# Developing Firebird Applications in Delphi

This chapter will describe the process of developing applications for Firebird databases with the FireDac™ data access components in the Embarcadero Delphi™ XE5 environment. FireDac™ is a standard set of components for accessing various databases from Delphi XE3 and higher versions.

Starting a Project

Create a new project using File->New->VCL Forms Application - Delphi.

Add a new data module using File->New->Other and selecting Delphi Projects->Delphi Files->Data Module in the wizard. This will be the main data module in our project. It will contain some instances of global access components that must be accessible to all forms that are intended to work with data. TFDConnection is an example of this kind of component.

## TFDConnection component

The TFDConnection component provides connectivity to various types of databases. We will specify an instance of this component in the Connection properties of other FireDac components. The particular type of the database to which the connection will be established depends on the value of the DriverNameproperty. To access Firebird, you need to set this property to FB.

For the connection to know exactly which access library it should work with, place the TFBPhysFBDriverLink component in the main data module. Its VendorLib property enables the path to the client library to be specified precisely. If it is not specified, the component will attempt to establish a connection via libraries registered in the system, for example, in <filename>system32</filename>, which might not be what you want at all.

### Path to the client library

We will place the necessary library in the fbclient folder located in the application folder and use the following code for the OnCreate event of the data module.

xAppPath := ExtractFileDir(Application.ExeName) + PathDelim;  
FDPhysFBDriverLink.VendorLib := xAppPath + 'fbclient' + PathDelim + 'fbclient.dll';

**Important notes about "bitness"!**

If you compile a 32-bit application, you should use the 32-bit fbclient.dll library. For a 64-bit application, it should be the 64-bit library. Along with the file fbclient.dll, it is advisable to place the following libraries in the same folder: msvcp80.dll and msvcr80.dll (for Firebird 2.5) as well as msvcp100.dll and msvcr100.dll (for Firebird 3.0). These libraries are located either in the bin subfolder (Firebird 2.5) or in the root folder of the server (Firebird 3.0).

For the application to show internal firebird errors correctly, it is necessary to copy the file firebird.msg as well.

* For Firebird 2.5 or earlier, the libraries must be one level up from the folder with the client library, i.e. in the application folder for our purposes.
* For Firebird 3, they must be in the same folder as the client library, i.e. in the fbclient folder.

### Developing for Embedded Applications

If you need your application to run without the installed Firebird server, i.e. in the Embedded mode, you should replace fbclient.dll with fbembed.dll for Firebird 2.5, making sure that the width of the CPU register (64-bit or 32-bit) matches the application. If necessary, the name of the library can be placed in the configuration file of your application. It is not necessary to change anything for Firebird 3.0, in which the working mode depends on the connection string and the value of the Providers parameter in the file firebird.conf/databases.conf.

**Tip**

Even if your application is intended to work with Firebird in the Embedded mode, it is advisable to attach to the full server during development. The reason is that embedded Firebird runs in the same address space as the application and any application connecting to a database in embedded mode must be able to obtain exclusive access to that database. Once that connection succeeds, no other embedded connections are possible. When you are connected to your database in the Delphi IDE, the established connection is in Delphi's application space, thus preventing your application from being run successfully from the IDE.

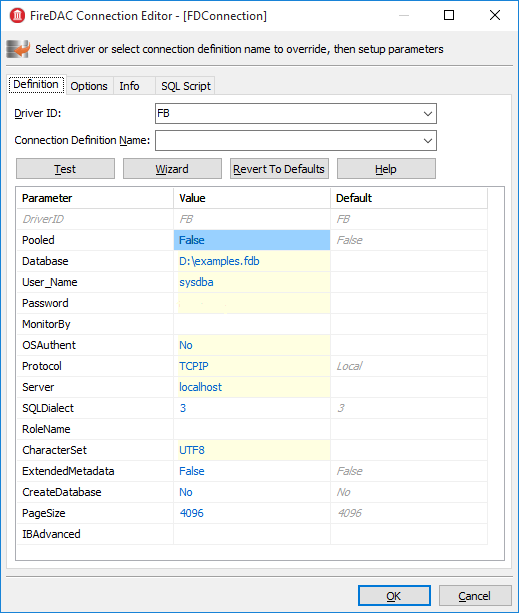
(Note, Firebird 3 embedded still requires exclusive access if the installed full server is in Super (Superserver) mode.)

