset's GeneratorField property.

To specify a generator and the linked field, at design time, open the GeneratorField property editor by clicking on the button next to the property in the Object Inspector. The property editor should then appear.

Note: this property is part of the dataset (e.g. TIBQuery) and not the update object.

**Illustration 9: Define Generator Editor**

You may then select the generator from those available in the database and the dataset's field that is to be set from the generator. The field can be set either when the record is inserted or when it is posted. In most cases "On New Record" is appropriate. "On Post" is only really useful when the field is set in a "just in time" manner i.e. to avoid unused sequence numbers.

### 6.6.4   Updating Datasets

The following rules apply:

1.  Only the current row can be modified at any one time.

2.  Before a row can be modified, the dataset must be placed into the edit state by calling the `Edit` method.

    Note that data aware controls usually call the "Edit" method automatically whenever the data they present is modified.

3.  A new row can be added by calling the `Append, AppendRecord, Insert or InsertRecord` method. The dataset then enters the "insert" state.

4.  Changes are written to the database by the `Post` method. This can be used whenever the dataset is in the "edit" or "insert" state. On completion, the dataset returns to the "browse" state.

5.  Changes can be abandoned (before a post) using the `Cancel` method.

Note that enabling ForceRefresh is useful whenever the dataset includes dependent fields in the query (e.g. from a "JOINed" table) and ensures that the dataset presented is always consistent with the data in the database.

### 6.6.5    Automatic Posting

An updateable IBX dataset will automatically post a modified record when scrolling between rows. When the dataset is closed, the behaviour depends on the setting of the `DataSetCloseAction` property:

- If set to discard changes (the default) a modified record is "cancelled" and the changes lost.

- If set to save changes, the modified record is "posted" before the dataset is closed. The exception is when the dataset is closed due to transaction rollback when the changes are discarded.

### 6.6.6    The OnValidatePost Event

```
TOnValidatePost = procedure (Sender: TObject; var CancelPost: boolean) of object;
```

This event is available for all updateable datasets and is called as the first action in the Post method. If it returns with "CancelPost" set to true, then the "Cancel" method is called and the "Post" terminated. The event handler can thus decide if the Post should be cancelled by checking the actual field values, even prompting the user to decide.

In terms of event sequencing, the event  occurs before an `OnBeforePost` Event. Thus when the dataset is scrolled and the current record is in the "modified" state, then the following events will occur:

- OnValidatePost (returns CancelPost = false)
- OnBeforePost
- OnAfterPost
- OnBeforeScroll
- OnAfterScroll

or

- OnValidatePost (returns CancelPost = true)
- OnBeforeCancel
- OnAfterCancel
- OnBeforeScroll
- OnAfterScroll

Note that trying to call "Cancel" in an OnBeforePost handler does not work as the Post still proceeds and, in IBX, an error will be reported.

An exception could be raised in either the `OnValidatePost` handler or in the `OnBeforePost` handler to report an actual error in the data.

### 6.6.7    Cached Updates

If an updatable dataset's `CachedUpdates` property is set to true then updates are cached rather than written through to the database when `Post` is called. Instead, they are only written to the database when the `ApplyUpdates` method is called. This method flushes the cache and writes the changes to the database. Alternatively, the `CancelUpdates` method also flushes the cache but discards the changes instead of writing them to the database; the original row values are restored.

Note that if `DatasetCloseAction  is` set to Save Changes then ApplyUpdates is automatically called after any pending change has been posted.

Cached updates are often very useful when a form is used to edit a multi-row dataset. In this case, the updates need only be written to the database when the form is closed with a `ModalResult` of `mrOK`. If it is closed with mrCancel, then `CancelUpdates` may be called and the changes are not written through to the database.

The same effect could be achieved with transactions, but this may not be always as useful or efficient.

### 6.6.7.1   Cached Updates using OnUpdateRecord

Cached updates can be filtered or performed by an external function (to IBX) by assigning an OnUpdateRecord event handler. This has the signature:

```
TIBUpdateRecordEvent = procedure(DataSet: TDataSet; UpdateKind: TUpdateKind;
                                 var UpdateAction: TIBUpdateAction) of object;
```

The ApplyUpdates method will cycle through all modified, inserted or deleted records in the record cache in turn. For each one, it will call the OnUpdateRecord event handler specifying the UpdateKind as ukModify, ukInsert or ukDelete respectively and expecting the event handler to return with an appropriate UpdateAction specified.

When OnUpdateRecord is called, the current dataset record is set to the row for which an UpdateAction is requested and hence the OnUpdateRecord event handler can determine the field values for this row by using TDataset.Fields or TDataset.FieldByName as normal.

On return, the UpdateAction must be set to one of the following:

| `uaFail` | ApplyUpdates terminates with a user abort error. |
|---|---|
| `uaAbort` | Sysutils.Abort is called |
| `uaSkip` | The current record is skipped by ApplyUpdates and remains in the cache with its cache status unchanged i.e. as a modified, inserted or deleted record. |
| `uaRetry` | The OnUpdateRecord event handler will be recalled. |
| `uaApply` | The updated should be applied by IBX. |
| `uaApplied` | The OnUpdateRecord handler has performed the database update itself. Note: returning uaApplied without actually performing the update will leave the dataset out of sync with the database. |

### 6.6.7.2   The OnUpdateError Event

The OnUpdateError event is called when an exception is raised within IBX when a cached update is applied. Its signature is:

```
TIBUpdateErrorEvent = procedure(DataSet: TDataSet; E: EDatabaseError;
                        UpdateKind: TUpdateKind; var TheUpdateAction: TIBUpdateAction)
```

```
                              of object;
```

The event handler should return uaFail, uaAbort or uaSkip. uaFail causes the exception to be re-raised while the other two have the same semantics as above. Any other response causes the exception to be ignored and may leave the dataset out of sync with the database.

### 6.6.8    Identity Columns

Firebird 3 has introduced the Identity column, where an identity column is a column associated with an internal sequence generator. Its value is set automatically when the column is omitted in an INSERT statement.

IBX supports Identity columns by support of the INSERT … RETURNING clause (see 6.6.1.4). When an Identity column is used (e.g. to generate a unique integer for a primary key), the Insert statement in a TIBUpdateSQL should not include the column in its list of inserted values. Instead, the Identity column should be named in the RETURNING clause.

For example, if a simple example table is defined as:

```
CREATE TABLE ITEST
(
  MYKEY integer GENERATED BY DEFAULT AS IDENTITY,
  SOMETEXT varchar(64),
  PRIMARY KEY (MYKEY)
);
```

then an appropriate Insert SQL would be:

```
Insert Into ITEST(SOMETEXT)
Values(:SOMETEXT) RETURNING MYKEY;
```

When IBX executes the above Insert statement, the current value of the "SOMETEXT" column is inserted into a new dataset row and the value of the KEY column is set (by Firebird) to the next value returned from the corresponding generator. Once the statement has been executed, IBX reads the returned value of the MYKEY column and replaces the current value of the MYKEY field (NULL) with the returned value i.e. the value inserted into the new row.

When using IBX with an Identity column, you should note the following:

• The Identity column should not be included in the Insert SQL list of inserted values.

• The Identity column should be included in the values returned by the Insert statement's RETURNING clause.

• Until a new dataset row has been posted, the value of the Identity column in the current row is null. This is true between executing TDataset.Append and TDataset.Post.

• The generated value of the Identity column is available from the TDataset.AfterPost event handler onwards.

• Cached Updates: the value of an Identity column is only known once the row has been inserted into the database table, hence when cached updates are used, any Identity columns in newly inserted rows remain null until "ApplyUpdates" is called and the cached updates are applied to the database. After a call to "ApplyUpdates" the Identity Column values are set to the value assigned by Firebird.

The IBX InsertSQL property editor generates an Insert SQL statement compatible with the above.

An example of the use of Identity columns is provided in examples/IdentityColumn.

### 6.6.9    Row Refresh

The Refresh Query is used to refresh the current row by executing (typically) a select query that returns the current value of all fields in the row from the database. A refresh is used to ensure that the row is up-to-date and in sync with the database. It is often used to return the values of computed by fields after a row is posted.

A row Refresh is performed when a Refresh Query is present and:

1. The TDataset.Refresh method is called, or

2. Immediately after an inserted or updated row is posted and either:

   a) The ForcedRefresh property is true, or

   b) The dataset's select query contains read only (computed by) fields and the Insert, or Update, query used to update the database did not return any values.

In general, using INSERT...RETURNING or UPDATE...RETURNING is a more efficient way of returning the updated value of computed by fields than is a row refresh (which involves executing a select query) and is hence preferred.

Prior to IBX 2.2.0, INSERT...RETURNING and UPDATE...RETURNING was not supported and hence a row refresh was always performed after an inserted or updated row was posted to the database when the dataset fields contained read only (computed by) fields. When INSERT...RETURNING or UPDATE...RETURNING is used, it is assumed by IBX that this more efficient mechanism is used for returning the values of computed by columns and hence the automatic row refresh is not used.

The ForcedRefresh property can always be used to ensure a row refresh after an inserted or updated row has been posted regardless of whether INSERT...RETURNING or UPDATE...RETURNING is or is not used.

Note: the AfterRefresh event is only called when a row refresh is performed by a call to TDataset.Refresh.

When cached updates are used, the database is only updated when the TDataset.ApplyUpdates method is called. Hence, with cached updates, the insert or update query is only called from TDataset.ApplyUpdates and not from TDataset.Post and, consequentially, row refresh is only performed, as above, during the processing of TDataset.ApplyUpdates.

Note: cached updates may not be appropriate when the application demands that a dataset row is up-to-date immediately after a TDataset.Post.

## 6.7    TIBDataSet

The `TIBDataset` is `TIBQuery` and `TIBUpdateSQL` rolled into one. It defines an updateable dataset and in a single object. All of the above applies.

### 6.7.1   Highlighted Properties

| | |
|---|---|
| `SelectSQL` | A StringList that defines a select SQL Statement used to define the dataset. |
| `RefreshSQL` | A StringList that defines an SQL Statement used to refresh a single row in the dataset. |
| `ModifySQL` | A StringList that defines an SQL Statement used to update a single row in the database from updated field values in the current row of the dataset. |
| `InsertSQL` | A StringList that defines an SQL Statement used to insert a single row into the database from field values in the current row of the dataset |
| `DeleteSQL` | A StringList that defines an SQL Statement used to delete a single row in the database corresponding to the deleted (current) row of the dataset. |

A component editor is also available for `TIBDataset`. This is accessed by right clicking on the component once it has been placed on the form. The component editor allows all five SQL statements to be generated with a single click.

## 6.8   Dataset Fields

The TDataset model introduces both FieldDefs and Fields where:

- a FieldDef (base class TFieldDef) is created for each column in a dataset and defines various properties including the data type and size. Under IBX, the Fielddef is created from the underlying select query metadata.

- A Field (base class TField) is either dynamically created for each column when the dataset is opened or is created at design time as a published property of the Form on which the dataset is placed, and bound to the dataset column when the dataset is opened. The Field provides the interface between the dataset column and the dataset user and all data access and modification is performed through a Field. Fields are created from the FieldDefs.

### 6.8.1   FieldDefs

IBX subclasses TIBFieldDef from TFieldDef in order to hold extended Firebird specific information about each database column. This information is available as the following properties:

| | |
|---|---|
| `CharacterSetName` | Applies to string and Blob string fields and gives the Firebird character set name of the string's character set. |
| `CharacterSetSize` | Applies to string and Blob string fields and gives the maximum number of bytes in each character. |

| CodePage | Applies to string and Blob string fields and gives the AnsiString code page associated with the column's Firebird Character Set. |
|---|---|
| DataSize | The maximum number of bytes required to hold data in this column. |
| RelationName | The Firebird Table Name in which the column is located, if any. |
| ArrayDimensions | Applies to array columns only and gives the number of dimensions in the array. |
| ArrayBounds | Applies to array columns only and gives the bounds for each dimension. |
| IdentityColumn | Applies to integer and numeric columns only and is set to true if the underlying column is an "Identity Column". |

The FieldDefs can be accessed using the TDataset.FieldsDefs property. This returns a list of FieldDefs of type TFieldDef. For IBX Datasets these may be cast to TIBFieldDef in order to access the extended properties.

## 6.8.2   IBX Fields

The FCL defines several TField subclasses (e.g. TStringField). Each is specialised to a specific column data type. When the fields are created from a FieldDef, the data type of the column is used to determine the actual TField subclass that is created for the field.

IBX defines several extended TField subclasses (e.g. TIBStringField is subclassed from TStringField) in order to provide extended functionality and to make available Firebird specific column properties.

A field is typically accessed using the TDataset.FieldByName method or, for fields created at design time as Form properties, by the corresponding identifier name given to the Form property.

### 6.8.2.1   TIBBCDField, TIBSmallintField, TIBIntegerField and TIBLargeIntField

These are subclasses of the corresponding FCL classes and provide the additional property:

| IdentityColumn | Set to true if the underlying column is an "Identity Column" (see 6.6.8). |
|---|---|

### 6.8.2.2   TIBStringField

This is a subclass of TStringField. It's primary responsibility is to ensure that the ANSI code page assigned to strings read from the database matches the Firebird character set of the text string returned by the database. For strings written to the database, it compares the ANSI code page of the string with the Firebird Character set specified for the column and transliterates the string if necessary to that expected by Firebird.

It also provided the following properties:

| | |
|---|---|
| CharacterSetName | The Firebird character set name of the string's character set. |
| CharacterSetSize | The maximum number of bytes in each character. |
| CodePage | The AnsiString code page associated with the Firebird Character Set. |
| AutoFieldSize | By default true. It is applicable for Fields created at design time as Form properties. When, at run time, the dataset is opened and the field is bound to the dataset column, if true then the column data size provided by the database over-writes that specified at design time. If false then the design time value is used.<br><br>It is very rare that AutoFieldSize is not set to true. If the value set at design time is too small then this could lead to memory corruption. |

### 6.8.2.3   TIBMemoField

This is subclass of TMemoField and is used for text blobs (see also 8.1.1). It is similar to TIBStringField in that it also manages string code pages and provides for automatic transliteration of strings written to the database.

Additionally, it also allows for automatic string truncation when the contents is retrieved as "DisplayText". This is the case when, for example, a blob string is the source for a column of a TDBGrid. When a TIBMemoField's contents is retrieved as DisplayText then, depending on its DisplayTextAsClassName property:

- The text returned is the field's classname enclosed in brackets (FCL default).

- The text returned is the contents of the field truncated, if necessary, to the number of characters (including trailing ellipses) given by the inherited DisplayWidth property.

The field provides the following additional properties:

| | |
|---|---|
| CharacterSetName | The Firebird character set name of the string's character set. |
| CharacterSetSize | The maximum number of bytes in each character. |
| CodePage | The AnsiString code page associated with the Firebird Character Set. |
| DisplayTextAsClassName | If true then the inherited default behaviour is used for display text and the display text is no more than the classname. If false, then the display text is the contents of the field truncated, if necessary, as described above. |

### 6.8.2.4   TIBArrayField

This is a specialised TField subclass used for Firebird array columns. It is described in section 9.2.

## 6.9    Query Statistics

TIBDataSet, TIBQuery and TIBTable can all return query statistics.

### 6.9.1    Select Query Statistics

The following methods are supported by TIBDataSet, TIBQuery and TIBTable.

```
function GetPerfStatistics(var stats: TPerfCounters): boolean;
property EnableStatistics: boolean read FEnableStatistics write SetEnableStatistics;
```

If EnableStatistics is true when a dataset is opened then performance statistics will be accumulated for the select query used to read the dataset from the database. This is a published property and can be set at design time if required.

The function GetPerfStatistics can be called at any time after the dataset has been opened and will return the accumulated select query statistics. The statistics are initialised after the dataset is opened and then accumulate until the last record has been read from the dataset. The statistics counters returned are described in the Firebird Pascal API Guide **[6]** section 6.9.

The function returns false if statistics have not be requested or the dataset is not open.

### 6.9.2    Insert, Update and Delete Query Statistics

```
function GetRowsAffected(var SelectCount, InsertCount, UpdateCount, DeleteCount:
integer): boolean;
```

After changes to a dataset have been posted, or a row deleted, the GetRowsAffected function may be called to retrieve the affected rows counts. These counts are updated after each such action.

The function returns false if the dataset is closed.

### 6.9.3    GetRowsAffected for Select Queries

It is also possible to obtain the affected rows statistics for select queries if useful. IBX Datasets, also expose the select query statement handle as:

```
property SelectStmtHandle: IStatement {TIBDataset and TIBTable)
```

or

```
property StatementType: IStatement {TIBQuery}
```

In turn, this offers direct assess to the IStatement GetRowsAffected method. For example:

```
var SelectCount, InsertCount, UpdateCount, DeleteCount: integer;
begin
  with TIBQuery.Create(Application) do
  try
    Active := true;
    Last; {fetchall}
    if StmtHandle.GetRowsAffected(SelectCount, InsertCount, UpdateCount, DeleteCount)
then
      begin
        writeln('Selects = ',SelectCount);
        writeln('Inserts = ',InsertCount);
```

```
        writeln('Updates = ',UpdateCount);
        writeln('Deletes = ',DeleteCount);
      end;
  finally
      Free
  end;
end;
```

# 7

# IBX Support Components

## 7.1    The IBX Script Engine

TIBXScript script engine runs an SQL script from a file or stream. The text is parsed into SQL statements which are executed in turn. The intention is to be compatible with Firebird's ISQL command line utility, but with extensions:

- All DML and DDL Statements are supported.

- CREATE DATABASE, DROP DATABASE, CONNECT and COMMIT are supported.

- The following SET statements are supported:

  ◦ SET SQL DIALECT
  ◦ SET TERM
  ◦ SET AUTODDL
  ◦ SET BAIL
  ◦ SET ECHO
  ◦ SET COUNT
  ◦ SET STATS
  ◦ SET NAMES <character set>

- New Command: RECONNECT. Performs a commit followed by disconnecting and reconnecting to the database.

- Procedure Bodies (BEGIN .. END blocks) are self-delimiting and do not need an extra terminator. If a terminator is present, this is treated as an empty statement. The result is ISQL compatible, but does not require the use of SET TERM.

- DML statements may have arguments in IBX format (e.g UPDATE MYTABLE Set data = :mydata). Arguments are valid only for BLOB columns and are resolved using the

GetParamValue event. This returns the blobid to be used. A typical use of the event is to read binary data from a file, save it in a blob stream and return the blob id.

- The simple XML formats for binary blob data (see 7.6.3) and array data (see 7.6.4) as exported by TIBExtract (see 7.6) are supported.

- C++ style comment lines.

Select SQL statements are not directly supported but can be handled by:

1. An external handler (OnSelectSQL event).

2. A DataOutputFormatter. This formats the dataset returned by the select statement and writes the result to the Output Log.

If an SQL handler or a DataOutputFormatter is not present then an exception is raised if a Select SQL statement is found.