### Connection parameters

The Params property of the TFDConnection component contains the database connection parameters (username, password, connection character set, etc.). If you invoke the TFDConnection property editor by double-clicking on the component, you will see that those properties have been filled automatically. The property set depends on the database type.

Table 1. TFDConnection component main properties

|  |  |
| --- | --- |
| **Parameter** | **Purpose** |
| Pooled | Whether a connection pool is used. |
| Database. | The path to the database or its alias as defined in the aliases.conf configuration file (or in databases.conf) of the Firebird server. |
| User\_Name | Firebird user name. Not used if OSAuthent is True. |
| Password. | Firebird password. Not used if OSAuthent is True. |
| OSAuthent | Whether operating system authentication is used. |
| Protocol | Connection protocol. Possible values:   * Local – local protocol; * NetBEUI – named pipes, WNET; * SPX –This property is for Novell's IPX/SPX protocol, which has never been supported in Firebird * TCPIP – TCP/IP. |
| Server | Server name or its IP address. If the server is run on a non-standard port, you also need to append the port number after a slash, for instance, localhost/3051. |
| SQLDialect | SQL Dialect. It must match that of the database. |
| RoleName | Role name, if required. |
| CharacterSet | Connection character set name. |



Additional properties:

**Connected** – used to manage the database connection or check the connection status. This property must be set to True in order for the wizards of other FireDac components to work. If your application needs to request authentication data, it is important to remember to reset this property to False before compiling your application.

**LoginPrompt** – whether to request the username and password during a connection attempt.

**Transaction** – the TFDTransaction component that will be used as default to conduct various TFDConnection transactions. If this property is not explicitly specified, TFDConnection will create its own TFDTransaction instance. Its parameters can be configured in the TxOptions property.

**UpdateTransaction** – the TFDTransaction component that is to be used as default for the UpdateTransaction property of TFDQuery components , unless explicitly specified for the dataset. If this property is not specified explicitly, the value from the Transaction property of the connection will be used, unless it is explicitly specified for the dataset.

### Connection Parameters in a Configuration File

Since the connection parameters, except for the username and password and possibly the role, are usually common to all instances the application, we will read them from the configuration file.

xIniFile := TIniFile.Create(xAppPath + 'config.ini');  
**try**  
 xIniFile.ReadSectionValues('connection', FDConnection.Params);  
**finally**  
 xIniFile.Free;  
**end**;

### A Typical Configuration File

Typically, the config.ini file contains the following lines:

**[connection]**  
**DriverID**=FB  
**Protocol**=TCPIP  
**Server**=localhost/3051  
**Database**=examples  
**OSAuthent**=No  
**RoleName**=  
**CharacterSet**=UTF8

You can get the contents of the connection section by copying the contents of the Params property of the TFDConnection component after the wizard finishes its work.

**Note**

Actually, the common settings are usually located in %AppData%\*Manufacturer*\*AppName* and are saved to that location by the application installation software. However, it is convenient for the configuration file to be stored somewhere closer during the development, for instance, in the application folder.

Note that if your application is installed into the Program Files folder and the configuration file is located there as well, it is likely that the file will be virtualized in Program Data and issues could arise with modifying it and reading the new settings subsequently.

### Connecting to the database

To connect to the database, it is necessary to change the Connected property of the TFDConnection component to True or call the Open method. You can use the Open method to pass the username and password as parameters.

#### A Little Modification

We will replace the standard database connection dialog box in our application and allow users to make three mistakes while entering the authentication information. After three failures, the application will be closed.

To implement it, we will write the following code in the OnCreate event handler of the main data module.