### 7.1.1   Properties:

| | |
|---|---|
| `Database` | Link to a `TIBDatabase` component |
| `Transaction` | Link to a `TIBTransaction`. Defaults to internal transaction (concurrency, wait) |
| `AutoDDL` | When true, DDL statements are always committed after execution |
| `Echo` | When true, all SQL statements are echoed to log |
| `StopOnFirstError` | When true the script engine terminates on the first SQL Error. |
| `IgnoreGrants` | When true, grant statements are silently discarded. This can be useful when applying a script using the Embedded Server. |
| `ShowAffectedRows` | When true, the number of affected rows is written to the log after a DML statement is executed. |
| `ShowPerformanceStats` | When true, performance statistics (in ISQL format) are written to the log after a DML statement is executed. |
| `DataOutputFormatter` | Identifies a Data Output Formatter component used to format the results of executing a Select Statement. |

### 7.1.2   Events:

| | |
|---|---|
| `GetParamValue` | called when an SQL parameter is found (in PSQL :name format). This is only called for blob fields. Handler should return the BlobID to be used as the parameter value. If not present an exception is raised when a |

| | |
|---|---|
| | parameter is found. Hint: use TIBBlobStream to create and read the blob from a file. |
| `OnOutputLog` | Called to write SQL Statements to the log (stdout) |
| `OnErrorLog` | Called to write all other messages to the log (stderr) |
| `OnProgressEvent` | Progress bar support. If Reset is true the value is maximum value of progress bar. Otherwise called to step progress bar. |
| `OnSelectSQL` | handler for select SQL statements. If not present, then the DataOutputFormatter is used to process select SQL statements. If neither an OnSelect Handler or a DataOutputFormatter is defined then select statements. result in an exception. An OnSelectSQL handler may either process the select statement itself or call TIBXScript.DefaultSelectSQLHandler to invoke default processing as described above. |
| `OnSetStatement` | Handler for unrecognised SET Statements. |
| `OnCreateDatabase` | This event is called immediately prior to executing a Create Database SQL statement. For example, it gives an opportunity to review the filename given for the database and to replace it with an alternative. |

## 7.1.3   Usage

The following TIBXScript functions may be used to execute an SQL statement or script:

```
function RunScript(SQLFile: string): boolean; overload;
function RunScript(SQLStream: TStream): boolean; overload;
function RunScript(SQLLines: TStrings): boolean; overload;
function ExecSQLScript(sql: string): boolean;
```

An SQL script may be passed as a File, a stream, a TStrings or as a single string. The above functions differ only in the way the script is provided. Otherwise, they are identical. The script is parsed into statements and executed one statement at a time in the order given in the script. The function returns true if all statements have been successfully executed and false otherwise.

## 7.1.4   Examples

Two example programs are provided in the "ibx/examples" directory that illustrate the use of `TIBXScript` in both GUI and console mode. These are:

1. [ibx/examples/scriptengine](ibx/examples/scriptengine)

2. [ibx/examples/fbsql](ibx/examples/fbsql)

### 7.1.4.1    The Script Engine Example

This example application illustrates use of the TIBXScript SQL script engine. It works with the example employee database and comes with various test scripts to illustrate how it works. These are all located in the "tests" directory.

Compile and run the application after first ensuring that the example employee database is available on the local server. If it is on a remote server, then you will have to adjust the `IBDatabase1.DatabaseName` property accordingly.

You can just type SQL queries into the left hand text box and click on "Execute" to run them. The results appear in the right hand text box. Select queries are supported by opening a new dynamically created window with a grid containing the query results. This window is non-modal and multiple query results can be shown simultaneously. The grid is a TIBDynamicGrid and clicking on the column header will resort the grid using the selected column.

The test scripts are loaded in the left hand text box by clicking on the "Load Script" button. The provided scripts are:

**1. CreateCountriesTable.sql**

This adds a new table "COUNTRIES" to the employee database and then populates it with country data including the country name and ISO2 and 3 character short names. At the end of the script, the contents of the new table are displayed.

**2. CreateCountriesTablewithError.sql**

This does the same as the above, except that the first insert statement contains a syntax error. It may be used to experiment with the "Stop on First Error" checkbox, and shows how the script engine can recover and continue from (some) syntax errors.

**3. DeptListView.sql**

This script adds a complex View to the database and tests the script engine in complex scenarios, such as recursive queries.

**4. createproc.sql**

This script adds three simple stored procedures. It demonstrates the different ways that procedure bodies can be declared (ISQL compatible, standard terminator and no terminator). Use of comments is also demonstrated.

**5. ParameterisedQueries.sql**

This script demonstrates the use of PSQL style query parameters for BLOB columns. In this case a new column "Image" is added to the COUNTRY Table and an image in png format (the flag of St George) is added to the entry for England. The value of the Image column is given by a parameter ":MyImage". This is resolved by the application which asks for the file containing the image to be placed in the field.

You should locate and return the "flag_en.png" file.

Note that the interactive resolution of the parameter is an example. The parameter resolution is carried out by an event handler that could, for example, have looked for a file which might conventionally have been called "MyImage.bin" to correspond to the query parameter.

**6. Reverseall.sql**

Reverses out the above.

**7. SelectQuery.sql**

Illustrates handling of select queries.

### 7.1.5    The fbsql Console Mode Application

*fbsql* is more than just a simple example and is an ISQL replacement console mode program for both interactive and non-interactive use. fbsql uses `TIBXScript` as its SQL Script Engine and `TIBExtract` (See 7.6) to extract metadata from the database. Select queries are handled by by outputing the query results to *stdout* in CSV format suitable for loading into a spreadsheet, as insert statements, or in a block format. It also includes an interactive version of TIBXScript.

Usage: fbsql <options> <database name>

```
Options:
-a             write database metadata to stdout
-A             write database metadata and table data to stdout
-b             stop on first error
-e             echo sql statements to stdout
-i <filename> execute SQL script from file
-h             show this information
-o <filename> output to this file instead of stdout
-p <password> provide password on command line (insecure)
-r <rolename> open database with this rolename
-s <sql>      Execute SQL text
-t             specify output format for SQL Statements
               BLK (default) for block format
               CSV (default) for CSV format
               INS (default) for Insert Statement format
-u <username> open database with this username (defaults to SYSDBA)


Environment Variables:
ISC_USER      Login user Name
ISC_PASSWORD  Login password
```

Saving the username and/or password as environment variables avoids having to enter them on the command line and is a more secure means of provding the password.

If no password is provided on the command line or through the environment, then the user is prompted for a password to be entered securely.

If neither an "-s" or a "-i" option is provided on the command line, then fbsql runs interactively.

*fbsql* uses IBX in console mode. Before opening this project you should tell the Lazarus IDE about the ibexpressconsolemode package. All you need to do in the IDE is to select "Packages->Open Package File" and open ibexpressconsolemode.lpk which you can find in the ibx root directory. You should then close it again  immediately afterwards. There is no need to install or compile it. Opening the package is sufficient for Lazarus to remember it.

SQL Statements Supported

- All DML and DDL Statements are supported.
- CREATE DATABASE, DROP DATABASE, CONNECT and COMMIT are supported.
- Additionally, RECONNECT is interpreted as dropping the connection and reconnecting.

ISQL Command Support

- SET SQL DIALECT
- SET TERM
- SET AUTODDL
- SET BAIL
- SET ECHO
- SET COUNT
- SET STATS
- SET NAMES <character set>
- SET HEADING
- SET ROWCOUNT
- SET PLAN
- SET PLAN ONLY
- QUIT
- EXIT

To use, compile the program in the Lazarus IDE and run it from the command line. See above for the command line parameters. For example:

```
fbsql -a -u SYSDBA -p masterkey employee
```

will write out the metadata for the local employee database to *stdout* (assuming default password).

```
fbsql -A -u SYSDBA -p masterkey -o employeedump.sql employee
```

will dump the employee database, include data, to a text file (employeedump.sql).

```
fbsql -u SYSDBA -p masterkey -i employeedump.sql
```

will recreate the database dumped in the file "employeedump.sql". Note that the "CREATE DATABASE" statement is at the start of this file and should be edited to identify the database file that is to be created. Alternatively,

```
fbsql -u SYSDBA -p masterkey -i employeedump.sql new-employee.fdb
```

will restore the database to the database file 'new-employee.fdb' provided that it has already been created as an empty database. Note that in this case, the "CREATE DATABASE" statement should remain commented out.

```
fbsql -s "Select * From EMPLOYEE" -u SYSDBA -p masterkey employee
```

will write out the contents of the EMPLOYEE table in the local employee database to stdout (assuming default password).

```
fbsql -b -e ../scriptengine/tests/CreateCountriesTable.sql -u SYSDBA -p masterkey employee
```

will run the script CreateCountriesTable.sql from the script engine test suite and apply it to the local employee database. Each statement will be echoed to stdout and processing will stop on the first error.

Note that on Linux, to run a program from the command line that is not on the PATH, you need to:

cd to the example directory "ibx/examples/fbsql"

run the program as "./fbsql" e.g.

```
./fbsql -a -u SYSDBA -p masterkey employee
```

## 7.2 The Data Output Formatters

These are helper components, primarily for use with TIBXScript, but which are also used by TIBExtract (for formatting data as SQL Insert statements). Their purpose is to execute SQL SELECT statements and to format the results of the query. Data Output Formatters are currently available for:

- Block Format Output          (TIBBlockFormatOut)
- CSV Format                   (TIBCSVDataOut)
- SQL Insert Statements        (TIBInsertStmtsOut).

### 7.2.1 Usage

For use with IBXScript: simply drop the appropriate component on to your form and link the TIBXScript DataOutputFormatter property to the required Data Output Formatter.

The Data Output Formatters may also be used directly. The properties listed below apply. All components support the following methods:

```
    procedure Assign(Source: TPersistent); override;
    procedure DataOut(SelectQuery: string; Add2Log: TAdd2Log);
    procedure SetCommand(command, aValue, stmt: string; var Done: boolean); virtual;
    class procedure ShowPerfStats(Statement: IStatement; Add2Log: TAdd2Log);
```

`Assign`: is used to copy the properties from one component to another.

`DataOut`: executes the supplied query. It formats the results as one or more lines and returns each line by calling the supplied "Add2Log" event handler.

`SetCommand`: is used by TIBXScript  to extend the processing of SET commands to the Data Output Formatter. SET (HEADING | ROWCOUNT | PLAN | PLANONLY) commands are handled this way.

`ShowPerfStats`: is a common utility function used to format IStatement performance statistics in an ISQL compatiable fashion.

### 7.2.2 Properties

| Database | Link to a `TIBDatabase` component |
|---|---|
| Transaction | Link to a `TIBTransaction`. Defaults to internal transaction |

| | (concurrency, wait) |
|---|---|
| PlanOptions | Determines whether the execution plan is returned instead of, with, or not at all, when the query results are formatted. |
| RowCount | When non-zero, limits the number of output rows. |
| ShowPerformanceStats | When true, ISQL compatible Performance Statistics are included after the query results. |
| IncludeHeader | When true, a header row is included in the results (CSV and Block Formats only). |
| QuoteChar | Character used to delimit text in CSV format output (defaults to single quotes). |
| IncludeBlobsAndArrays | When true, insert statements include blob and array data formatted as XML (see 7.6.3 and 7.6.4) |

## 7.3    The SQL Parser

IBX 1.2 introduced the `TSelectSQLParser` class (located in the `IBSQLParser` unit). This class supports the parsing and modification of Firebird Select SQL statements. It is intended to parse all such statements including UNIONs and Common Table Expressions.

Note: its purpose is to permit reliable modification of "Where", "Having" and "Order by" clauses in particular, and is not an SQL validator. While invalid SQL will often generate an exception, this is not guaranteed.

### 7.3.1    The Parser

The parser can be used as a standalone class, but is typically accessed using the "Parser" property of a `TIBDataSet` or a `TIBQuery`, and in a "BeforeOpen" event handler. Accessing the Parser property causes a `TSelectSQLParser` object to be created and its result is used when the dataset is opened.

An example of use may be found in ibx/examples/employee where it is used to filter the EMPLOYEE table query according to user selectable criteria. In this example, the BeforeOpen handler is

```
procedure TForm1.EmployeesBeforeOpen(DataSet: TDataSet);
begin
  if BeforeDate.Date > 0 then
    (DataSet as TIBParserDataSet).Parser.Add2WhereClause('HIRE_DATE < :BeforeDate');
  if AfterDate.Date > 0 then
    (DataSet as TIBParserDataSet).Parser.Add2WhereClause('HIRE_DATE > :AfterDate');

  case SalaryRange.ItemIndex of
  1:
    (DataSet as TIBParserDataSet).Parser.Add2WhereClause('Salary < 40000');
  2:
    (DataSet as TIBParserDataSet).Parser.Add2WhereClause('Salary >= 40000
                                              and Salary < 100000');
```

```
  3:
    (DataSet as TIBParserDataSet).Parser.Add2WhereClause('Salary >= 100000');
  end;

  {Parameter value must be set after all SQL changes have been made}
  if BeforeDate.Date > 0 then
     (DataSet as TIBParserDataSet).ParamByName('BeforeDate').AsDateTime
                                                  := BeforeDate.Date;
  if AfterDate.Date > 0 then
    (DataSet as TIBParserDataSet).ParamByName('AfterDate').AsDateTime := AfterDate.Date;
end;
```

In the example, two filters are available for user use:

- Restriction of "Hire Date" to a selected date range

- Restriction of salary to a drop down list of salary bands.

In each case, the filters need to be added to the SQL"Where" clause.

When the Parser object is first invoked, it is created using the original SQL text as set at design time. Calls to the method "Add2WhereClause" then do as expected – the supplied condition is ANDed with the existing "Where" Clause. An optional second parameter to Add2WhereClause is also available to OR the condition with the current "Where" clause (not shown).

In the above example, if the user has selected a given filter, then the SQL is updated as appropriate. Add2WhereClause can be called multiple times and each time it adds to the current text of the "Where" clause. Parentheses are automatically added in order to ensure that the semantics of the original condition are maintained.

In this example, the requested Hire Date could have been formatted as text e.g.

HIRE_DATE < '2015-01-01'

However, it is generally more reliable to let IBX handle date time conversions and so a parametrised query is used instead, with the parameter value being applied later on in the event handler.

Note that the example also illustrates an important rule: in a BeforeOpen event handler, parameter values must be set only after all SQL manipulation is complete. This is because the query must be "prepared" before parameter values are set and modifying the SQL always causes the query to be "unprepared" with the consequence that any parameter values are discarded.

In use, when a user changes a filter selection, the dataset is closed and re-opened causing the SQL to be re-generated and the result set appears with the filter applied.

A TIBDataSet SelectSQL or a TIBQuery SQL statement can still be updated at runtime. As before, this will close the dataset and unprepare the query. In addition, the initial SQL used for the Parser is also changed to the new value set at runtime.

### 7.3.2    Use with IBControls

The TSelectSQLParser is used by the IBControls (see chapter 12). These controls also use the Parser property and access it before the BeforeOpen event handler is called.

### 7.3.3    Example

An example of direct use of the TSelectSQLParser can be found in ibx/examples/sqlparser. This is a simple form that can be used to experiment with the parser and see how the SQL statement is affected by calling methods such as `Add2WhereClause`.

As shown in Illustration 10, you can use the example program to test out the parser by:

- pasting an SQL Query into the "Original SQL" text box

- entering an SQL Condition into one or more of the text boxes below

- selecting the required options,

- and clicking on the "Generate Updated SQL" button.

The updated SQL Statement should now appear in the right hand text box.

The example chosen here is a fairly trivial one taken from the ibx/examples/employee program and shows a single filter clause being added to the SQL used to generate the employees list.



**Illustration 10: SQL Parser Example**

### 7.3.4    TSelectSQLParser Reference

For all properties and methods consult the source code. The following are those intended to be used in a BeforeOpen event handler:

- ```
  procedure Add2WhereClause(const Condition: string; OrClause: boolean=false;
  IncludeUnions: boolean = false);
  ```

  This method is used to add an SQL condition to an SQL "Where" clause. If one does not exist in the original query, then the clause is added. By default, the condition is ANDed with the current "Where" condition. If the "OrClause" argument is true, then it is ORed.

  By default, the condition is only added to the first select statement in a UNION. If the "IncludeUnions" argument is true, then it is added to every select statement in the UNION.

- ```
  procedure Add2HavingClause(const Condition: string; OrClause: boolean=false;
  IncludeUnions: boolean = false);
  ```

  The behaviour of this method is identical to `Add2WhereClause`, except that it applies to the "Having" clause of the select statement.

- ```
  property Union: TselectSQLParser;
  ```

  When the select statement is a union, the second select statement is accessible through the "Union" property. Each select statement in the union is recursively added to the preceding statement via this property.

- ```
  property OrderByClause: string;
  ```

  The current "Order By" is accessed and replaced via this property. The text is the clause less the "Order by" keyword.

- ```
  property SQLText: string
  ```

  This property returns the current SQL statement complete with any modifications. This property may be useful when debugging.

## 7.4    ISQL Monitor

The `TIBISQLMonitor` component is a debugging aid that lets you see all SQL operations performed by IBX. It can be used to identify bottlenecks and performance problems amongst other problems. An IBX application can monitor itself or, if permitted, it can also monitor another IBX application,

### 7.4.1    TIBISQLMonitor

This component needs only to dropped on to a form and its `Enabled` property set tp true in order to start monitoring. It does not have to be connected to any other IBX component. A `TIBISQLMonitor` component can act as a source or a sink, or both for SQL trace information.

#### 7.4.1.1    Selecting what to monitor

The `TIBDatabase TraceFlags` property determines the SQL actions to be reported to a `TIBSQLMonitor` object. The available actions are:

tfQPrepare, tfQExecute, tfQFetch, tfError, tfStmt, tfConnect, tfTransact, tfBlob, tfService, tfMisc

### 7.4.1.2   SQL Reports

The `TIBSQLMonitor OnSQL` event handler is used to report each SQL action as a text message. Define a suitable event handler to receive text reports. These can be written to stdout, some log file or added to a `TMemo` for on screen reporting. The `TIBSQLMonitor TraceFlags` determine which reported actions results in OnSQL handler calls.

### 7.4.1.3   Application Monitoring

Once a `TIBSQLMonitor` has been enabled, its output is available to other applications running on the same computer. Under Unix derivatives, this is limited to other applications running under the same user.

A monitoring application requires only a `TIBSQLMonitor` component. No other IBX component need be included in the application. Otherwise, the `TraceFlags` and `OnSQL` event handler are used identically to in application monitoring.

Note: if more than one source application is active, the monitoring application will report both sources and cannot readily filter one and not the other.

## 7.4.2   Examples

There are two simple example applications used here to show the power of `TIBISQLMonitor`. These are located in "ibx/examples/isqlmonitor"

### 7.4.2.1   Integrated Monitoring

This is a minor change to the Employee example and adds a second "Monitor Form" to record a selected set of SQL events in a TMemo journal. The events monitored can be changed by changing the trace options in the TIBDatabase.

### 7.4.2.2   Remote Monitoring

This example show how `TIBISQLMonitor` can be used to monitor another application. This application comprises one form containing a single `TMemo` used to record the SQL event journal. Run it at the same time as IntegratedMonitoring and you will see the SQL event journal here as well. Note that what is monitored is controlled from the IntegratedMonitoring application which sets the monitored events in its trace flags and must call "EnableMonitoring" for any monitoring to take place.

## 7.5   TIBDatabaseInfo

Firebird provides access to database properties and statistics and this access is supported in IBX through the `TIBDatabaseInfo` component.

To use this component simply drop it on to a form and link it to the database for which information is required. You can have more than one TIBDatabaseInfo component linked to the same database.

At run time, the component properties are used to get the current state of each property and statistic available. These are:

| DBFileName | Database File Name |
|---|---|
| DBSiteName | Database site name |
| Allocation | Number of database pages allocated |
| BaseLevel | Database Version (level) number |
| DBImplementationNo | Database Implementation Number |
| NoReserve | Is space reserved for backup records |
| ODSMinorVersion | ODS minor version number |
| ODSMajorVersion | ODS version number |
| PageSize | Number of bytes per page |
| Version | Database implementation version no. |
| CurrentMemory | Amount of server memory (in bytes) currently in use |
| ForcedWrites | Number specifying the mode in which database writes are performed (0 for asynchronous, 1 for synchronous) |
| MaxMemory | Maximum amount of memory (in bytes) used at one time since the first process attached to the database |
| NumBuffers | Number of memory buffers currently allocated |
| SweepInterval | Number of transactions that are committed between "sweeps" to remove database record versions that are no longer needed |
| UserNames | List of Logged in users |
| Fetches | Number of reads from the memory buffer cache |
| Marks | Number of writes to the memory buffer cache |
| Reads | Number of page reads |
| Writes | Number of page writes |
| PagesFree | Number of free pages in the database. |
| PagesUsed | Number of used pages in the database. |
| BackoutCount | Number of removals of a version of a record by table. Formatted as a string list (see below). |
| DeleteCount | Number of database deletes since the database was last attached by table. Formatted as a string list (see below). |
| ExpungeCount | Number of removals of a record and all of its ancestors, for |

| | records whose deletions have been committed by table. Formatted as a string list (see below). |
|---|---|
| `InsertCount` | Number of inserts into the database since the database was last attached by table. Formatted as a string list (see below). |
| `PurgeCount` | Number of removals of old versions of fully mature records (records that are committed, so that older ancestor versions are no longer needed) by table. Formatted as a string list (see below). |
| `ReadIdxCount` | Number of reads done via an index since the database was last attached by table. Formatted as a string list (see below). |
| `ReadSeqCount` | Number of sequential table scans (row reads) done on each table since the database was last attached by table. Formatted as a string list (see below). |
| `UpdateCount` | Number of database updates since the database was last attached by table. Formatted as a string list (see below). |
| `DBSQLDialect` | Database SQL Dialect |
| `ReadOnly` | True if database is read only |
| `DateDBCreated` | Date/Time when the database was originally created |
| `TransactionCount` | Number of active transactions. |
| `Encrypted` | True if database is encrypted (Firebird 3.0.3 and later) |
| `EncryptionKeyName` | Name of the encryption key if available (Firebird 3.0.3 and later) |

From IBX 2.2, TIBDatabaseInfo also supports the following function:

```
function GetDatabasePage(PageNo: integer): string;
```

This returns the contents of database "PageNo" as a string with a codepage of CP_NONE. This feature is allowed only for SYSDBA or database owner for security reasons. Introduced in Firebird v2.5.

### 7.5.1    Per Table Counts

Several of the above properties return database information as a string list giving the count per table. Each line is in the format:

<Relation ID>=<count>

The <Relation ID> is an integer and may be resolved to a table name using the RDB$RELATIONS table. For example, the SQL query:

```
SELECT trim(r.RDB$RELATION_NAME) as RDB$RELATION_NAME
FROM RDB$RELATIONS r Where r.RDB$RELATION_ID = ?
```

may be used to look up the table name from the RDB$RELATIONS table.

# 7.6    TIBExtract

This component allows the extract of database metadata. The component is intended to be compliant with all Firebird extensions to the DDL up to and including Firebird 3.

The "ibx/examples/fbsql" example provides an example of the use of this component in a console mode application.

To use: at design time simply drop the component on to a form and, in the Object Inspector, link it to the required TIBDatabase.

## 7.6.1    Properties

| AlwaysQuoteIdentifiers | If true then in database dialect 3 SQL identifiers are always delimited by double quotes. If false, then they are only delimited if the identifier is a reserved word, contains lower case or special characters. |
|---|---|

## 7.6.2    Usage

At run time, the `ExtractObject` method may be used at any time when the database is connected, and in order to extract selected metadata.

```
procedure ExtractObject(ObjectType : TExtractObjectTypes; ObjectName : String = '';
                               ExtractTypes : TExtractTypes = [])
```

After the method has completed, the `TIBExtract.Items` string list property will hold the extracted metadata.

The extract object types determine the scope of the metadata extracted, while the Extract Type set further refines what is generated:

| Extract Object | Extract Type | Metadata |
|---|---|---|
| eoDatabase | | The whole database schema including grants to all objects except users other than user PUBLIC. |
| | etGrantsToUser | Include grants to all users. |
| | etData | The whole database schema plus the data as DML Insert statements immediately after the table definitions. This include binary Blobs (see 7.6.3) and Array data (see 7.6.4) in text format. |
| eoDomain | etDomain | All Domains |

| Extract Object | Extract Type | Metadata |
|---|---|---|
| | etTable | Domains used in the specified table |
| eoTable | | Specified table or all tables if no table given |
| | etDomain | Adds domains defined for the table |
| | etIndex | Adds indexes defined for the table |
| | etForeign | Adds Foreign Key Constraints defined for the table |
| | etCheck | Adds Check Constraints defined for the table |
| | etTrigger | Adds Table Triggers defined for the table |
| | etGrant | Adds Table Grants defined for the table |
| | etData | Adds table data as DML INSERT statements. |
| eoView | | List all views or just the one given by the ObjectName |
| | etTrigger | Adds Triggers defined for the view |
| | etGrant | Additionally includes grants on the View and to any triggers. |
| eoProcedure | | List all Procedures or just the one given by the ObjectName |
| | etGrant | Additionally includes grants to and on the procedure. |
| eoFunction | | List all Functions or just the one given by the ObjectName |
| eoGenerator | | List all Generators or just the one given by the ObjectName |
| | etData | Additionally includes an "ALTER SEQUENCE" statement to set the generator to the current value |
| eoException | | List all Exceptions or just the one given by the ObjectName |
| eoBLOBFilter | | List all Blob Filters or just the one given by the ObjectName |

| Extract Object | Extract Type | Metadata |
|---|---|---|
| eoRole | | List all Roles or just the one given by the ObjectName |
| | etGrant | Additionally includes grants to the Role |
| eoTrigger | | List all Triggers or just the one given by the ObjectName |
| | etTable | List all Triggers for the ObjectName (table name) |
| | etGrant | Additionally includes grants to the Trigger |
| eoForeign | | List all Foreign Keys or just the one given by the ObjectName |
| | etTable | List all Foreign Keys for the ObjectName (table name) |
| eoIndexes | | List all indexes or just the one given by the ObjectName |
| | etTable | List all indexes for the ObjectName (table name) |
| eoChecks | | List all Check Constraints or just the one given by the ObjectName |
| | etTable | List all Check Constraints for the ObjectName (table name) |
| eoData | | Lists table data as DML INSERT statements for the specified table. |

When an entire database is extracted, the order of objects is:

- Commented out CREATE DATABASE statement
- Filters
- Functions
- Domains
- Tables
- Data (if requested)
- Indexes
- Foreign Key constraints
- Generators including setting the value if requested
- Views
- Check Constraints
- Exceptions
- Create Procedure Stubs

- Triggers
- Alter Procedure statements to add procedure body
- Grants

This order is important to avoid dependency problems. The data is added as soon as the tables have been defined. This avoids problems with Foreign Key and check constraints without having to add the data in dependency order.

Procedure stubs are created before triggers to enable triggers to use procedures. However, the procedures cannot be fully defined until the triggers have been defined. Otherwise, an error will occur if a stored procedure is used to update a writeable views; views are only writeable once triggers have been defined.

The store procedure bodies are also added in dependency order, as a stored procedure may reference a procedure in a select query. This is only valid once the procedure has been defined and includes a SUSPEND statement.

### 7.6.3 Extract of Binary Blobs

TIBExtract exports binary blobs in a simple XML format and as hexadecimal characters. For example:

```
<blob subtype="0">
89504E470D0A1A0A0000000D494844520000012200000AE0803000000A565F09300000015504C54
45FFFFFFFCE1124CC0005EDB8BBEEBCBFCD0016E1858B135C5E220000018949444154789CEDDAB10D
C3301004415AA4D47FC97607133E04EF5470D8F8D69AB5AF0F5C7B78E2B4125189A84454222A1195
884A4425A2125189A84454222A1195884A4425A2125189A84454222A1195884A4425A2125189A844
54222A1195884A4425A2125189A84454222A1195884A4425A2125189A84454222A1195884A4425A2
125189A84454222A1195884A4425A2125189A84454222A1195884A4425A2125189A844749CE84C6F
DCA3CE732BD1FD9CD98DEB9AC542BF46C3139727FEBB125189A84454222A1195884A4425A2125189
A84454222A1195884A4425A2125189A84454222A1195884A4425A2125189A84454222A1195884A44
5DB0A8231F4D5F2DDF70071DD6A9984A4425A2125189A84454222A1195884A4425A2125189A84454
222A1195884A4425A2125189A84454222A1195884A4425A2125189A84454222A1195884A4425A212
5189A84454222A1195884A4425A2125189A84454222A1195884A4425A2125189A84454222A119588
4A4425A2125189A84454222A1195884A4425A21724FA0255D0459DFD53A3DD0000000049454E44AE
426082
</blob>
```

In the above example, a binary blob of subtype '0' is formatted as lines of hexadecimal characters. It is terminated by the </blob> end tag. White space is intended to be ignored when the blob data is read back in. There are always an even number of hexadecimal characters on each line.

The above can be placed in a DML statement instead of a placeholder for a blob value. For example:

```
INSERT INTO MyTable (KeyValue,BlobData) Values(1,<blob subtype="0">
89504E470D0A1A0A0000000D494844520000012200000AE0803000000A565F09300000015504C54
...
</blob>);
```

### 7.6.4 Extract of Array Data

TIBExtract exports array data in a similar simple XML format. For example:

```
<array dim = "1" sqltype = "448" length = "60" relation_name = "JOB" column_name =
    "LANGUAGE_REQ" charset = "NONE" bounds="1:5">
 <elt ix="1">Japanese
</elt>
 <elt ix="2">Mandarin
```

```
</elt>
 <elt ix="3">English
</elt>
 <elt ix="4">
</elt>
 <elt ix="5">
</elt>
</array>
```

The above is more complex than a blob and reflects the structural information that is necessary to define an array. The above example comes from the example employee database and is the value of a LANGUAGE_REQ column in the JOB table.

The "array" tag identifies:

- dim: the number of dimensions in the array

- sqltype: the SQL type of the array data (using blr type codes)

- length: the size of each array element in bytes.

- relation_name: the name of the table in which the array is defined.

- column_name: the name of the column in that table.

- charset: the name of the character set (text data only)

- bounds: a comma separated list of lower and upper bound pairs, one pair for each dimension. Each pair is separated by a ':' character.

The array element values are then nested within "elt" tags. The first level element tag is for the first dimension, and provides its index, and then so on. The inner set of "elt" tags encloses the value of the array element.

For example, in a 2D array:

```
<array dim="2" ... bounds="1:5,1:2">
  <elt ix="5">
    <elt ix="2">English</elt>
  <elt>
</array>
```

In the above, a single element is defined with co-ordinated (5,2).

# 8

# Using Firebird Blobs

Binary Large Objects (Blobs) are containers for almost unlimited amounts of binary data held within a Firebird Database. In practice, Blobs are limited by the database architectural limits and available disk storage but, perhaps the most important point is that their individual size limit is not part of the metadata. IBX supports the use of Firebird Blobs.

## 8.1    Blob Types

From the IBX viewpoint, there are two types of Firebird Blob:

- Text mode Blobs (sub type 1) that consist of character data in known character set (e.g. UTF8), or

- Binary mode Blobs for which the data type is unknown to IBX.

Note: Firebird allows for many different Blob types in addition to text blobs without giving any semantics to them. IBX does not impose any additional semantics on non-text Blob types.

## 8.1.1    Text Mode Blobs

Text Mode Blobs are represented by a `TIBMemoField` dataset field type. This is a `TBlobField` descendent and provides access to the Blob Data as AnsiStrings:

- Use the `AsString` property to both read and write the entire text blob. On reading, the AnsiString code page corresponds to the character set used to transfer the Blob. On writing, transliteration may occur if the AnsiString code page of the string is different to that which  corresponds to the character set used to transfer the Blob.

- Use the `SaveToFile` method to save the contents of a text blob to a file.

- Use the `LoadFromFile` method to load the contents of a text blob from a file.

The data aware control `TMemo`, can be used to both display and edit the text in a text blob.

In practice, the main difference between `TIBMemoFields` and basic string fields is that the former has no strong limits on how long the string is, and which is limited only by architectural constraints.

### 8.1.2 Binary Blobs

Binary blobs are used for many different purposes including holding image data. Binary Blobs are represented by the `TBlobField` dataset field type:

- Use the `AsString` property to both read and write the entire text blob. The string is always read as an untyped string and, on write, any string code page is ignored.

- Use the `SaveToFile` method to save the contents of a binary blob to a file.

- Use the `LoadFromFile` method to load the contents of a binary blob from a file.

If the Blob contains image data, then the `TDBImage` control may be used to display and update the Blob data.

## 8.2 Stream Mode access to Blobs

IBX also allows Blob Fields to be read and updated using the `TStream` class. IBX datasets give access to a Blob stream using the `CreateBlobStream` method inherited from `TDataset`. i.e.

```
function CreateBlobStream(Field: TField; Mode: TBlobStreamMode): TStream;
```

The function requires a `TBlobField` or a `TIBMemoField` to identify the blob field. The mode indicates  whether the stream is for reading (bmRead) or writing (bmWrite). Read/Write access is also possible. The dataset must be in Edit mode if the Blob is to be written to.

A Blob stream may be read from or written to in the same way as any other TStream descendent.

A Blob updated using a Blob stream updates the data in the current row. This must be "posted" in order to be written to the database.

For example:

```
var S, F: TStream;
begin
  MyDataset.Edit;
  S := MyDataset.CreateBlobStream(MyBlobField,bmWrite);
  F := TFileStream.Create('someimage.png',fmOpenRead);
  S.CopyFrom(F,0);
  MyDataset.Post;
  S.Free;
  F.Free;
```

**9**

# Using Firebird Arrays

Firebird allows you to create arrays of data types. Using an array enables multiple data items to be stored in a single column. Firebird can perform operations on an entire array, effectively treating it as a single element, or it can operate on an array slice, a subset of array elements. An array slice can consist of a single element, or a set of many contiguous elements.

Starting with IBX2, IBX now offers full support for Firebird Arrays:

- The Firebird Language Bindings in the *fbintf* package provide both `IArray` and `IArrayMetaData` interfaces. The former is used to access and manage an array or array slice, while the latter can be queried to find out information about the array such as its type, number of dimensions, etc.

- IBX itself now includes `TIBArrayField`. This is a `TField` descendent and, like any other TField descendent (e.g. `TStringField`) provides the means to access an array element when it is returned by a `TDataSet` descendent as a field of the current row. `TIBArrayField` provides the `IArray` interface as one of its properties enabling direct access to the array.

- The IBControls Package now includes a `TIBArrayGrid` visual control (see 12.5). This is a `TCustomStringGrid` descendent and may be used to display and edit an array element. Examples are provided for both one and two dimensional arrays.

The IArray interface is documented as part of the *fbintf* package.. See the Firebird Pascal API Guide chapter 8.

## 9.1    Defining an Array Element

This is fully described in the Firebird Document. For example:

```
Alter Table MyData (
    ...
    MyArray VarChar(16) [0:16, -1:7] Character Set UTF8
```

```
    );
```

An array may have a different set of values for each row. In the above example,  a two dimensional array of strings is defined. The first index may vary from 0 to 16 and the second from -1 to 7.

## 9.2    TIBArrayField

TIBArrayField is used and behaves much the same as any other TField Descendent. It can be created in the IDE using the Fields Editor and saved as part of a form (and then accessed as a property of the form) or dynamically, when a TIBCustomDataSet descendent (e.g. TIBTable, TIBQuery, etc.) is opened (it can then be accessed (e.g.) using the "FieldByName" method).

TIBArrayField defines the following additional properties:

| ArrayID: TISC_QUAD | This provides access to the Firebird internal array identifier stored in the row itself. This is not normally of direct interest to users. |
|---|---|
| ArrayIntf: IArray | This provides access to the array interface used to access and update the array element. When the field is null, this returns an empty array and setting an element of this array to any non-empty value will make the field non-null after the record has been posted.<br><br>Assigning an empty array to this property provides an alternative means to setting the field to null. (Setting the field IsNull property to true is the recommended method). |
| ArrayDimensions: integer | Returns the number of dimensions in the array. |
| ArrayBounds: TArrayBounds | This is a (pascal) dynamic array and returns an element for each dimension in the array, providing the upper and lower bound for that dimension. |

TIBArrayField also provides an additional method:

```
function CreateArray: IArray;
```

This may be used to obtain an interface to a new empty array and which is compatible with the field. Once populated, it can be assigned to the ArrayIntf property and its contents will be saved as the value of the array field once the dataset has been "posted".

Note: it is possible to retain a copy of the ArrayIntf and access it after the dataset has been scrolled. However, an exception is raised if an attempt is made to alter the contents of an array that is not linked to the current row of its TDataSet.

# 10

# Using Firebird Services

The Firebird Services API was introduced in InterBase 6.0 and is available in all versions of Firebird. It supports:

- Access to server and per database properties and statistics
- Database Backup and Restore
- Security Database Management (user credentials)
- Database Validation and Error Recovery
- Database configuration parameter management.

These functions correspond to the functionality provided by the Firebird command line utilities gbak, gfix and gsec.

IBX provides access to the services API through a set of non-visual components located on the Firebird Admin palette. Each component is focused on a specific subset of the Services API. An example program illustrates the use of each of the Firebird Admin components. This is located in "ibx/examples/services".

The IBX Services components were re-architectured in IBX 2.3 with the intent of providing a clearer and easier to use set of components. This chapter describes the new set of components introduced with IBX 2.3. For documentation of the legacy IBX Services components see Appendix A.

## 10.1  Introducing the new IBX Services Components

At first sight, the new set of components looks very similar to the previous set with the component names differing only by being prefixed by "TIBX", instead of "TIB". However, there are also some important differences:

- There is a new TIBXServicesConnection component. This represents the connection to the Firebird Services API and provides a single focus for managing the connection. In the

legacy components, each component was designed to have a separate Services API connection. The programmer then had to copy connection parameters between components to avoid the user having to log in separately to each service. The new component subclasses TCustomConnection and is hence similar to TIBDatabase.

• The services components have lost the "ServerName" and "Params" properties and any others associated with the services connection. These are replaced with a reference to the common TIBXServicesConnection component.

• The "ServiceStart" and similar level methods are no longer publicly visible. Instead, methods are available that carry out the intended function in a single call. In many cases, a single "Execute" method is used.

• Separate Client and Server side backup and restore components are available with properties and methods appropriate to each type

• There are two TMemDataset subclasses available (TIBXServicesUserList and TIBXServicesLimboTransactionsList). These are available as managed datasets for User Management and Limbo Transaction Resolution and which can be used as Data Sources for a TDBGrid.