*// After three unsuccessful login attempts, we close the application.*

xLoginCount := 0;  
xLoginPromptDlg := TLoginPromptForm.Create(Self);  
**while** (xLoginCount < MAX\_LOGIN\_COUNT) **and**  
 (**not** FDConnection.Connected) **do**  
**begin**  
 **try**  
 **if** xLoginPromptDlg.ShowModal = mrOK **then**  
 FDConnection.Open(  
 xLoginPromptDlg.UserName, xLoginPromptDlg.Password)  
 **else**  
 xLoginCount := MAX\_LOGIN\_COUNT;  
 **except**  
 **on** E: Exception **do**  
 **begin**  
 Inc(xLoginCount);  
 Application.ShowException(E);  
 **end**  
 **end**;  
**end**;  
xLoginPromptDlg.Free;  
**if not** FDConnection.Connected **then**  
 Halt;

## Working with Transactions

The Firebird client allows any operations to be made only in the context of a transaction so, if you manage to access data without explicitly calling TFDTransaction.StartTransaction, it means that it was called automatically somewhere deep in FireDac. It is highly recommended to avoid this practice. For applications to work correctly with databases, it is advisable to manage transactions manually, which means starting and committing them or rolling them back with explicit calls.

TFDTransaction has three methods for managing a transaction explicitly: StartTransaction, Commit and Rollback. The following table summarizes the properties available to configure this component.

Table 2. TFDTransaction component main properties

|  |  |
| --- | --- |
| **Parameter** | **Purpose** |
| Connection | Reference to the FDConnection component. |
| Options.AutoCommit | Controls the automatic start and end of a transaction, emulating Firebird's own transaction management. The default value is True. See note (1) below for more details about behaviour if this option is True. |
| Options.AutoStart | Controls the automatic start of a transaction. The default value is True. |
| Options.AutoStop | Controls the automatic end of a transaction. The default value is True. |
| Options.DisconnectAction | The action that will be performed when the connection is closed while the transaction is active. The default value is xdCommit – the transaction will be committed. See note (2) below for details of the other options. |
| Options.EnableNested | Controls nested transactions. The default value is True. Firebird does not support nested transactions as such but FireDac can emulate them using savepoints. For more details, see note(3) below. |
| Options.Isolation | Specifies the transaction isolation level. It is the most important transaction property. The default value is xiReadCommitted. The other values that Firebird supports are xiSnapshot and xiUnspecified; also xiSerializable, to some degree. For more details about the available isolation levels, see note (4) below. |
| Options.Params | Firebird-specific transaction attributes that can be applied to refine the transaction parameters, overriding attributes applied by the standard implementation of the selected isolation level. For the attributes that can be set and the "legal" combinations, see note (5) below. |
| Options.ReadOnly | Indicates whether it is a read-only transaction. The default value is False. Setting it to True disables any write activity. Long-running read-only transactions in READ COMMITTED isolation are recommended for activities that do not change anything in the database because they use fewer resources and do not interfere with garbage collection. |

Note 1: AutoCommit=True

1. If the value of AutoCommit is set to True, FireDAC behaves as follows:

1. Starts a transaction (if required) before each SQL command and ends the transaction after the SQL command completes execution
2. If the command is successfully executed, the transaction will be ended by COMMIT. Otherwise, it will be ended by ROLLBACK.
3. If the application calls the StartTransaction method, automatic transaction management will be disabled until that transaction is ended by Commit or Rollback.

Note 2: DisconnectAction

The following values are possible:

* xdNone – nothing will be done. The DBMS will perform its default action.
* xdCommit – the transaction will be committed
* xdRollback – the transaction will be rolled back

Note that, in some other data access components, the default value for the DisconnectAction property is xdRollback and will need to be set manually to match the FDTransaction setting.

Note 3: EnableNested

If StartTransaction is called from within an active transaction, FireDac will emulate a nested transaction by creating a savepoint. Unless you are very confident in the effect of enabling nested transactions, set EnableNested to False. With this setting, calling StartTransaction inside the transaction will raise an exception.

Note 4: Isolation

FireBird has three isolation levels: READ COMMITTED, SNAPSHOT ("concurrency") and SNAPSHOT TABLE STABILITY ("consistency", rarely used). FireDac supports some but not all configurations for READ COMMITTED and SNAPSHOT. It uses the third level partially to emulate the SERIALIZABLE isolation that Firebird does not support.