## 10.2   Firebird Admin Component Overview

| | | |
|---|---|---|
|  | TIBXServicesConnection | This represents the connection to the Firebird Services API and provides a single focus for managing the connection. It is required for all applications that use the Services API. It is a TCustomConnection descendent. |
|  | TIBXClientSideBackupService | The client side backup service supports database backup to gbak format archives and with the destination on the client system. |
|  | TIBXServerSideBackupService | The server side backup service supports database backup to gbak format archives and with the destination on the server. |
|  | TIBXClientSideRestoreService | The client side restore service supports database restore from a gbak format archive located on the client system. |
|  | TIBXServerSideRestoreService | The server side restore service supports database restore from a gbak format archive located on the server. |
|  | TIBXConfigService | The configuration service allows database parameters to be modified, including whether the database is online, sync versus async writes, etc. |
|  | TIBXServerProperties | This service retrieves various server properties including the server version information, server |

| | | |
|---|---|---|
| | | parameters and the current status of database attachments. |
|  | `TIBXLogService` | This service supports the retrieval of the server log file contents. |
|  | `TIBXStatisticalService` | This service supports the retrieval of per database statistics. |
|  | `TIBXSecurityService` | This service supports management of the User Security Database. |
|  | `TIBXValidationService` | This service supports the invocation of various database repair actions, including validation and sweep. |
|  | TIBXLimboTransactionResolution Service | This services allows both the listing of any Limbo Transactions and their resolution. |
|  | TIBXOnlineValidationService | This service was introduced for Firebird 3 and provides for a table level validation of a database. It implements consistency checks that do not require exclusive access to a database. |
|  | TIBXServicesUserList | This is a TMemDataset descendent and works with a TIBXSecurityService to provide a means to display and manage the users on a given server. |
|  | TIBXServicesLimboTransactions List | This is a TMemDataset descendent and works with a TIBXLimboTransactionResolutionService to provide a means to display and fix a database's Limbo transactions, if any. |

## 10.3   Common Concepts

A TIBXServicesConnection represents the connection to a given Server's Services API and at least one TIBXServicesConnection component must be included in a project in order to use the Services API.

The functions available through the Services API are then packaged into individual components, each of which provides a related set of services.

- All components that provide access to the Services API have a "ServicesConnection" property that identifies the TIBXServicesConnection component providing the connection to the Server's Services API.

- Each component that accesses services targeted on a named database, have a "DatabaseName" property. This is either an alias name or the full pathname of a database on the server to which the TIBXServicesConnection is connected to.

### 10.3.1  Common Reporting and Event Handling

Many of the services components return a report in an OutputLog (TStrings). This is typically returned by the Execute Method i.e.

```
procedure Execute(OutputLog: TStrings);
```

Each line in the report is returned individually from the server. However, the OutputLog is only available when the action completes. However, it is possible to inspect each line of text as it is returned using the service's OnGetNextLine event handler:

```
TIBXOnGetNextLine = procedure(Sender: TObject; var Line: string) of object;
property OnGetNextLine: TIBXOnGetNextLine read FOnGetNextLine write FOnGetNextLine;
```

This can be used to give a line by line output, or simply to make the presentation of the log more responsive. For example, when setting the OutputLog to a TMemo.Lines, the TMemo can be made to update after each line is appended by creating an OnGetNextLine event handler containing:

```
Application.ProcessMessages;
```

If the event handler is used to display each line and there is no role for an OutputLog, then the parameter may be set to nil.

## 10.4   The TIBXServicesConnection

All applications using the IBX Services Components must have at least one TIBXServicesConnection component. This represents the connection to a server's Services API and is used by all the IBX Services components. It is a subclass of TCustomConnection and follows the same model as similar descendants such as TIBDatabase.

### 10.4.1  Properties

| Connected | Set to true to attach to the server and establish a connection with it. Set to false to terminate an active connection. |
|---|---|
| ConnectString | This contains the connect string used to establish a connection with the server. It may either be set directly or via the separate ServerName, Protocol and PortNo. Properties. |
| LoginPrompt | Set to true to enable use of the built-in login prompt dialog. |
| Params | Holds the login user name and password, as a list of keyword equals string (e.g. user_name=SYSDBA, password=masterkey). It is recommended that the password is not set at design time. A component editor is available to set each service's parameters. |
| Protocol | Determines the connection type (local, TCP, SPX, Named Pipe, inet, |

| | wnet or xnet). See the Firebird documentation for more information. Updating this property automatically updates the ConnectString. |
|---|---|
| ServerName | The (domain) name of the server. Updating this property automatically updates the ConnectString. |
| PortNo | The port number on which the server is listening. Only required for a non-default server port number. Updating this property automatically updates the ConnectString |
| ServerVersionNo | A read only integer array [1..4] giving the Firebird Release number (elements 1..3) and the build no. (element 4). |
| ServiceIntf | The public property **ServiceIntf** exposes the IServiceManager interface used to communicate with the server. This interface is available (non-nil) when Connected is true. |

## 10.4.2  Event handling

In addition to the standard TCustomConnection events, TIBXServicesConnection also supports:

| | |
|---|---|
| OnLogin | This can be used to replace the built-in login dialog to provide the servername, username and password either programatically or by your own dialog or some other means of determining the user name and password.<br><br>This event is only called if LoginPrompt is true. |
| OnSecurityContextException | This event is called when handling an exception due to a user database using a different security database to that to which the service is currently logged into. It is used to direct the services connection as to how it is to be resolved. The event handler should return:<br><br>• scRaiseError (default) to abort the current action and raise an exception.<br><br>• scReconnect to drop the current services API connection and to log in using the same security database as used by the user database. This will typically result in the user being prompted to enter a new username and password valid for the security database (either using the built-in login dialog or by invoking the OnLogin event handler if set). |

### 10.4.3 Connecting to the Services Manager

The simplest way to connect to the Services Manager is to set the "Connected" property to true. The TIBXServicesConnection will then attempt to connect to the Services Manager using the Connect String and parameters given in the component properties.

Alternatively, the "ConnectUsing" method may be used:

```
procedure ConnectUsing(aDatabase: TIBDatabase);
```

This creates a compatible Services API connection for the aDatabase database connection. The database connection must be in its connected state. The Services API connection is then established such that:

- The Server Name, Protocol and Port No are extracted from the database connect string.

- A Local connection (using the embedded server) is used if and only if the database connection is also using the embedded server.

- If the database connection was authenticated using an alternative security database then an "expected_db" parameter is added to the login parameters so that the services connection is also logged in using the same security database.

- All components using the TIBXServicesConnection and which have a DatabaseName property (e.g. TConfigService) will have their DatabaseName property set to the database name extracted from the TIBDatabase connect string.

The ConnectUsing procedure does not change the "Params" property. That is the user name, password and sql role name specified in the Params property will be those used for connecting to the Services API. If the LoginPrompt is true then a Login Dialog will still appear, or the OnLogin event will be called.

If it is desired to use the same Params settings as used for the database, then the SetDBParams method may be used to copy them prior to calling ConnectUsing.

```
procedure SetDBParams(DBParams: TStrings);
```

## 10.5   The Backup Service

The backup service supports database backup to gbak format archives. Both server side and client side backup file locations are supported by providing two specialised versions of the Backup Service. Before the backup is started, the following common properties and events may be set:

| | |
|---|---|
| ServicesConnection | Required: a reference to the TIBXServicesConnection providing the connection to the server. This must be in its connected state in order to perform a backup. |
| BackupFiles | Server Side Backups only: This is a list full pathnames to one or more backup files on the server. When more than one is specified, all but the last should be followed by "=nnn" where nnn is the maximum length in bytes for the file. |
| BlockingFactor | See gbak documentation  for non-zero values (probably obsolescent) |

| | |
|---|---|
| `DatabaseName` | Alias of or full pathname for database on the server. <br> Note: this is not a connect string and does not include the servername, portno or protocol components. |
| `Options` | See gbak documentation for interpretation of each option. |
| `Verbose` | Server Side Backups only: if true then additional text messages are generated. |
| `StatisticsRequested` | In verbose mode additional statistics may be provided with in the output log: <br><br> • Total Time <br> • Time Delta <br> • Page Reads <br> • PageWrites |
| `OnGetNextLine` | Server side backups only: This event is called whenever a line of text is received from the server. |

### 10.5.1 Server Side Backup

The following method is called to perform a server side backup:

```
procedure Execute(OutputLog: TStrings);
```

The backup is made from the database to the backup file(s). If OutputLog is non-nil then any lines of text received from the server during the backup (usually in verbose mode only) are added to the TStrings passed as the parameter. The method returns on completion.

### 10.5.2 Client Side Backup

The following methods are available for a client side backup:

```
procedure BackupToStream(S: TStream; var BytesWritten: integer);
procedure BackupToFile(aFileName: string; var BytesWritten: integer);
```

These are used to backup the database given by the DatabaseName to either a TStream or a named file. On completion, the number of bytes received and written to the file are returned.

## 10.6 The Restore Service

The Restore service supports database restore from gbak format archives. Both server side and client side backup file locations are supported. Before the restore is started, the common properties must be set plus the following:

| ServicesConnection | Required: a reference to the TIBXServicesConnection providing the connection to the server. This must be in its connected state in order to perform a database restore. |
|---|---|
| DatabaseFiles | A list of one or more full pathnames for the database file(s) on the server. |
| Options | See gbak documentation for interpretation of each option. This must include either CreateNewDB (default) or Replace, but not both. |
| PageBuffers | See gbak documentation |
| PageSize | See gbak documentation |
| Verbose | If true then additional text messages are generated. |
| StatisticsRequested | In verbose mode additional statistics are provided with the output log:<br><br>• Total Time<br>• Time Delta<br>• Page Reads<br>• PageWrites |
| OnGetNextLine | Server side restores only: This event is called whenever a line of text is received from the server. |

### 10.6.1 Server Side Restores

The method is used to perform a server side restore is performed after the above properties have been set:

```
    procedure Execute(OutputLog: TStrings);
```

The Database is restored to the specified file(s) using the archive(s) given in the BackupFiles list. If OutputLog is non-nil then any lines of text received from the server during the backup (usually in verbose mode only) are added to the TStrings passed as the parameter. The method returns on completion.

### 10.6.2 Client Side Restores

The following methods are available for a client side restore i:

```
  procedure RestoreFromStream(S: TStream; OutputLog: TStrings);
  procedure RestoreFromFile(aFileName: string; OutputLog: TStrings);
  procedure RestoreFromFiles(FileList: TStrings; OutputLog: TStrings);
```

These methods may be used, respectively, to restore the database from an archive sourced from a TStream, a named file, or a list of named files. If OutputLog is non-nil then any lines of text received from the server during the backup (usually in verbose mode only) are added to the TStrings passed as the parameter.

## 10.7    The Configuration Services

The TIBXConfigService comprises a set of methods, each of which performs a specific action. The following properties must be set:

| | |
|---|---|
| ServicesConnection | Required: a reference to the TIBXServicesConnection providing the connection to the server. This must be in its connected state to use any of the configuration services. |
| DatabaseName | Alias of or full pathname for database on the server.<br><br>Note: this is not a connect string and does not include the servername, portno or protocol components. |

The following methods are available, each of which sets the associated attribute for the database identified by DatabaseName,

| | |
|---|---|
| ShutdownDatabase | Puts the database into its shutdown state according to the selected options and within the given "wait" time (seconds). Once shutdown, only the SYSDBA user can log into the database. |
| BringDatabaseOnline | Puts the database into its online state (reverse of shutdown). |
| SetSweepInterval | Sets the automatic sweep interval |
| SetDBSqlDialect | Sets the default database SQL dialect (1 or 3) |
| SetPageBuffers | Set the default number of cache buffers to the specified number. |
| ActivateShadow | Activates a database "shadow file" See the Firebird Documentation for more information on database shadow files. |
| SetReserveSpace | Configure the database to fill data pages when inserting new records (true), or reserve 20% of each page for later record deltas (true) |
| SetAsyncMode | Toggles between async writes (true) and sync writes (false). |
| SetReadOnly | Sets read only or read/write mode. |
| SetNoLinger | Set the No Linger Flag on a database. This is a "one-shot" override. Once set, the server will close the database immediately the last attachment is gone, regardless of the LINGER setting in the database. The LINGER setting is retained and works normally the next time. |

## 10.8   The Server Properties Service

This service retrieves various server properties including the server version information, server parameters and the current status of database attachments. The following property is required in order to use the service:

| | |
|---|---|
| ServicesConnection | Required: a reference to the TIBXServicesConnection providing the connection to the server. This must be in its connected state in order to access server properties. |

The information returned is divided up into:

- Server Version Information
- Active Database Information, and
- Configuration Parameters

This information is made available as public properties which may then be read to access the requested information. Accessing a property automatically retrieves the requested information set from the database, which is then cached until the connection is terminated. These properties are:

| | |
|---|---|
| VersionInfo | ```
ServerVersion: String;
ServerImplementation: string;
ServiceVersion: Integer;
``` |
| DatabaseInfo | ```
NoOfAttachments: Integer;
NoOfDatabases: Integer;
DbName: array of string; {List of connected databases}
``` |
| ConfigParams | ```
ConfigFileData: TConfigFileData;
ConfigFileParams: array of string;
BaseLocation: string;
LockFileLocation: string;
MessageFileLocation: string;
SecurityDatabaseLocation: string;
``` |

For example:

```
var i: integer;
begin
  with IBServerProperties1 do
  begin
    writeln('Server Version = ' + VersionInfo.ServerVersion);
    writeln('Server Implementation = ' + VersionInfo.ServerImplementation);
    writeln('Service Version = ' + IntToStr(VersionInfo.ServiceVersion));

    writeln('No. of attachments = ' + IntToStr(DatabaseInfo.NoOfAttachments));
    writeln('No. of databases = ' + IntToStr(DatabaseInfo.NoOfDatabases));
    for i := 0 to DatabaseInfo.NoOfDatabases - 1 do
      writeln('DB Name = ' + DatabaseInfo.DbName[i]);

    writeln('Base Location = ' + ConfigParams.BaseLocation);
    writeln('Lock File Location = ' + ConfigParams.LockFileLocation);
    writeln('Security Database Location = ' + ConfigParams.SecurityDatabaseLocation);
  end;
end;
```

## 10.9   The Log Service

This is a simple service that may be used to retrieve the current server log file contents. The following property is required in order to use the service:

| | |
|---|---|
| ServicesConnection | Required: a reference to the TIBXServicesConnection providing the connection to the server. This must be in its connected state in order to access the server log. |

The server log is retrieved using the Execute method:

```
procedure Execute(OutputLog: TStrings);
```

When the method returns, the current server log will have been added to the OutputLog.

## 10.10  The Database Statistics Services

This service supports the retrieval of per database statistics as text data. The following properties must be set:

| | |
|---|---|
| ServicesConnection | Required: a reference to the TIBXServicesConnection providing the connection to the server. This must be in its connected state in order to retrieve database statistics. |
| DatabaseName | Alias of or full pathname for database on the server.<br><br>Note: this is not a connect string and does not include the servername, portno or protocol components. |
| Options | Used to request the category of the statistics returned. |

The options available are:

| | |
|---|---|
| HeaderPages | Request only the information in the database header page |
| DataPages | Request statistics for user data pages |
| IndexPages | Request statistics for user index pages |
| SystemRelations | Request statistics for system tables and indexes — in addition to user tables and indexes |

The statistics requested are retrieved using the Execute method:

```
procedure Execute(OutputLog: TStrings);
```

When the method returns, the requested statistics will have been added to the OutputLog.

## 10.11  The Security Service

The TIBXSecurityService service supports management of the User Security Database. It supports:

- The listing of all User Names, and other user identification information
- Adding New Users
- Modifying Existing Users (including changing passwords)
- Deleting Users.
- From Firebird 2.5 onwards, display and management of the user Admin Role.
- From Firebird 2.5 onwards, setting the global "Auto Admin" state.

The following property is required in order to use the service:

| | |
|---|---|
| ServicesConnection | Required: a reference to the TIBXServicesConnection providing the connection to the server. This must be in its connected state in order to access the security services. |

### 10.11.1 Listing all User Names

The `DisplayUsers` method is used to retrieve the list of user names and other user identification information to the `UserInfo` property. This information can then be displayed to the user. For example:

```
var i: integer;
begin
  with IBSecurityService1 do
  begin
    DisplayUsers;
    for i := 0 to UserInfoCount - 1 do
    with UserInfo[i] do
    begin
      writeln('User ID = ',UserID);
      writeln('Group ID = ',GroupID);
      writeln('User Name = ',UserName);
      writeln('First Name = ', FirstName);
      writeln('Middle Name = ', MiddleName);
      writeln('Last Name = ', LastName);
      write('Admin Role = ');
      if AdminRole then writeln('true') else writeln('false');
    end;
  end;
end;
```

### 10.11.2 Adding a User

The `AddUser` method is used to add a user to the Security Database. The `TIBSecurityService` `UserName` and `Password` properties should be set before this method is called to set the user name and password, respectively. The complete set of properties that may be set are:

| | |
|---|---|
| UserName | User (or Login) Name |
| Password | The user's password |
| FirstName | The user's first name |

| | |
|---|---|
| `MiddleName` | The user's middle name |
| `LastName` | The user's last name |
| `UserID` | The Unix UID |
| `GroupID` | The Unix GID |
| `AdminRole` | True if the user is granted the Admin Role, otherwise false. |

The following illustrates the use of the `AddUsers` method:

```
with IBSecurityService1 do
begin
  UserName := NewUserName;
  Password := NewPassword;
  AddUser;
end;
```

### 10.11.3 Updating User Details

The `ModifyUser` method is used to modify a users login details in the Security Database. The `UserName` property acts as the key identifying the user. The remaining properties listed above in 10.11.2 may be set as required to update the corresponding entry in the database. For example:

```
with IBSecurityService1 do
begin
  UserName := 'SYSDBA';
  FirstName := 'Donald';
  LastName := 'Duck';
  ModifyUser;
end;
```

### 10.11.4 Deleting a User

The `DeleteUser` method is used to remove a users login details from the Security Database. The `UserName` property acts as the key identifying the user. For example:

```
with IBSecurityService1 do
begin
  UserName := 'ALICE';
  DeleteUser;
end;
```

### 10.11.5 Setting the Auto Admin State

In Firebird 2.5, automatic SYSDBA mapping was given more control when operating under "Trusted Authentication" and with a new DDL instruction:

```
ALTER ROLE RDB$ADMIN SET/DROP AUTO ADMIN MAPPING
```

that may be used to configure this function on and off on a per server basis. This mapping can also be set through the services API and in IBX is performed using the Security Service's SetAutoAdmin method.

```
procedure SetAutoAdmin(Value: Boolean); {true = set auto admin mapping}
```

## 10.12  The User List DataSet

The TIBXServicesUserList is a subclass of a TMemDataset (from the FCP-db package). It is provided in order to provide a simple means of User Management through a dataset. The TIBXServicesUserList may be used as a data source for (e.g) a TDBGrid allowing all User Management to be performed by editing the rows in a grid.

It has the following required properties:

| | |
|---|---|
| `Source` | Reference to a TIBXSecurityService component that will be used as the user list source and to update the user credentials. |
| `FieldDefs` | A predefined set of Field definitions each corresponding to a user attribute in the server's security database. Do not remove or rename any FieldDef. However, the initial list may be added to if necessary. |

The predefined FieldDefs are:

| | | |
|---|---|---|
| UserID | Integer | Legacy Unix User ID |
| GroupID | Integer | Legacy Unix Group ID |
| SEC$USER_NAME | String(31) | User Name |
| SEC$FIRST_NAME | String(32) | First Name (info only) |
| SEC$MIDDLE_NAME | String(32) | Middle Name (info only) |
| SEC$LAST_NAME | String(32) | Last Name (info only) |
| SEC$PASSWORD | String(32) | Password.<br><br>Normally set to null. If a row is edited and the Password field is set to a non-null value then the user password is updated when row is posted.<br><br>Note: This field is not normally displayed in a TDBGrid and is intended to be updated by (e.g.) some change password dialog. |
| SEC$ADMIN | Boolean | If true then user has RDB$ADMIN role in security database. |

In use:

- Setting the dataset's Active property to true retrieves the user list from the server and populates the dataset (sorted by username) with the user attributes.

- Editing a dataset row permits user attributes to be changed. When the dataset's "Post" method is called, the changes are sent to the server.

- Adding a row is equivalent to adding a new user.

- Deleting a row is equivalent to deleting a user.

## 10.13  The Validation Service

The TIBXValidationService supports the invocation of various database repair actions, including validation and sweep. Exclusive access is required to the database in order to perform these validation activities.

Note: Limbo transaction resolution is provided by a separate service: TIBXLimboTransactionResolutionService (See 10.15).

The following properties must be set:

| | |
|---|---|
| ServicesConnection | Required: a reference to the TIBXServicesConnection providing the connection to the server. This must be in its connected state in order to perform database repair activities. |
| DatabaseName | Alias of or full pathname for database on the server. Note: this is not a connect string and does not include the servername, portno or protocol components. |
| Options | Used to request the repair activity and options. |

The following Database Repair services are available and selected by the service's options property:

| Title | Option | Description |
|---|---|---|
| Kill Shadow Files | KillShadows | Remove references to unavailable shadow files |
| Mend Database | MendDB | Mark corrupted records as unavailable, so subsequent operations skip them |
| Sweep Database | SweepDB | Request database sweep to mark outdated records as free space; |
| Validate Database | ValidateDB | Locate and release pages that are |

| | | allocated but unassigned to any data structures |
|---|---|---|
| Full Database Validation | ValidateFull | Check record and page structures, releasing unassigned record fragments. Use with Validate Database |
| Check Database | CheckDB | Request read-only validation of the database, without correcting any problem |
| Ignore all checksum errors | IgnoreChecksum | Refines database check |

In practice, the first four options are mutually exclusive, while the last three are options for "ValidateDB" while ValidateFull and IgnoreChecksum are options for "MendDB".

Once the repair action and options have been selected, the repair action is performed by the Execute method:

```
procedure Execute(OutputLog: TStrings);
```

When the method returns, the results, if any, reported for the repair action will have been added to the OutputLog.

## 10.14  The Online Validation Service

Online Validation was introduced in Firebird 2.5 that allows some consistency checks to be performed without exclusive access to the database. It may:

- validate some (or all) user tables in a database.

- validate some (or all) indices

Other ODS checks, such as Header/PIP/TIP/Generators pages, are not performed by Online Validation. The TIBXOnlineValidationService component is the IBX interface to online validation.

### 10.14.1 Properties

| ServicesConnection | Required: a reference to the TIBXServicesConnection providing the connection to the server. This must be in its connected state in order to perform database repair activities. |
|---|---|
| DatabaseName | Alias of or full pathname for database on the server. Note: this is not a connect string and does not include the servername, portno or protocol components. |
| IncludeIndexes | This is a regular expression selecting the indexes to include in online validation. Leave empty to include all indexes. |

| ExcludeIndexes | This is a regular expression selecting the indexes to exclude from online validation. Leave empty to exclude none. |
|---|---|
| IncludeTables | his is a regular expression selecting the tables to include in online validation. Leave empty to include all tables. |
| ExcludeTables | This is a regular expression selecting the tables to exclude from online validation. Leave empty to exclude none. |
| LockTimeout | This is the lock timeout in seconds. default is 10 secs. 0 is no-wait, -1 is infinite wait. |

The validation action is performed by the Execute method:

```
procedure Execute(OutputLog: TStrings);
```

When the method returns, the results, if any, reported for the online validation action will have been added to the OutputLog.

## 10.15  The Limbo Transaction Resolution Service

The TIBXLimboTransactionResolutionService may be used to list and fix any transactions remaining in limbo as a result of a failed two phase commit action across two or more databases. The following properties must be set:

| ServicesConnection | Required: a reference to the TIBXServicesConnection providing the connection to the server. This must be in its connected state in order to retrieve database statistics. |
|---|---|
| DatabaseName | Alias of or full pathname for database on the server.<br><br>Note: this is not a connect string and does not include the servername, portno or protocol components. |

### 10.15.1 Listing the In Limbo Transactions

There are two steps to the resolution of limbo transactions. The first step retrieves a list of all limbo transactions. The second step commits or rolls back each transaction as required.

The list of In Limbo Transactions is made available through two properties:

| LimboTransactionInfoCount | This returns the number of In Limbo Transactions for the identified database. |
|---|---|
| LimboTransactionInfo | This is an array property and for each index returns the TLimboTransactionInfo for an In Limbo Transaction. The array is zero based and has LimboTransactionInfoCount |

| | members. |
|---|---|

Accessing either of the above properties will cause the current list of In Limbo Transactions to be returned from the server and then cached locally.

For example:

```
var i: integer;
begin
  with IBXLimboTransactionResolutionService1 do
  begin
    for i := 0 to LimboTransactionInfoCount - 1 do
    with LimboTransactionInfo[i] do
    begin
      write('ID = ',ID);
      if MultiDatabase then
        write(', Multi DB')
      else
        write(' ,Single DB');
      write(', Host Site = ', HostSite);
      write(', Remote Site = ', RemoteSite);
      write(', Database Path = ', RemoteDatabasePath);
      write(', State = ', StateToStr(State));
      writeln(', Advise = ', AdviseToStr(Advise));
    end;
  end;
end;
```

where

```
function StateToStr(State: TTransactionState): string;
begin
  case State of
  LimboState:
    Result := 'Limbo';
  CommitState:
    Result := 'Commit';
  RollbackState:
    Result := 'Rollback';
  else
    Result := 'Unknown';
  end;
end;

function AdviseToStr(Advise: TTransactionAdvise): string;
begin
  case Advise of
  CommitAdvise:
    Result := 'Commit';
  RollbackAdvise:
    Result := 'Rollback';
  else
    Result := 'Unknown';
  end;
end;
```

This list identifies each limbo transaction and its current state, it also suggests an action (advises). The user can review the list and set the `TLimboTransactionInfo.Action` property to a desired outcome.

116

### 10.15.2 Fixing the In Limbo Transactions

The limbo transactions may then be resolved by setting the `GlobalAction` property and then calling the `Execute` method.

```
procedure Execute(OutputLog: TStrings);
```

The `GlobalAction` determines how `FixLimboTransactionErrors` processes the limbo transactions and may be set to:

| | |
|---|---|
| `CommitGlobal` | All limbo transactions are resolved by committing the transaction. |
| `RollbackGlobal` | All limbo transactions are resolved by rolling back the transaction. |
| `RecoverTwoPhaseGlobal` | All limbo transactions are resolved by performing a two phase commit of the transaction. |
| `NoGlobalAction` | Limbo transactions are resolved by either committing or rolling back the transaction, as specified by each limbo transaction's Action property. |

Each In Limbo transaction is processed in turn and the cached list of In Limbo Transaction is then emptied. Any text lines returned by the server is added to the OutputLog on return.

## 10.16 The Limbo Transaction List Dataset

The TIBXServicesLimboTransactionsList is a subclass of a TMemDataset (from the FCP-db package). It is provided in order to provide a simple means of In Limbo Transaction Resolution through a dataset. The TIBXServicesLimboTransactionsList may be used as a data source for (e.g) a TDBGrid allowing all In Limbo Transaction Resolution to be performed by editing the rows in a grid.

It has the following required properties:

| | |
|---|---|
| `Source` | Reference to a TIBXServicesLimboTransactionsList component that will be used as the user list source and to update the user credentials. |
| `FieldDefs` | A predefined set of Field definitions each corresponding to a In Limbo Transaction attribute. Do not remove or rename any FieldDef. However, the initial list may be added to if necessary. |

The predefined FieldDefs are:

| | | |
|---|---|---|
| TransactionID | Integer | Limbo Transaction ID |
| TransactionType | String(16) | 'Multi DB' or 'Single DB' |

| HostSite | String(256) | Server Domain Name |
|---|---|---|
| RemoteSite | String(256) | Client Domain Name |
| DatabasePath | String(256) | Database Name on Server |
| State | String(32) | 'Limbo'; 'Commit'; 'Rollback'; or 'Unknown'; |
| RecommendedAction | String(32) | 'Commit'; 'Rollback'; or 'Unknown'; |
| RequestedAction | String(32) | 'Commit' or 'Rollback' |

In use:

- Setting the dataset's Active property to true retrieves the In Limbo Transaction list from the server and populates the dataset with the Limbo Transaction attributes.

- Only the RequestedAction should be editable and constrained to one of the above values.

- Row Deletion is not permitted (ignored).

- Row Append is not permitted (exception raised).

Once the In Limbo Transaction Resolution options have been selected, the current list can be resolved by calling the TIBXServicesLimboTransactionsList method:

```
procedure FixErrors(GlobalAction: TTransactionGlobalAction; OutputLog: TStrings);
```

This method calls the TIBXLimboTransactionResolutionService.Execute method after setting the requested GlobalAction. On successful completion, any information returned by the server is added to the OutputLog and the dataset is refreshed.

# 11

# Personal Databases

A Personal Database is one held on the same system on the client and, where possible, file system access rights ensure that only the owner has access to the data. Instead of database access using a remote server running as a separate process, the server is embedded in the client and is both inherits and is constrained by the user's access rights.

In Firebird 2.5 and earlier, the embedded server is deployed as a separate package, while in Firebird 3 the same code libraries can be as part of a standalone server or as an embedded server.

The embedded server will be used if available and the database pathname is to a local file without a preceding server name. Access to the database will fail if the user has insufficient access rights.

The *fbintf* package provides direct and largely transparent support for use of the embedded server. See section 4.10 of the Firebird Pascal API Guide for more information. Deployment Guidelines are available in chapter 13 of the same guide.

IBX additionally recognises the case where a local database path has been specified but a database open error prohibits use of the embedded server. It will automatically prefix the database path with "localhost:" and try again hoping to use the local server, if available.

IBX also provides additional support for Personal Databases that are accessible via the embedded server.

## 11.1    TIBLocalDBSupport

`TIBLocalDBSupport` is non-visual component supporting a `TIBDatabase` and intended to simplify the use of the embedded firebird server for Personal Database Applications, on both Linux and Windows platforms. The `TIBLocalDBSupport` component supports GUI programs, while the `TIBCMLocalDBSupport` provides the same support for console mode programs. [Example applications](#) are provided for both GUI and console mode in the ibx/examples/local-employeedb directory.

When enabled, `TIBLocalDBSupport` provides:

- Verification that the embedded Firebird Server is in use.

- Setup of the FIREBIRD environment variable for the embedded server.

- DatabaseName, and login parameters management.

- Use of the Firebird Services API to initialise an empty local database from a gbak format Firebird archive.

- Use of the Firebird Services API to save the current local database to a gbak format Firebird archive.

- Use of the Firebird Services API to replace the contents of the current local database from a gbak format Firebird archive.

- Use of the `TIBXScript` Engine for automated field upgrade of the local database.

To use the component, simply drop it onto a form or data module and link it to the TIBDatabase.

## 11.1.1  Properties

| | |
|---|---|
| `Database` | reference to the `TIBDatabase` component for the local database |
| `DatabaseName` | filename (no path) to use for the Firebird Database file. |
| `EmptyDBArchive` | filename (optional path) holding the database initialisation archive. May either be absolute path or relative to <u>shared data directory</u>. |
| `Enabled` | when false component does nothing |
| `FirebirdDirectory` | Full path to directory holding firebird.conf. May either be absolute path or relative to the <u>shared data directory</u>. If empty, defaults to  shared data directory. |
| `Name` | Component Name |
| `Options` | <ul><li> iblAutoUpgrade: Automatically apply upgrade when database schema version is lower than required.</li><li>lblAllowDowngrade: Automatically apply downgrade when available to schema version compatible with the required version.</li><li>iblQuiet:  true then no database overwrite warnings</li></ul> |
| `RequiredVersionNo` | The schema version number required by the application. `TIBLocalDBSupport` will normally try to upgrade/downgrade the schema to satisfy this requirement. |

| | |
|---|---|
| `UpgradeConfFile` | Path to upgrade configuration file. May either be absolute path or relative to the <u>shared data directory</u>. |
| `VendorName` | Used to construct path to Database Directory. |

Note that at design time paths may use '/' or '\' as directory separator. At run time, they must be specified using the appropriate Directory Separator for the current platform.

## 11.1.2  Events:

| | |
|---|---|
| `OnGetDatabaseName` | The database path name is normally computed automatically. However, this event allows an application to inspect and override the result. |
| `OnNewDatabaseOpen` | called after the successful initialisation of an empty local database. |
| `OnGetDBVersionNo` | called to get the current database schema version number. If this event is not handled then schema upgrade/downgrade is never performed. |
| `OnGetSharedDataDir` | The shared data directory is normally computed automatically. However, this event allows an application to inspect and override the result. |

## 11.1.3  Shared Data Directory

The shared data directory is the base directory for all static data files used by `TIBLocalDBSupport`. This is determined as follows:

- Windows: the application executable's location.

- Unix: the application executable's location unless this is /usr/bin, /usr/local/bin, /usr/sbin or /usr/local/sbin, when the shared data directory is set to /usr/share/<application name> or /usr/local/share/<application name> depending on whether the application is in /usr or /usr/local.

  Note: that the <application name> is taken from sysutils.ApplicationName and defaults to the filename of the application executable less any extension.

## 11.1.4  DatabaseName, and login parameters management

When `TIBLocalDBSupport` is in use, the `TIBDatabase.DatabaseName` property is ignored and instead, it is generated algorithmically as:

- Windows: "User Application Directory"\VendorName\DatabaseName

- Unix: "User Home Directory"/."VendorName"/DatabaseName

The "DatabaseName" comes from the `TIBLocalDBSupport.DatabaseName` property.

The "VendorName" comes from the `TIBLocalDBSupport.VendorName` property. If the `TIBLocalDBSupport.VendorName` property is left empty then `Sysutils.VendorName` is used. If this is empty then no VendorName component is present in the path.

Note the use of a hidden directory under Unix.

If the generated DatabaseName is not appropriate then the `TIBLocalDB.OnGetDatabaseName` event handler gives a chance to inspect it and change it to something different.

The Database Params are copied from the `TIBDatabase` component except that the "user_name" and "password" parameters are removed if present. When running under Windows, the "user_name" is then set to "SYSDBA" and the "password" to "masterkey". Under Unix, these parameters are omitted.

### 11.1.5  Database Initialisation

When the linked `TIBDatabase` connected property is set to "true", `TIBLocalDBSupport` generates the DatabaseName (as described above) and then if it does not correspond to an existing file, `TIBLocalDBSupport` uses the Firebird Services API to create the database file from an "empty database" archive in gbak format, or an SQL Script. In practice, the archive can contain both the database metadata and initial table data. An SQL Script may also contain data inline compatible with TIBXScript (see 7.1)

The "empty database" archive is given by the `TIBLocalDBSupport.EmptyDBArchive` property. This should be a filename (with the .gbk extension or .sql extension) and may include an optional path. Relative paths are interpreted as relative to the shared data directory.

For a gbak archive, the Services API (see 10.6) is then used to create the initial database from this archive. The IBXScript component is used to create a database from an SQL Script. An error is raised if the archive is not present.

The SQL script may or may not include a CREATE DATABASE SQL statement. If it does then the file name is replaced with the required filename for the database. If no such statement is present then one is generated using the required filename for the database and the character set specified in the database parameters as the connection character set.

The local database can be re-initialised at any time by calling the `TIBLocalDBSupport.NewDatabase` method.

### 11.1.6  Saving the Current Database

The current database contents can be saved at any time by a call to `TIBLocalDBSupport.SaveDatabase`. The filename for the archive can be provided in the method call. If empty, then the user is prompted to enter a filename (default extension .gbk).

The Services API (see 10.5) is then called to archive the database to the specified file in gbak format.

### 11.1.7 Restoring the Database from an Archive

The local database can be overwritten (restored) from any archive in gbak format (including those saved using the `SaveDatabase` method) by calling the `TIBLocalDBSupport.RestoreDatabase` method. The filename for the source archive can be provide in the method call. If empty, then the user is prompted to locate the file.

The Services API (see 10.6) is then called to restore the local database from the archive.

### 11.1.8 Database Schema Upgrade

A Software Application Update can also require a corresponding update to the database schema. With embedded Firebird server applications where the user may not even be aware that a database server is in use, it is important to have a means to field upgrade the database schema in as seamless and automatic a manner as possible. `TIBLocalDBSupport` supports a suitable mechanism using the `TIBXScript` engine (see 7.1).

The underlying idea is that the database schema comes with a version number given as a single integer. The first version to be released is version 1, the second is version 2 and so on. The current schema version number must be saved as data somewhere in the database. As this is database schema dependent, TIBLocalDBSupport does not know how to determine the current database schema number and instead relies upon the application responding to the OnGetDBVersionNo event.

Each version of an application will have a maximum and minimum version of the database schema that it can support, and it is expected to check that the schema version is acceptable in its `TIBDatabase OnConnect` handler. However, before this handler is called, `TIBLocalDBSupport` will itself check the current schema version against its `RequiredVersionNo` property (which should be set to the maximum supported schema version no).

- If iblAutoUpgrade is given in the Options property and the current schema is less than the Required Version no., then `TIBLocalDBSupport` will attempt to apply the upgrade rules to raise the version number to that required.

- If iblAllowDowngrade is given in the Options property and the current schema is greater than the Required Version no., then `TIBLocalDBSupport` will attempt to locate a suitable backup archive and restore this as the current database. This case is usually only found in the unlikely event of a failed upgrade and the user has installed an older version of the software in order to recover from the problem.

The schema upgrade rules are read from the upgrade configuration file. This is a text file in "ini" file format with the following sections:

`[status]`

This should have a single named value "current" giving the current database schema number as in integer e.g.

`current = 2`

This should normally be set to the same value as the RequiredVersionNo property and acts as a check to ensure that both are in sync.

```
[Version.nnn]
```

Where nnn is an integer with leading zeroes. For example, "Version.002" is the section read to upgrade the database schema from version 1 to version 2. This section can contain the following named values:

| Name | Type | Use |
|------|------|-----|
| Upgrade | string | Name and optional path to the SQL script used to perform the upgrade. May either be absolute path or relative to the upgrade configuration file. Either forwards or back slashes may be used as the path delimiter. |
| Msg | string | Text message displayed in progress dialog while script is active. Defaults to "Upgrading Database Schema to Version nnn". |
| BackupDatabase | yes/no | If present and set to "yes" then a database backup in gbak format is made before the upgrade is performed. The backup file is located in the same directory as the database file and is given the same name as the database file with the extension replaced with ".nnn.gbak". Where "nnn" is the current schema version number (i.e. prior to running the upgrade script). |
| <Parameter Name> | string | Name and optional path to binary data file. May either be absolute path or relative to the upgrade configuration file. Either forwards or back slashes may be used as the path delimiter. |

For example:

```
[Version.002]
Msg = Upgrading to Version 2
BackupDatabase = yes
Upgrade = patches/02-patch.sql
mugshot = images/man.png.gz
```

Note that in the above, "mugshot" is intended to be used to resolve an Update, Insert or Delete query parameter in the 02-patch.sql file. E.g.