* xiReadCommitted – the READ COMMITTED isolation level. FireDac starts ReadCommitted transactions in Firebird with the following parameters: read/write, rec\_version, nowait
* xiSnapshot – the SNAPSHOT (concurrency) isolation level. FireDac starts Snapshot transactions in Firebird with the following parameters: read/write, wait
* xiUnspecified – Firebird's default isolation level (SNAPSHOT) with the following parameters: read/write, wait
* xiSerializable – the SERIALIZABLE isolation level. Firebird does not support serializable isolation, but FireDac emulates it by starting a SNAPSHOT TABLE STABILITY ("consistency") transaction with the following parameters: read/write, wait.

Other parameters, not supported by Firebird at all, are:

* xiDirtyRead – if this is selected (not a good idea!) READ COMMITTED will be used instead
* xiRepeatableRead – if this is selected, SNAPSHOT will be used instead

Note 5: Firebird-specific Transaction Attributes

Attributes that can be customised in Options.Params are:

* read write, the default read mode for all of the options.isolation selections--see note (4) above. Set write off if you want read-only mode. Alternatively, you can set Options.ReadOnly to True to achieve the same thing. There is no such thing as a "write-only" transaction.
* read\_committed, concurrency and consistency are isolation levels.
* wait and nowait are conflict resolution settings, determining whether the transaction is tto wait for a conflict to resolve or return conflict errors immediately.
* rec\_version and no rec\_version provide an option that is applicable only to READ COMMITTED transactions. The default rec\_version lets this transaction read the latest committed version of a record and overwrite it if the transaction ID of the latest committed version is newer (higher) than the ID of this transaction. The no rec\_version setting will block this transaction from reading the latest committed version if an update is pending from any other transaction.

### Multiple Transactions

Unlike many other DBMSs, Firebird allows as many TFDTransaction objects as you need to associate with the same connection. In our application, we will use one common read transaction for all primary and secondary modules and one read/write transaction for each dataset.

We do not want to rely on starting and ending transactions automatically: we want to have full control. That is why Options.AutoCommit = False, Options.AutoStart = False and Options.AutoStop = False in all of our transactions.

## Datasets

The components TFDQuery, TFDTable, TFDStoredProc and TFDCommand are the components for working with data in FireDac. TFDCommand does not deliver a dataset and, when TFDStoredProc is used with an executable stored procedure, rather than a selectable one, it does not deliver a dataset, either.TFDQuery, TFDTable and TFDStoredProc are inherited from TFDRdbmsDataSet. Apart from datasets for working with the database directly, FireDac also has the TFDMemTable component for working with in-memory datasets. It is functionally equivalent to TClientDataSet.

The main component for working with datasets, TFDQuery, can be used for practically any purpose. The TFDTable and TFDStoredProc components are justvariants, expanded or reduced to meet differences in functionality. No more will be said about them and we will not be using them in our application. If you wish, you can learn about them in the FireDac documentation.

The purpose of a dataset component is to buffer records retrieved by the SELECT statement, commonly for displaying in a grid and providing for the current record in the buffer (grid) to be editable. Unlike the IBX.IBDataSet component, TFDQuery does not have the properties RefreshSQL, InsertSQL, UpdateSQL and DeleteSQL. Instead, a separate TFDUpdateSQL object specifies the statement for a dataset modification and the dataset component carries a reference to that component in its UpdateObject property.

**Note**

Sometimes it is possible to make an FDQuery object editable without referring, through the UpdateObject property, to an FDUpdateSQL object that specifies queries for insert, update and delete. The property UpdateOptions.RequestLive can be set to True for sets that are naturally updatable and the object will generate the modification queries for you. However, because this approach puts strict limitations on the SELECT query, it is not always useful to rely on it.

Table 3. TFDQuery component main properties

|  |  |
| --- | --- |
| **Parameter** | **Purpose** |
| Connection | Reference to the FDConnection object. |
| MasterSource | If the dataset is to be used as detail to a master dataset, this property refers to the data source (TDataSource) of the master set |
| Transaction | If specified, refers to the transaction within which the query will be executed. If not specified, the default transaction for the connection will be used. |
| UpdateObject | Reference to the FDUpdateSQL object providing for the dataset to be editable when the SELECT query does not meet the requirements for automatic generation of modification queries with UpdateOptions.RequestLive=True. |
| UpdateTransaction | The transaction within which modification queries will be executed. If the property is not specified the transaction from the Transaction property of the connection will be used. |
| UpdateOptions.CheckRequired | If set to True (the default) FireDac controls the Required property of the corresponding fields NOT NULL fields. By default, it is set to True.  If you keep it True and a field with the property Required=True has no value assigned to it, an exception will be raised when the Post method is called. This might not be what you want if a value is going to be assigned to this field later in BEFORE triggers. |
| UpdateOptions.EnableDelete | Specifies whether a record can be deleted from the dataset. If EnableDelete=False, an exception will be raised when the Delete method is called. |
| UpdateOptions.EnableInsert | Specifies whether a record can be inserted into the dataset. If EnableInsert=False, an exception will be raised when the Insert/Append method is called. |
| UpdateOptions.EnableUpdate | Specifies whether a record can be edited in the dataset. If EnableUpdate=False, an exception will be raised when the Edit method is called. |
| UpdateOptions.FetchGeneratorPoint | Controls the moment when the next value is fetched from the generator specified in the UpdateOptions.GeneratorName property or in the GeneratorName property of the auto-incremental field AutoGenerateValue=arAutoInc. The default is gpDeferred, causing the next value to be fetched from the generator before a new record is posted in the database, i.e., during Post or ApplyUpdates. For the full set of possible values, see note (1) below. |
| UpdateOptions.GeneratorName | The name of the generator from which the next value for an auto-incremental field is to be fetched. |
| UpdateOptions.ReadOnly | Specifies whether it is a read-only dataset. The default value is False. If the value of this property is set to True, the values of the EnableDelete, EnableInsert and EnableUpdate properties will be automatically set to False. |
| UpdateOptions.RequestLive | Setting RequestLive to True makes a query editable, if possible. Queries for insert, update and delete will be generated automatically. This setting imposes strict limitations on the SELECT query. It is supported for backward compatibility with the ancient BDE and is not recommended. |
| UpdateOptions.UpdateMode | * Controls how to check whether a record has been modified. This property allows control over possible overwriting of updates in cases where one user is taking a long time to edit a record while another user has been editing the same record simultaneously and completes the update earlier. The default is upWhereKeyOnly. For information about the available modes, see note (2) below. |
| CachedUpdates | Specifies whether the dataset cache defers changes in the dataset buffer. If this property is set to True, any changes (Insert/Post, Update/Post, Delete) are saved to a special log and the application must apply them explicitly bycalling the ApplyUpdates method. All changes will be made within a small period of time and within one short transaction. The default value of this property is False. |
| SQL | Contains the text of the SQL query. If this property is a SELECT statement, execute it by calling the Open method. Use Execute or ExecSQL for executing a statement that does not return a dataset. |

Note 1: UpdateOptions.FetchGeneratorPoint

The property UpdateOptions.FetchGeneratorPoint can take the following values:

* gpNone--no value is fetched from the generator
* gpImmediate--the next value is fetched from the generator right after the Insert/Append method is called
* gpDeferredthe next value is fetched during Post or ApplyUpdates

Note 2: UpdateOptions.UpdateMode

The user in a lengthy editing session could be unaware that a record has been updated one or more times during his editing session, perhaps causing his own changes to overwrite someone else's updates. The UpdateOptions.UpdateMode property allows a choice of behaviours to lessen or avoid this risk:

* upWhereAll--check whether a record exists by its primary key + check all columns for old values, e.g.,

update table set ...

where pkfield = :old\_ pkfield and

client\_name = :old\_client\_name and

info = :old\_info ...

With upWhereAll set, the update query will change content in a record only if the record has not been edited by anyone else since our transaction started. It is especially important if there are dependencies between values in columns, such as minimum and maximum wages, etc.

* upWhereChanged set--check whether a record exists by its primary key + check for old values only in the columns being edited.

update table set ...

where pkfield = :old\_pkfield and

client\_name = :old\_client

* upWhereKeyOnly--check whether a record exists by its primary key. This check corresponds to the automatically generated UpdateSQL query.