```
   Update EMPLOYEE Set Photo =:MUGSHOT Where Emp_no = 2;
```

This is only applicable to BLOB columns and the above is interpreted as update the EMPLOYEE table where the Emp_no is "2" and set the value of the Photo column to the binary data contained in the file "images/man.png.gz". The ".gz" extension is recognised as a gzip compressed file and decompressed before updating the table.

When the current database schema is more than one version number less than that required, the upgrade rules are applied iteratively to upgrade the database to the required schema version.

## 11.2   Local EmployeeDB Example

The purpose of this example is to demonstrate the use of the `TIBLocalDBSupport` component. This component is used with a `TIBDatabase` when the database is accessed using the Firebird Embedded Server. `TIBLocalDBSupport` takes care of checking the environment and setting up FIREBIRD environment variables and DB parameters. It also supports initialisation of the local database from an archive in gbak format, plus save and restore of the local database. It can also run SQL scripts to upgrade the database schema when a new software version is released.

The example can be found under: ibx/examples/local-employeedb/project1.lpi

See also [console mode](#).

Before compiling and running the example, the Firebird embedded server must be installed. The Chapter 13 of the Firebird Pascal API Guide for deployment guidelines for the embedded server.

### 11.2.1   Running the application

The example should just compile and run. An archive of the Firebird example employee database is provided with the example. This will be used to create the initial database. It should then be automatically upgraded to "version 2" using the scripts provided in the "patches" directory. (see also the file upgrade.conf).

Note that you will not be prompted for a username/password. The embedded server uses normal file permissions to control access. Otherwise you can edit the employee database as in the client/server version.

The local database will be created in:

- Linux: $HOME/.MWA Software/employee.fdb

- Windows: <User Application Data Folder>\MWA Software\employee.fdb

The File menu provides actions to save the current database to a gbak format archive, restore it again (replacing the current database) or to restore the database to its initial state.

### 11.2.2   Console Mode

A console mode version of the example application is also provided under ibx/examples/local-employeedb/ConsoleModeExample.lpi.

Like all IBX console mode applications, this uses the ibexpressconsolemode package. The IBCMLocalDBSupport unit is used to provide the TIBCMLocalDBSupport compoent.

The application is similar to the above and uses the same archive database and upgrade scripts. Instead of displaying the employee table, when run it will print out the first two rows.

# 12

# **The IBX Controls**

IBX 1.2 introduced a new Component Palette entry "Firebird Data Controls". This has four new data aware controls dependent on IBX and which make use of the SQL Parser (see 7.3). In IBX2, TIBArrayGrid was added.

The IBX Controls are:

- TIBLookupComboEditBox
- TIBDynamicGrid
- TIBTreeview
- TDBControlGrid
- TIBArrayGrid

**TIBLookupComboEditBox** is a TDBLookupComboBox descendent that implements "autocomplete" of typed in text and "autoinsert" of new entries. Autocomplete uses SQL manipulation to revise the available list and restrict it to items that are prefixed by the typed text (either case sensitive or case insensitive). Autoinsert allows a newly typed entry to be added to the list dataset and included in the available list items.

**TIBDynamicGrid** is a TDBGrid descendent that provides for:

- automatic resizing of selected columns to fill the available row length

- automatic positioning and sizing of a "totals" control, typically at the column footer, on a per column basis.

- DataSet resorting on header row click, sorting the dataset by the selected column. A second click on the same header cell reversed the sort order.

- Support for a "Panel Editor". That is on clicking the indicator column, the row is automatically expanded and a panel superimposed on it. The panel can have any number

of child controls, typically data aware controls with the same datasource as the grid allowing for editing of additional fields and more complex editors.

- Reselection of the same row following resorting.

- A new cell editor that provides the same functionality as TIBLookupComboEditBox. Its properties are specified on a per column basis and allows for one or more columns to have their values selected from a list provided by a dataset. Autocomplete and autoinsert are also available. The existing picklist editor is unaffected by the extension.

**TIBTreeView** is a data aware TCustomTreeView.

**TDBControlGrid** is a lookalike rather than a clone for the Delphi TDBCrtlGrid. TDBControlGrid is a single column grid that replicates a TWinControl - typically a TPanel or a TFrame in each row. Each row corresponds to a row of the linked DataSource. Any data aware control on the replicated (e.g.) TPanel will then appear to have the appropriate value for the row.

**TIBArrayGrid** is a data aware control derived from TCustomStringGrid and which may be used to display/edit the contents of a one or two dimensional Firebird array Field.

Examples are provided to illustrate the use of the new controls.

NOTE: all the dynamic IBX Controls (TIBLookupComboEditBox, TIBDynamicGrid and TIBTreeview) manipulate the SQL "behind the scenes". If you are using parameterised  queries in your select SQL then it is important that parameter values are set in the dataset's BeforeOpen event handler. This is guaranteed to be called whenever the SQL is updated and the query re-opened and locating setting the parameter values here ensures that they are set correctly each time.
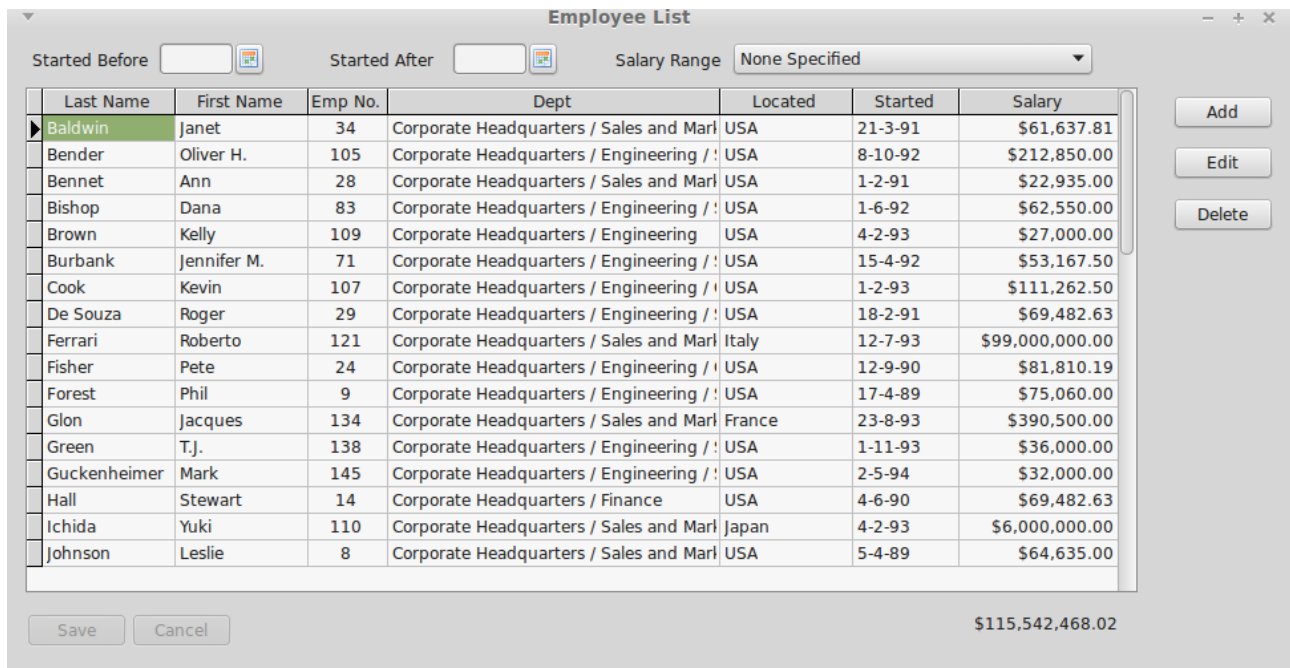
## 12.1   TIBDynamicGrid



**Illustration 11: The TIBDynamicGrid**

The TIBDynamicGrid is illustrated above using Firebird's example "employee" databases.

In use, it looks just like a TDBGrid and is a TDBGrid descendent. Any project that uses IBX and TDBGrid can thus be quickly converted to using TIBDynamicGrid. The control uses SQL Manipulation to manage column sorting.

The above example can be found in "ibx/examples/employee" and illustrates most of the benefits of TIBDynamicGrid.

- Resize the form and you will see how the "Dept" column automatically grows/shrinks to ensure that the grid always fills the available space and how the Salary "Total" control (TDBText) moves so that it is always aligned with the grid. Column resizing is controlled at design time by setting the AutoSizeColumn property for each column that it is to be dynamically resized, with its design time width interpreted as the minimum column width. All other column widths remain unchanged.

- Click on the "Started" column header (or any other column header) and the table will be resorted by that column. A second click on the same header reverses the sort order.

- Select a row and press "F2", or click on "Edit" or the left hand indicator column and the Editor Panel is revealed (See Illustration 12). This allows the row to be edited free of the constraints imposed by a simple column editor.

- After reopening the dataset (e.g. after a re-sort or change of filters) the previously selected row is automatically reselected.

- The filters, such a "salary range", also illustrate how the new IB SQL Parser works with the TIBDynamicGrid. For example, where a salary range is selected, the dataset is re-opened and the filters are applied in the BeforeOpen event handler.

- Each row can still be edited without having to open the panel editor. The column "located" is an example of the use of TIBLookupComboEditBox as a column editor. Note that the country list is dynamically generated and varies according to Job Code (an Employee Database constraint).

**Illustration 12: TIBDynamicGrid with an Editor Panel Visible**

### 12.1.1   Column Properties

Most of `TIBDynamicGrid's` new features are accessed via the column editor and are properties of each column in the grid. The new column properties are given below.

| | | |
|---|---|---|
| `AutoSizeColumn` | Boolean | If true then the column is automatically resized to fill the grid. More than one column can have this property set to true. |
| `ColumnTotalsControl` | TControl | Optional. Used to identity a control (typically a TDBEdit or TDBText) to be kept in vertical alignment with the column, and to have the same width. <br><br> Note that the horizontal positioning is unaffected by grid resize, and hence the total can be placed either above or below the grid. |
| `InitialSortColumn` | Boolean | Identifies the column used to sort the grid when the dataset is first opened. |
| `DBLookupProperties` | TDBLookupProperties | These properties are copied to a TIBLookupComboBox when it is used as a column editor. Setting TDBLookupProperties.ListSource implictly requests this as the column editor instead of a normal pick list. <br><br> If the  TDBLookupProperties.DataFieldName is not set then the control works as a "pick list" with |

| | | its values taken from the List Source DataSet.<br><br>If the TDBLookupProperties.DataFieldName is set then it works as full lookup list. The DataFieldName identifies a field in the parent TIBDynamicGrid.DataSource.DataSet. This field does not have to be visible in the grid. When the editor completes, the identified field is set to the value of the List Source field identified by TDBLookupProperties.KeyField. |
|---|---|---|

## 12.1.2 TIBDynamicGrid New Properties

| EditorPanel | TControl | When set, this control (typically a TPanel or TFrame) is used as the Editor Panel (see below). |
|---|---|---|
| ExpandEditorPanelBelowRow | Boolean | When set and an editor panel is displayed, the row height is set to the current row height plus the panel height and the Editor Panel placed under the row. That is, the original row is still displayed with the editor panel beneath it. The default is that the editor panel appears to replace the row. |
| AllowColumnSort | Boolean | Enables column sorting by column header click (default true). |
| Descending | Boolean | Determines the initial sort order. Default is false i.e. ascending sort order. |
| DefaultPositionAtEnd | Boolean | Determines the initially selected row when the dataset is first opened. If true then the last row is selected, otherwise the first row. Default: false. |
| IndexFieldNames | String | This is a semi-colon separated list of one or more dataset fieldnames. Typically this is the primary key for the dataset. Used for automatic reselection of rows after the dataset is reopened.<br><br>A property editor is available for design time field name selection. |

### 12.1.3 TIBDynamicGrid new Events

| | |
|---|---|
| `OnBeforeEditorHide` | This event is called before the Editor Panel is hidden. Can be used to validate changes. |
| `OnEditorPanelShow` | This event is called after the Editor Panel is made visible |
| `OnEditorPanelHide` | This event is called after the Editor Panel is hidden. Can be used to do any additional tidying up needed. |
| `OnKeyDownHander` | The TIBDynamicGrid uses a KeyDown handler to intercept edit keys while the Editor Panel is active. For example, to process an "escape" key as a cancel edit. You can write your own keydown handler to modify this behaviour. |
| `OnColumnHeaderClick` | Called when a column header is clicked and before the dataset is re-sorted. Can be used to modify the column index for the sort. |
| `OnUpdateSortOrder` | Called when the dataset select SQL is being modified prior to resorting the dataset. Can be used to modified the SQL "Order by" clause. e.g. to add a subsort column. For example, useful when one column has a "year" and the next column is the "month". Clicking on "year" can then made to subsort on "month". Can also return an empty string in order to prevent sorting of the dataset. |
| `OnRestorePosition` | Called when the dataset is opened and may be used to override the initially selected record. The event provides a read/write argument (Location) that is an array of variants. This is either an empty zero length array or contains the same number of elements as there are indexnames (See IndexFieldNames property). In the latter case, it contains the index key values for the previously selected row (i.e. when the dataset was last closed). The first time the dataset is opened the array is empty.<br><br>The location can be inspected and replaced by an alternative location (index key values) or set to empty. In the former case, the grid will attempt to locate the selected row. In the latter case, the default position is selected (see DefaultPositionAtEnd property). |

### 12.1.4 The Editor Panel

An Editor Panel may be any `TControl` available on the form. However, in practice, it is typically either a `TPanel` or a `TFrame`. The example shows a `TPanel` being used as an Editor Panel.

You can create an Editor Panel by simply dropping it on to the same form as the `TIBDynamicGrid` and then selecting it as the value of the `TIBDynamicGrid.EditorPanel` property.

To be useful, the Editor Panel should be populated with data aware controls that use the same DataSource as the grid and are individually used to edit fields in the same row. The height of the panel should be the minimum necessary as this will determine the row height when it is visible.

At run time, the Editor Panel is automatically hidden until called into use by either:

a) Pressing "F2" when the Dynamic Grid has the focus.

b) Clicking on the left hand indicator column, or

c) Calling the `TIBDynamicGrid.ShowEditorPanel` method.

In order to show the editor panel, the following actions are performed by the `TIBDynamicGrid`:

- The current row is resized to the height of the Editor Panel.

- The Editor Panel is resized and repositioned so that it fits exactly over the current row.

- The Editor Panel is made visible.

The current row can now be edited using the child controls on the Editor Panel – that is as long as their DataSource is the same as the grid's.

The Editor Panel is hidden (and any changes Posted to the DataSet) when:

a) A different row is selected by the mouse or up/down arrow keys

b) The Escape Key is Pressed (cancels the changes)

c) "F2" is pressed.

d) The `TIBDynamicGrid.HideEditorPanel` method is called.

Once the Editor Panel is hidden, the current row is re-sized back to its correct height.

## 12.1.5 Setting Query Parameters

The dataset used as the TIBDynamicGrid's data source may have a select query that contains query parameters. However, in order to perform column sorting, TIBDynamicGrid manipulates the SQL query "behind the scenes" to change the "order by" clause. Because of this, the only "safe" place to set values for query parametersis in the in dataset's "BeforeOpen" event handler. This is guaranteed to be called every time the grid updates the SQL order by clause and re-executes the query.

Parameter values set before the dataset is opened and outside of the "BeforeOpen" event handler will be lost when the grid updates the SQL and reset to the default null value.

## 12.2 TDBControlGrid



**Illustration 13: Example Control Grid**

TDBControlGrid is a lookalike rather than a clone for the Delphi TDBCrtlGrid. TDBControlGrid is a single column grid that replicates a TWinControl - typically a TPanel or a TFrame in each row. Each row corresponds to a row of the linked DataSource. Any data aware control on the replicated (e.g.) TPanel will then appear to have the appropriate value for the row.

Unlike the Delphi TDBCtrlGrid, there are no restrictions on which controls can be used on the replicated panel. In principle, any visual control may be used. The "csReplicable" property is not used by TDBControlGrid. However, there can be performance issues with a large number of controls on the panel or when there is a high latency to draw one or more controls.

To use the new control, simply drop it on to a form at design time and size it appropriately. Then separately drop a TPanel on to the same form and populate it with appropriate child controls, typically data aware controls using the same DataSource.

Now link to TDBControlGrid DrawPanel property to this panel. The panel should then be repositioned as a child control of the TDBControlGrid and occupying the top and only row of the grid. The row height should be set to the panel height and the panel width will the set to the with of the grid row. The panel can be unlinked at any time.

Now set the TDBControlGrid.DataSource to the common data source for the controls on the panel.

**Important** Note: It is strongly recommended *not* to open the source DataSet for a DBControlGrid during a Form's "OnShow" event handler. Under GTK2 this is known to risk corrupt rendering of row images when the control is first displayed. If necessary use "Application.QueueAsyncCall" to delay opening of the dataset (see DBControlGrid examples) until the Form"s Window has been created. See the example application.

When you build and run your project and open the DataSource's dataset, the `TDBControlGrid` should show a row for each row in the dataset and the child controls on each row should have the appropriate values for the row.

When the grid has the focus, you can move between rows using the up and down arrow keys, page Up and Page Down, Ctrl+Home and Ctrl+End jump to beginning and end respectively. You can also use the mouse to change between rows, either by clicking on a row or the scroll bar.

Pressing the down arrow key on the last row should append a new row – as long as the "Disable Insert" `TDBControlGrid.Option` is not selected.

All rows may be edited in situ. Moving between rows should automatically post the changes. The "escape" key may be used to cancel row edits before they are posted.

A row may be deleted by calling the underlying DataSet's Delete method.

See the TDBControlGrid example code for guidance on how to use the control. This example requires IBX and uses the Firebird example employee database.

## 12.2.1 TDBControlGrid Properties

| DrawPanel | TWinControl | This control will be replicated for each row in the DataSet. Typically a TPanel or a TFrame. |
|---|---|---|
| Options | TPanelGridOptions | Similar to a TDBGrid, but limited to:<br><br>• Cancel On Exit<br><br>• Disable Insert<br><br>• Show Indicator Column |
| DataSource | TDataSource | A row is replicated for every row in this dataset. |
| DefaultPositionAtEnd | Boolean | When the dataset is opened then it is initially positioned at the last record if this property is true, |

## 12.2.2 TDBControlGrid Events

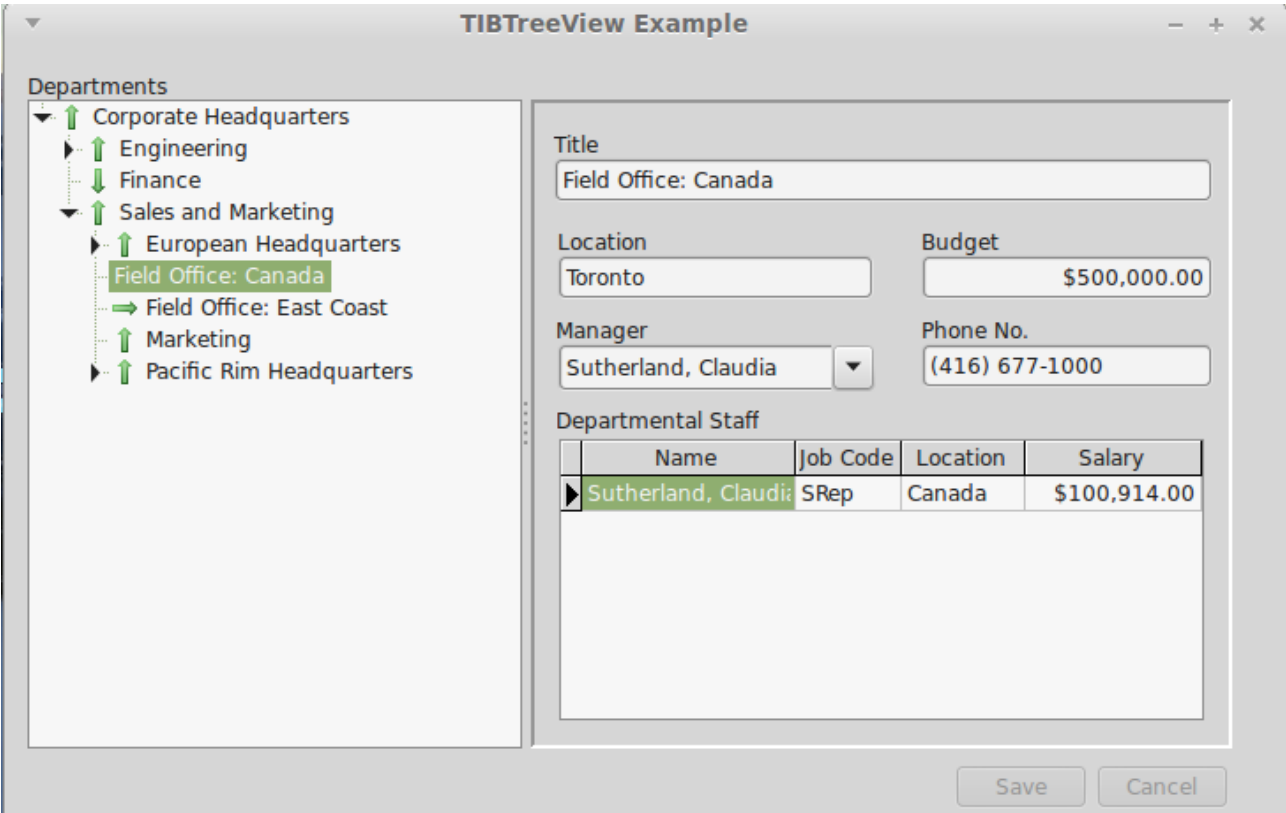| OnKeyDownHander | The TDBControlGrid uses a KeyDown handler to intercept edit keys while the Draw Panel is active. For example, to process an "escape" key as a cancel edit. You can write your own keydown handler to modify this behaviour. |
|---|---|

## 12.3 TIBTreeView



**Illustration 14: TIBTreeView Example**

`TIBTreeView` is a data aware descendent of a `TCustomTreeView` and is used to present a hierarchically organised data set in a tree view. Tree Node Insertion, Deletion and Modification are supported, as is moving (e.g. using drag and drop) nodes from one part of the tree to another. The underlying dataset cursor is always positioned to reflect the currently selected tree node. It can thus be used to select a row for detailed editing. SQL Manipulation is used to load the tree as a series of separate queries.

Illustration 14 Is taken from ibx/examples/ibtreeview and uses the Firebird example "employee" database. This database contains a hierarchically organised table "DEPARTMENT" and which is used for the example.

To use a `TIBTreeView`, simply drop it on to a form, set the `DataSource` property, and, as a minimum, the `TextField, ParentField` and `KeyField` properties as defined below.

The DataSet must have a single primary key field.

### 12.3.1 TIBTreeView Properties

| DataSource | TDataSource | Identifies the source of the data to present using the tree view |
|---|---|---|
| TextField | string | The field name of the column used to source each node's display text |

| KeyField | string | The field name of the column used to source each node's primary key. |
|---|---|---|
| ParentField | string | The field name of the column used to identify the primary key of the parent row. This field is null for a root element. |
| HasChildField | string | Optional. The field name of the column used to indicate whether or not the row has child nodes. When present, the field should return an integer value with non-zero values implying that child nodes exist. |
| RelationName | string | Optional. The Child Field is typically the result of joining the table to itself and is a count of child rows. However, this can result in ambiguous column names when the SQL is manipulated. This property should contain the Table Alias used to select the Key, Text and Parent Fields (see example application). |
| ImageIndexField | string | Optional. If specified then the image index for each node is read from this (integer) field. |
| SelectedIndexField | string | Optional. If specified then the selected image index for each node is read from this (integer) field. |

## 12.3.2 TIBTreeView Methods