To avoid (or handle) update conflicts in a multi-user environment, typically you need to add WHERE conditions manually. You would need a similar tactic, of course, to implement a process that emulates upWhereChanged, removing the unused column modifications from the update table set, leaving in the update list only the columns that are actually modified. The update query could otherwise overwrite someone else's updates of this record.

Obviously, the UpdateSQL needs to be created dynamically.

If you want to specify the settings for detecting update conflicts individually for each field, you can use the ProviderFlags property for each field.

### TFDUpdateSQL component

The TFDUpdateSQL component enables you to refine or redefine the SQL command that Delphi generates automatically for updating a dataset. It can be used to update an FDQuery object, an FDTable object or data underlying an FDStoredProc object.

Using TFDUpdateSQL is optional for TFDQuery and TFDTable because these components can generate statements automatically, that can sometimes be used for posting updates from a dataset to the database.

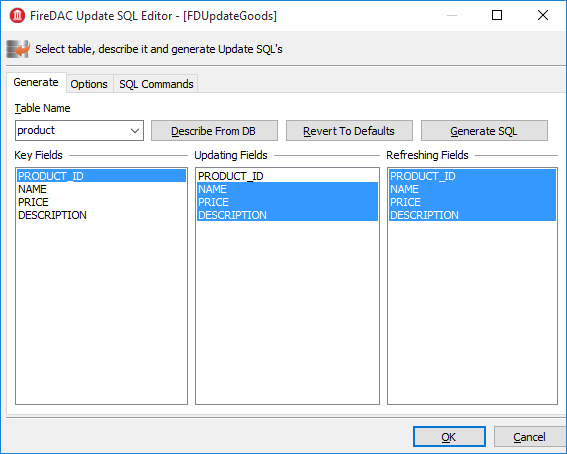
|  |
| --- |
| Note  Automatic generation of modifying queries works only in the simplest cases, when a query accesses one table. In addition, there are other limitations, such as the inability to use aggregate functions, groupings, window functions, etc. |

For updating a dataset that is delivered into an FDStoredProc object, use of the TUpdateSQL is not optional. The developer must figure out a statement that will result in the desired updates. If only one table is updated, a direct DML statement might be sufficient. Where multiple tables are affected, an executable stored procedure will be unavoidable.

We recommend that you always use it, even in the simplest cases, to give yourself full control over the queries that are requested from your application. To specify SQL commands at design time, double-click on the component to open the property editor.

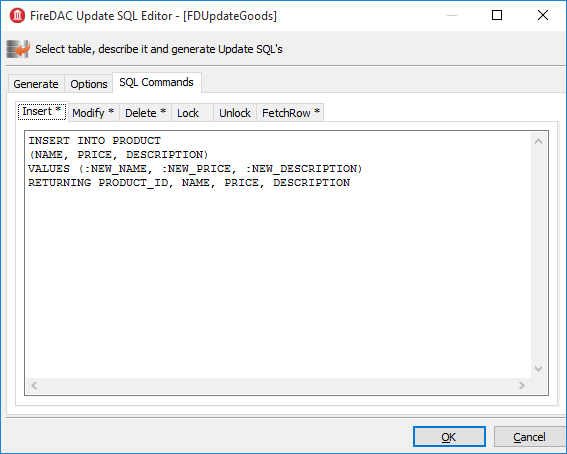
**Note**

For multiple design-time editors to run, FireDac needs an active connection to the database (TFDConnection.Connected = True) and a transaction in the autostart mode (TFDTransaction.Options.AutoStart = True). But settings like that will interfere with the work of the application. For instance, the user is supposed to log into the program using his username, but TFDConnection connects to the database as SYSDBA. So we recommend that you check the TFDConnection.Connected property and reset it each time you use design-time editors. Besides, you will have to enable and disable autostart for a read-only transaction.



You can use the Generate tab to make writing Insert/Update/Delete/Refresh queries easier for yourself. Select the table to be updated, its key fields, the fields to be updated and the fields that will be reread after the update and click the Generate SQL button to have Delphi generate the queries automatically.

You will be switched to the SQL Commands tab where you can correct each query.