```
function GetNodePath(Node: TTreeNode): TVariantArray
```

Returns a Variant array containing the primary key values of the Node and its parents from the root node downwards.

```
function FindNode(KeyValuePath: TVariantArray; SelectNode: boolean): TIBTreeNode;
```

Returns the `TTreeNode` identified by the `KeyValuePath`. The `KeyValuePath` is an array comprising a list of primary key values walking the tree down from the root node to the requested node.

If `SelectNode` is true then the returned node is also selected.

This function can be used to select the tree node using the node path returned by an earlier call to the function `GetNodePath`.

```
function FindNode(KeyValue: variant): TIBTreeNode;
```

Returns the tree node with the primary key given by `KeyValue`. Note: this forces the whole tree to be loaded by a call to `TCustomTreeView.FullExpand`.

### 12.3.3 Drag and Drop

Drag and drop is supported by `TCustomTreeView` without the need for additional support from `TIBTreeView`. In the example, drag and drop is enabled by:

- DragMode set to automatic

- The `OnDragOver` Event handled by:

```
procedure TForm1.IBTreeView1DragOver(Sender, Source: TObject; X, Y: Integer;
  State: TDragState; var Accept: Boolean);
begin
  Accept := Source = Sender
end;
```

- The `OnDragDrop` Event Handled by:

```
procedure TForm1.IBTreeView1DragDrop(Sender, Source: TObject; X, Y: Integer);
var Node: TTreeNode;
    tv: TTreeView;
begin
  if Source = Sender then {Dragging within Tree View}
  begin
    tv := TTreeView(Sender);;
    Node := tv.GetNodeAt(X,Y); {Drop Point}
    if assigned(tv.Selected) and (tv.Selected <> Node) then
    begin
      if Node = nil then
        tv.Selected.MoveTo(nil,naAdd) {Move to Top Level}
      else
      begin
        if ssCtrl in GetKeyShiftState then
        begin
          Node.Expand(false);
          tv.Selected.MoveTo(Node,naAddChildFirst)
        end
        else
          tv.Selected.MoveTo(Node,naInsertBehind)
      end;
    end;
  end;
end;
```

Note that the above applies the convention that if the "control" key is held down while the node is "dropped" then it is added as a child node. Otherwise, it is added as a sibling.

### 12.3.4 Setting Query Parameters

The dataset used as the TIBTreeView's data source may have a select query that contains query parameters. However, in order to determine child nodes, TIBTreeView manipulates the SQL query "behind the scenes" to change the "Where" clause in order to select only the current node and its child nodes. Because of this, the only "safe" place to set values for query parameters is in the in dataset's "BeforeOpen" event handler. This is guaranteed to be called every time the control updates the SQL where clause and re-executes the query.

Parameter values set before the dataset is opened and outside of the "BeforeOpen" event handler will be lost when the control updates the SQL and reset to the default null value.

## 12.4   TIBLookupComboEditBox

TIBLookupComboEditBox is a TDBLookupComboBox descendent that implements "autocomplete" of typed in text and "autoinsert" of new entries.

- Autocomplete uses SQL manipulation to revise the available list and restrict it to items that are prefixed by the typed text (either case sensitive or case insensitive).

- Autoinsert allows a newly typed entry to be added to the list dataset and included in the available  list items.

Although TDBLookupComboBox also supports auto-complete, the benefit of using TIBLookupComboEditBox comes with long lookup lists as typing in one or more characters forces the list to be queried again and restricted to list members beginning with the same characters. The list of alternatives becomes much shorter.

Auto-insert normally uses the list dataset's insert query to add a new row and depends upon the dataset's "After Insert" event handler to set the other fields of the row to appropriate values and/or the generator assigned to the dataset.

### 12.4.1  TIBLookupComboEditBox Example



**Illustration 15: Using the TIBLookupComboEditBox**

The above example can be found in ibx/examples/lookupcombobox and uses the Firebird "employee" example database. The "Employee Name" is a TIBLookupComboEditBox and is used here to:

a)  Select an employee record for editing

b)  Initiate the entry of a new employee record.

139

First, you should explore the use of the new control. Click on the drop down arrow and a drop down list of all employee names (in lastname/firstname syntax) will be shown. This is typically longer than can be displayed on a single screen.



**Illustration 16: Selection of a Different Employee**

Now close the drop down list, select all characters in the Employee Name edit box and enter "pa". After a short (600ms) delay, after you stop typing, the employee details should change to that shown in Illustration 16 i.e. for the first employee with a lastname beginning with "pa", i.e. Mary Page.

Of course, auto-complete to the first employee beginning "pa" may not get the actual employee you want. Now click on the drop down list and this will show all employees with a last name starting with "pa". This is a much shorter list than the full list and allows you to quickly focus in on the employee you want.
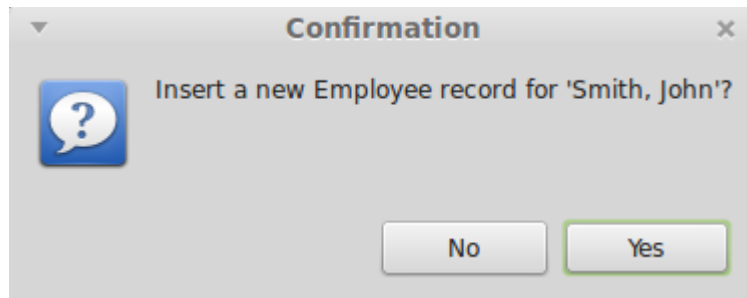
Indeed, this can also be done from the keyboard. Start again, and enter "pa", now press the down arrow and you can cycle quickly through all employees starting "pa". The up arrow also works. Use the Enter key to select the employee record.

Alternatively, after entering "pa" and seeing the entry for Mary Page, then press "r" to extend the entry to "par" and you get the record for Bill Parker.

To return to the full list, just press the escape key while the control has the focus.

#### 12.4.1.1 Auto-insert

Auto-insert allows quick insertion of new employee records. For example, start by selecting all text in the Employee Name edit box and enter the name of the new employee (e.g. Smith, John), and press the "Enter" key. You should now get a prompt confirming the entry of the new employee record:

If you click on "yes" then a new employee record is created and displayed as show below.



**Illustration 17: New Employee Record**

The employee name is parsed from the text entered into the Employee Name box. The remaining fields come from defaults taken from the "OnInsert" event handler. You can now amend the defaults as required.

### 12.4.2 TIBLookupComboEditBox Properties

TIBLookupComboEditBox inherits TDBLookupComboBox properties. In addition, it defines:

| AutoInsert | Boolean | Set to true to enable auto-insert |
|---|---|---|
| AutoComplete | Boolean | Default: true in TIBLookupComboEditBox |
| KeyPressInterval | Integer | Delay in milliseconds between last key press and auto-complete (Default: 500ms). |
| RelationName | String | TIBLookupComboEditBox updates the "Where" clause in the ListSource select SQL query in order |

| | | to refine the list, and uses the value of the "ListField" property as the column name. If this name is ambiguous in the SQL query then the "RelationName" property must be set to the name of the table or table alias to qualify the column name and remove the ambiguity. |
|---|---|---|

### 12.4.3 TIBLookupComboEditBox Event Handlers

| OnAutoInsert | TIBLookupComboEditBox will normally use the ListSource's Insert query to perform auto-insert. If this is not possible or inappropriate then an OnAutoInsert handler must be provided to perform the insertion. The handler is provided with the value of the display text to insert and must return the new key value. |
|---|---|
| OnCanAutoInsert | This handler is called immediately before auto-insertion is performed and is typically used to validate the insert and obtain user agreement (e.g. via a dialog box).  The handler is provided with the value of the display text to insert and must set the "Accept" boolean on return to true to accept the insert or to false to reject it. |

### 12.4.4 Setting Query Parameters

The dataset used as the TIBLookupComboEditBox data source may have a select query that contains query parameters. However, in order to perform auto-complete, TIBLookupComboEditBox manipulates the SQL query "behind the scenes" to change the "Where" clause in order to select only the rows matching the current text (as a prefix). Because of this, the only "safe" place to set values for query parameters is in the in dataset's "BeforeOpen" event handler. This is guaranteed to be called every time the control updates the SQL where clause and re-executes the query.

Parameter values set before the dataset is opened and outside of the "BeforeOpen" event handler will be lost when the control updates the SQL and reset to the default null value.

## 12.5   TIBArrayGrid

TIBArrayGrid is a visual control that can be linked to a `TIBArrayField` and used to display/edit the contents of a one or two dimensional Firebird array. It may be found in the "Firebird Data Controls" palette.

To use a `TIBArrayGrid`, simply drop it onto a form and set the `DataSource` property to the source dataset and the `DataField` property to the name of an array field. The grid should then be automatically sized to match the dimensions of the array.

Note that the array bounds can be refreshed at any time in the IDE, by right clicking on the control and selecting "Update Layout" from the pop up menu.

At runtime, the `TIBArrayGrid` will always display/edit the value of the array element in the current row. If this element is null then the array is empty. However, data can be inserted into an empty array. When the row is posted, the field will be set to the new/updated array.

### 12.5.1 Properties

Most `TIBArrayGrid` properties are the same as for `TStringGrid`. The following are specific to `TIBArrayGrid`. Note that you cannot set the Row or column counts directly as these are always set to match the array field.

**Public Properties**

| | |
|---|---|
| `ArrayIntf` | Provides direct access to the array itself. |
| `DataSet` | The DataSet provided by the DataSource (read only). |
| `Field` | The source field |

**Published:**

| | |
|---|---|
| `DataField` | The name of the array column. |
| `DataSource` | The data source providing the source table. |
| `ReadOnly` | Set to true to prevent editing |
| `ColumnLabels` | A string list that provides the labels for each column in the grid. Provide one line per column. If non empty then a column label row is created as a fixed row at the top of the grid. |
| `ColumnLabelAlignment` | Sets the text alignment for column Labels |
| `ColumnLabelFont` | Sets the font used for column labels |
| `RowLabels` | A string list that provides the labels for each row in the grid. Provide one line per row. If non empty then a row label column is created as a fixed column to the left of the grid. |
| `RowLabelAlignment` | Sets the text alignment for row Labels |
| `RowLabelFont` | Sets the font used for row labels |
| `RowLabelColumnWidth` | Width of the Fixed Column used for row labels. |
| `TextAlignment` | Alignment of all cells other that those containing labels. |

## 12.5.2  Examples

Example applications are provided for both one and two dimensional arrays. In each case, the example applications create their own database and populate it with test data when first run. Note that you will typically need to run the application before accessing database properties in the IDE. This is in order to create the database referenced by the IDE.

### 12.5.2.1  Database Creation

The `TIBDatabase` property "`CreateIfNotExists`" is set to true in both examples. This means that if the database does not exist when an attempt is made to connect to it then the database is created. After it is created, the "`OnCreateDatabase`" event handler is used to add a table to the newly created database and to populate it with test data. The application then continues as if the database already existed.

By default, the database is created in the defined temporary directory. This behaviour can be overridden by editing the example's "unit1" unit to remove the "{$DEFINE LOCALDATABASE}" directive and setting the const "sDatabaseName" to the required path e.g.

```
const
  sDatabaseName = 'myserver:/databases/test.fdb';
```

### 12.5.2.2  1D Array Example

A screenshot from this example program is illustrated below.



In this case, the test data table is defined as
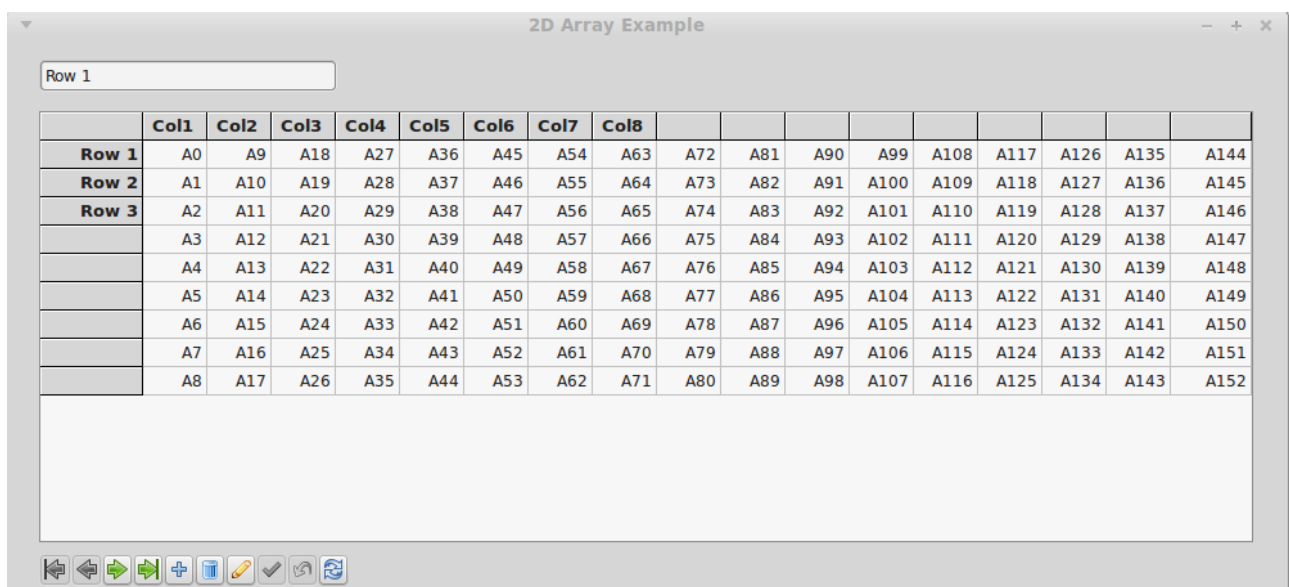
```
  Create Table TestData (
    RowID Integer not null,
    Title VarChar(32) Character Set UTF8,
    MyArray Double Precision [1:12],
    Primary Key(RowID)
  );
```

Each row includes a floating point array with twelve elements. In the example application, the table is displayed and edited using a DBControlGrid. The title field is interpreted as a "Department" and displayed using a TDBEdit control. The array field is interpreted as sales by month and displayed as a one dimensional TIBArrayGrid with column labels. The example allows both the Department Name and monthly sales values to be updated and changes saved. New rows can be inserted and existing rows deleted.

Note: there is an LCL bug (http://bugs.freepascal.org/view.php?id=30892) which will cause the 1D array example to render incorrectly under Windows. That is only the focused row will show the array. The bug report includes an LCL patch to fix this problem. It is believed to be fixed in Lazarus 1.8.0.

### 12.5.3 2D Array Example

A screenshot from this example program is illustrated below.



In this case, the test data table is defined as

```
Create Table TestData (
  RowID Integer not null,
  Title VarChar(32) Character Set UTF8,
  MyArray VarChar(16) [0:16, -1:7] Character Set UTF8,
  Primary Key(RowID)
);
```

Each row includes a two dimensional string array with indices 0..16 and -1 to 7. The grid interprets the first index as a column index and the second as a row index (i.e. x,y Cartesian co-ordinates).

The example program displays a row at a time with a navigation bar providing the means to scroll through the dataset, as well as saving or cancelling changes, inserting and deleting rows.

This example illustrates the use of both column and row labels.

145

# Appendix A. The Legacy IBX Services Components

This appendix describes the legacy IBX Components. These may be found on the "Firebird Legacy Admin" palette. They have been superseded by the re-architectured Services components described in chapter 10. The legacy components are deprecated and should not be used in new applications.

## A.1 Firebird Legacy Admin Component Overview

| | | |
|---|---|---|
|  | `TIBBackupService` | The backup service supports database backup to gbak format archives. Both server side and client side backup file locations are supported. |
|  | `TIBRestoreService` | The restore service supports database restore from gbak format archives. Both server side and client side backup file locations are supported. |
|  | `TIBConfigService` | The configuration service allows database parameters to be modified, including whether the database is online, sync versus async writes, etc. |
|  | `TIBServerProperties` | This service retrieves various server properties including the server version information, server parameters and the current status of database attachments. |
|  | `TIBLogService` | This service supports the retrieval of the server log file contents. |
|  | `TIBStatisticalService` | This service supports the retrieval of per database statistics. |
|  | `TIBSecurityService` | This service supports management of the User Security Database. |
|  | `TIBValidationService` | This service supports the invocation of various database repair actions, including validation and sweep. Limbo Transactions can also be resolved. |
|  | TIBOnlineValidationService | This service was introduced for Firebird 3 and provides for a table level validation of a database. It implements consistency checks that do not require exclusive access to a database. |

## A.2  Common Service Properties

All Firebird Admin components derive from a common ancestor class and are used similarly. A common service editor is available at design time to set the login parameter defaults. The following properties are in common:

| | |
|---|---|
| `Active` | Set to true to attach to the server and establish a connection with it.<br><br>Set to false to terminate an active connection. |
| `LoginPrompt` | Set to true to enable use of the built-in login prompt dialog. |
| `Params` | Holds the login user name and password, as a list of keyword equals string (e.g. user_name=SYSDBA, password=masterkey). Is is recommended that the password is not set at design time. A component editor is available to set each service's parameters. |
| `Protocol` | Determines the connection type (local, TCP,SPX or Named Pipe). Only the first two should be considered for use in current systems. |
| `ServerName` | The (domain) name of the server. |
| `ServerVersionNo` | An integer array [1..4] giving the Firebird Release number (elements 1..3) and the build no. (element 4). |

The public property **ServiceIntf** exposes the IServiceManager interface used to communicate with the server. This interface is available (non-nil) when Active is true. It can be assigned between Firebird Admin components allowing them to share the same connection without having to separately log in for each separate service.

Note: setting active to false disconnects the connection and invalidates the shared interface.

From IBX 2.2 and Firebird 3 onwards, a Params line in the format

 "expected_db=<databasename>"

is supported.

This is used when logging in with the service manager with user credentials that have to be authenticated in an alternative security database. The <databasename> is the name of a database on the server that uses the intended alternative security database.

The purpose of this is (e.g.) to perform a backup/restore for a database using an alternative security database. It is also needed for accessing database statistics or Limbo Transaction recovery.

# A.3 The Backup Service

The backup service supports database backup to gbak format archives. Both server side and client side backup file locations are supported. Before the backup is started, the common properties must be set plus the following:

| | |
|---|---|
| BackupFile | Server Side Backups only: This is a list full pathnames to one or more backup files on the server. When more than one is specified, all but the last should be followed by "=nnn" where nnn is the maximum length in bytes for the file. |
| BackupFileLocation | ServerSide or ClientSide. This determines whether the backup is to a file located on the server side or on the client side of the connection. |
| BlockingFactor | See gbak documentation  for non-zero values (probably obsolescent) |
| DatabaseName | Alias of or full pathname for database on the server. |
| Options | See gbak documentation for interpretation of each option. |
| Verbose | Server Side Backups only: if true then additional text messages are generated. |
| StatisticsRequested | In verbose mode additional statistics are provided with output:<br><br>• Total Time<br>• Time Delta<br>• Page Reads<br>• PageWrites |

## A.3.1 Server Side Backup

The following code illustrates how a server side backup is performed after the above properties have been set:

```
IBBackupService1.Active := true;
IBBackupService1.ServiceStart;
  while not IBBackupService1.Eof do
    writeln(IBBackupService1.GetNextLine);
  Application.ProcessMessages
IBBackupService1.Active := false; {only if you no longer need the connection}
```

Once the service has been started, running the service until completion is a simple loop checking for "EOF", while calling the GetNextLine method. This returns lines of text from the server (most relevant in verbose mode). In this example, they are written to stdout.

### A.3.2 Client Side Backup

The following code illustrates how a client side backup is performed after the above properties have been set:

```
var bakfile: TFileStream;
begin
  bakfile := TFileStream.Create('<path to backup file>',fmCreate);
  try
    IBBackupService1.Active := true;
    IBBackupService1.ServiceStart;
    while not IBBackupService1.Eof do
    begin
      IBBackupService1.WriteNextChunk(bakfile);
      Application.ProcessMessages
    end;
  finally
    bakfile.Free;
  end;
end;
```

The above is very similar to server side case, except that the service user has to provide a `TStream` (in the above `TFileStream`) as the destination of the backup archive. Instead of looking on `GetNextLine`, a client side backup loops on `WriteNextChunk`.

## A.4 The Restore Service

The Restore service supports database restore from gbak format archives. Both server side and client side backup file locations are supported. Before the restore is started, the common properties must be set plus the following:

| BackupFile | Server Side Restores only: This is a list full pathnames to one or more backup files on the server. When more than one is provided, these are read in the same order that the are defined |
|---|---|
| BackupFileLocation | ServerSide or ClientSide. This determines whether the restore is from a file located on the server side or on the client side of the connection. |
| DatabaseName | A list of Aliases of or full pathnames for database on the server. |
| Options | See gbak documentation for interpretation of each option. This must include either CreateNewDB (default) or Replace, but not both. |
| PageBuffers | See gbak documentation |
| PageSize | See gbak documentation |
| Verbose | If true then additional text messages are generated. |
| StatisticsRequested | In verbose mode additional statistics are provided with output:<br><br>• Total Time<br>• Time Delta |

| | • Page Reads<br>• PageWrites |
|---|---|

### A.4.1 Server Side Restores

The following code illustrates how a server side restore is performed after the above properties have been set:

```
    IBRestoreService1.Active := true;
    IBRestoreService1.ServiceStart;
    while not IBRestoreService1.Eof do
    begin
      writeln(IBRestoreService1.GetNextLine);
      Application.ProcessMessages
    end;
```

Once the service has been started, running the service until completion is a simple loop checking for "EOF", while calling the GetNextLine method. This returns lines of text from the server (most relevant in verbose mode). In this example, they are written to stdout.

### A.4.2 Client Side Restores

The following code illustrates how a client side restore is performed after the above properties have been set:

```
var bakfile: TFileStream;
    line: string;
begin
  bakfile := TFileStream.Create('<path to backup file>',fmOpenRead);
  try
    IBRestoreService1.Active := true;
    IBRestoreService1.ServiceStart;
    while not IBRestoreService1.Eof do
    begin
      IBRestoreService1.SendNextChunk(bakfile,line);
      if line <> '' then
        writeln(line);
      Application.ProcessMessages
    end;
  finally
    bakfile.Free;
  end;
end;
```

The above is very similar to server side case, except that the service user has to provide a `TStream` (in the above `TFileStream`) as the source of the backup archive. Instead of looking on `GetNextLine`, a client side backup loops on `SendNextChunk`. This both reads from the stream and may return a line of text when one is received from the server.

## A.5 The Configuration Services

The TIBConfigService must also have its common properties set as described above. Otherwise it comprises a set of methods, each of which performs a specific action.

| ShutdownDatabase | Puts the database into its shutdown state according to the selected options and within the given "wait" time (seconds). Once shutdown, only the SYSDBA user can log into the database. |
|---|---|
| BringDatabaseOnline | Puts the database into its online state (reverse of shutdown). |
| SetSweepInterval | Sets the automatic sweep interval |
| SetDBSqlDialect | Sets the default database SQL dialect (1 or 3) |
| SetPageBuffers | Set the default number of cache buffers to the specified number. |
| ActivateShadow | Activates a database "shadow file" See the Firebird Documentation for more information on database shadow files. |
| SetReserveSpace | Configure the database to fill data pages when inserting new records (true), or reserve 20% of each page for later record deltas (true) |
| SetAsyncMode | Toggles between async writes (true) and sync writes (false). |
| SetReadOnly | Sets read only or read/write mode. |
| SetNoLinger | Set the No Linger Flag on a database. This is a "one-shot" override. Once set, the server will close the database immediately the last attachment is gone, regardless of the LINGER setting in the database. The LINGER setting is retained and works normally the next time. |

## A.6 The Server Properties Service

This service retrieves various server properties including the server version information, server parameters and the current status of database attachments. The information returned is divided up into:

- Server Version Information
- Active Database Information, and
- Configuration Parameters

Each information set has a corresponding method to request the current information, which then sets the values of the linked property. The property may then be read to access the requested information. For example:

```
var i: integer;
begin
  with IBServerProperties1 do
  begin
    Active := true;
    FetchVersionInfo;
    writeln('Server Version = ' + VersionInfo.ServerVersion);
    writeln('Server Implementation = ' + VersionInfo.ServerImplementation);
    writeln('Service Version = ' + IntToStr(VersionInfo.ServiceVersion));
```

```
    FetchDatabaseInfo;
    writeln('No. of attachments = ' + IntToStr(DatabaseInfo.NoOfAttachments));
    writeln('No. of databases = ' + IntToStr(DatabaseInfo.NoOfDatabases));
    for i := 0 to DatabaseInfo.NoOfDatabases - 1 do
      writeln('DB Name = ' + DatabaseInfo.DbName[i]);

    FetchConfigParams;
    writeln('Base Location = ' + ConfigParams.BaseLocation);
    writeln('Lock File Location = ' + ConfigParams.LockFileLocation);
    writeln('Security Database Location = ' + ConfigParams.SecurityDatabaseLocation);
  end;
end;
```

## A.7  The Log Service

This is a simple service that may be used to retrieve the current server log file contents. For example:

```
  with IBLogService1 do
  begin
    Active := true;
    ServiceStart;
    while not Eof do
    begin
      writeln(GetNextLine);
      Application.ProcessMessages;
    end;
  end;
```

## A.8  The Database Statistics Services

This service supports the retrieval of per database statistics as text data. The use of this service is very similar to the Log Service except that:

- The DatabaseName property must be set to alias or full path name on the server of the database for which the statistics are requested.

- The Options property must be set to identify which statistics are requested.

Otherwise, statistics retrieval is the same as for the log file. For example:

```
  with IBStatisticalService1 do
  begin
    DatabaseName := 'myDatabase';
    Options := [HeaderPages];
    Active := true;
    ServiceStart;
    while not Eof do
    begin
      writeln(GetNextLine);
      Application.ProcessMessages;
    end;
  end;
```

The above returns the Header Page statistics. The options available are:

| HeaderPages | Request only the information in the database header page |
|---|---|
| DataPages | Request statistics for user data pages |
| IndexPages | Request statistics for user index pages |
| SystemRelations | Request statistics for system tables and indexes — in addition to user tables and indexes |

## A.9  The Security Service

This service supports management of the User Security Database. It supports:

- The listing of all User Names, and other user identification information
- Adding New Users
- Modifying Existing Users (including changing passwords)
- Deleting Users.
- From Firebird 2.5 onwards, display and management of the user Admin Role.

### A.9.1  Listing all User Names

The `DisplayUsers` method is used to retrieve the list of user names and other user identification information to the `UserInfo` property. This information can then be displayed to the user. For example:

```
var i: integer;
begin
  with IBSecurityService1 do
  begin
    Active := true;
    DisplayUsers;
    for i := 0 to UserInfoCount - 1 do
    with UserInfo[i] do
    begin
      writeln('User ID = ',UserID);
      writeln('Group ID = ',GroupID);
      writeln('User Name = ',UserName);
      writeln('First Name = ', FirstName);
      writeln('Middle Name = ', MiddleName);
      writeln('Last Name = ', LastName);
      write('Admin Role = ');
      if AdminRole then writeln('true') else writeln('false');
    end;
  end;
end;
```

### A.9.2  Adding a User

The `AddUser` method is used to add a user to the Security Database. The `TIBSecurityService` `UserName` and `Password` properties should be set before this method is called to set the user name and password, respectively. The complete set of properties that may be set are:

| UserName | User (or Login) Name |
|---|---|

| Password | The user's password |
|---|---|
| FirstName | The user's first name |
| MiddleName | The user's middle name |
| LastName | The user's last name |
| UserID | The Unix UID |
| GroupID | The Unix GID |
| AdminRole | True if the user is granted the Admin Role, otherwise false. |

The following illustrates the use of the `AddUsers` method:

```
with IBSecurityService1 do
begin
  Active := true;
  UserName := NewUserName;
  Password := NewPassword;
  AddUser;
end;
```

### A.9.3  Updating User Details

The `ModifyUser` method is used to modify a users login details in the Security Database. The `UserName` property acts as the key identifying the user. The remaining properties listed above in 10.11.2 may be set as required to update the corresponding entry in the database. For example:

```
with IBSecurityService1 do
begin
  Active := true;
  UserName := 'SYSDBA';
  FirstName := 'Donald';
  LastName := 'Duck';
  ModifyUser;
end;
```

### A.9.4  Deleting a User

The `DeleteUser` method is used to remove a users login details from the Security Database. The `UserName` property acts as the key identifying the user. For example:

```
with IBSecurityService1 do
begin
  Active := true;
  UserName := 'ALICE';
  DeleteUser;
end;
```

## A.10 The Validation Service

The TIBValidationService supports the invocation of various database repair actions, including validation and sweep. Limbo Transactions can also be resolved. It is effectively two services in one. The first case is used to perform a variety of repair actions. The second is more specific to resolving Limbo Transactions. Exclusive access is required to the database in order to perform these validation activities.

### A.10.1 Database Repair

The following Database Repair services are available and selected by the service's options property:

| Title | Option | Description |
|---|---|---|
| List Limbo Transactions | LimboTransactions | Returns a text list of limbo transactions |
| Check Database | CheckDB | Request read-only validation of the database, without correcting any problem |
| Ignore all checksum errors | IgnoreChecksum | Refines database check |
| Kill Shadow Files | KillShadows | Remove references to unavailable shadow files |
| Mend Database | MendDB | Mark corrupted records as unavailable, so subsequent operations skip them |
| Sweep Database | SweepDB | Request database sweep to mark outdated records as free space; |
| Validate Database | ValidateDB | Locate and release pages that are allocated but unassigned to any data structures |
| Full Database Validation | ValidateFull | Check record and page structures, releasing unassigned record fragments. Use with Validate Database |

For example:

```
with IBValidationService1 do
begin
  DatabaseName := 'MyDatabase';
  Options := [ValidateDB,ValidateFull];
  Active := true;
  ServiceStart;
  while not Eof do
  begin
    writeln(GetNextLine);
```

```
        Application.ProcessMessages;
      end;
    end;
```

## 12.5.4  Resolving Limbo Transactions

There are two steps to the resolution of limbo transactions. The first step retrieves a list of all limbo transactions. The second step commits or rolls back each transaction as required.

The `FetchLimboTransactionInfo` method is used to retrieve list of limbo transactions. After completion, the list may be found in the `LimboTransactionInfo` property. For example:

```
var i: integer;
begin
  with IBValidationService1 do
  begin
    Active := true;
    ServiceStart;
    FetchLimboTransactionInfo;
    for i := 0 to LimboTransactionInfoCount - 1 do
    with LimboTransactionInfo[i] do
    begin
      write('ID = ',ID);
      if MultiDatabase then
        write(', Multi DB')
      else
        write(' ,Single DB');
      write(', Host Site = ', HostSite);
      write(', Remote Site = ', RemoteSite);
      write(', Database Path = ', RemoteDatabasePath);
      write(', State = ', StateToStr(State));
      writeln(', Advise = ', AdviseToStr(Advise));
    end;
  end;
end;
```

where

```
function StateToStr(State: TTransactionState): string;
begin
  case State of
  LimboState:
    Result := 'Limbo';
  CommitState:
    Result := 'Commit';
  RollbackState:
    Result := 'Rollback';
  else
    Result := 'Unknown';
  end;
end;

function AdviseToStr(Advise: TTransactionAdvise): string;
begin
  case Advise of
  CommitAdvise:
    Result := 'Commit';
  RollbackAdvise:
    Result := 'Rollback';
  else
    Result := 'Unknown';
  end;
end;
```

This list identifies each limbo transaction and its current state, it also suggests an action (advises). The user can review the list and set the `TLimboTransactionInfo.Action` property to a desired outcome.

The limbo transactions may then be resolved by setting the `GlobalAction` property and then calling the `FixLimboTransactionErrors` method.

The `GlobalAction` determines how `FixLimboTransactionErrors` processes the limbo transactions and may be set to:

| | |
|---|---|
| `CommitGlobal` | All limbo transactions are resolved by committing the transaction. |
| `RollbackGlobal` | All limbo transactions are resolved by rolling back the transaction. |
| `RecoverTwoPhaseGlobal` | All limbo transactions are resolved by performing a two phase commit of the transaction. |
| `NoGlobalAction` | Limbo transactions are resolved by either committing or rolling back the transaction, as specified by each limbo transaction's Action property. |

For example:

```
with IBValidationService1 do
begin
  GlobalAction := NoGlobalAction
  FixLimboTransactionErrors;
  while not Eof do
  begin
    writeln(GetNextLine);
    Application.ProcessMessages;
  end;
end;
```

## A.11 The Online Validation Service

Online Validation was introduced in Firebird 2.5 that allows some consistency checks to be performed without exclusive access to the database. It may:

- validate some (or all) user tables in a database.

- validate some (or all) indices

Other ODS checks, such as Header/PIP/TIP/Generators pages, are not performed by Online Validation. The TIBOnlineValidationService component is the IBX interface to online validation.

### A.11.1 Properties

| | |
|---|---|
| `IncludeTables` | This is a case sensitive regular expression selecting the tables to include in online validation. For example, to select a list of tables, separate the tables names with the '|' character. |

| | Leave empty to include all tables. |
|---|---|
| `ExcludeTables` | This is a regular expression selecting the tables to exclude from online validation. Leave empty to exclude none. |
| `IncludeIndexes` | This is a regular expression selecting the indexes to include in online validation. Leave empty to include all indexes. |
| `ExcludeIndexes` | This is a regular expression selecting the indexes to exclude from online validation. Leave empty to exclude none. |
| LockTimeout | This is the lock timeout in seconds. default is 10 secs. 0 is no-wait, -1 is infinite wait. |

## A.11.2  Usage

For example:

```
with IBOnlineValidationService1 do
begin
  Active := true;
  StartService;
  while not Eof do
  begin
    writeln(GetNextLine);
    Application.ProcessMessages;
  end;
end;
```