**Note**

Since product\_id is not included in Updating Fields, it is absent from the generated Insert query. It is assumed that this column is filled automatically by a generator call in a BEFORE INSERT trigger or, or, from Firebird 3.0 forward, it could be an IDENTITY column. When a value is fetched from the generator for this column at the server side, it is recommended to add the PRODUCT\_ID column manually to the RETURNING clause of the INSERT statement.

#### The Options Tab

The Options tab contains some properties that can affect the process of query generation. These properties are not related to the TFDUpdateSQL component itself. Rather, for the sake of convenience, they are references to the UpdateOptions properties of the dataset that has the current TFDUpdateSQL specified in the UpdateObject property

Table 4. TFDUpdateSQL component main properties

|  |  |
| --- | --- |
| **Parameter** | **Purpose** |
| Connection | Reference to the FDConnection component. |
| DeleteSQL | The SQL query for deleting a record. |
| FetchRowSQL | The SQL query for returning a current record after it has been updated or inserted--"RefreshSQL". |
| InsertSQL | The SQL query for inserting a record. |
| LockSQL | The SQL query for locking a current record. (FOR UPDATE WITH LOCK). |
| ModifySQL | The SQL query for modifying a record. |
| UnlockSQL | The SQL query for unlocking a current record. It is not used in Firebird. |

Notice that, because the TFDUpdateSQL component does not execute modification queries directly, it has no Transaction property. It acts as a replacement for queries automatically generated in the parent TFDRdbmsDataSet.

### TFDCommand component

The TFDCommand component is used to execute SQL queries. It is not descended from TDataSet so it is valid to use only for executing SQL queries that do not return datasets.

Table 5. TFDCommand component main properties

|  |  |
| --- | --- |
| **Parameter** | **Purpose** |
| Connection | Reference to the FDConnection component. |
| Transaction | The transaction within which the SQL command will be executed |
| CommandKind | Type of command. The types are described in the section below. |
| CommandText | SQL query text |

#### Types of Command

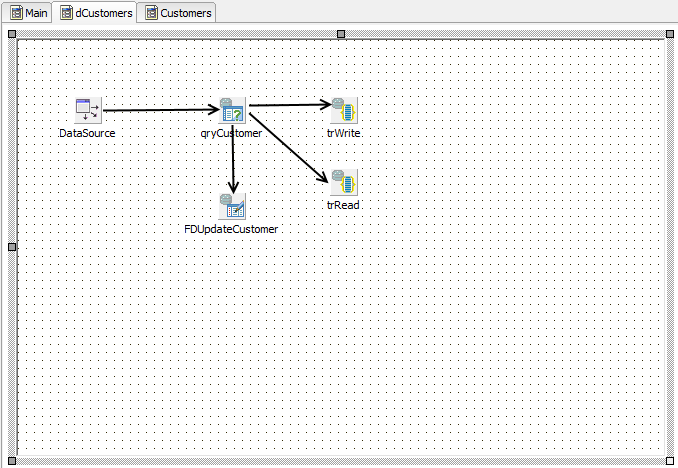
Usually, the command type is determined automatically from the text of the SQL statement. The following values are available for the property TFDCommand.CommandKind to cater for cases where the internal parser might be unable to make correct, unambiguous assumptions based on the statement text alone:

* skUnknown--unknown. Tells the internal parser to determine the command type automatically from its analysis of the text of the command
* skStartTransaction--a command for starting a transaction
* skCommit--a command for ending and committing a transaction
* skRollback--a command for ending and rolling back a transaction
* skCreate--a CREATE … command for creating a new metadata object
* skAlter--an ALTER … command for altering a metadata object
* skDrop--a DROP … command for deleting a metadata object
* skSelect--a SELECT command for retrieving data
* skSelectForLock--a SELECT … WITH LOCK command for locking the selected rows
* skInsert--an INSERT … command for inserting a new record
* skUpdate--an UPDATE … command for modifying records
* skDelete--a DELETE … command for deleting records
* skMerge--a MERGE INTO … command
* skExecute--an EXECUTE PROCEDURE or or EXECUTE BLOCK command
* skStoredProc--a stored procedure call
* skStoredProcNoCrs--a call to a stored procedure that does not return a cursor
* skStoredProcWithCrs&mdash;a call to a stored procedure that returns a cursor

## Creating the Primary Modules

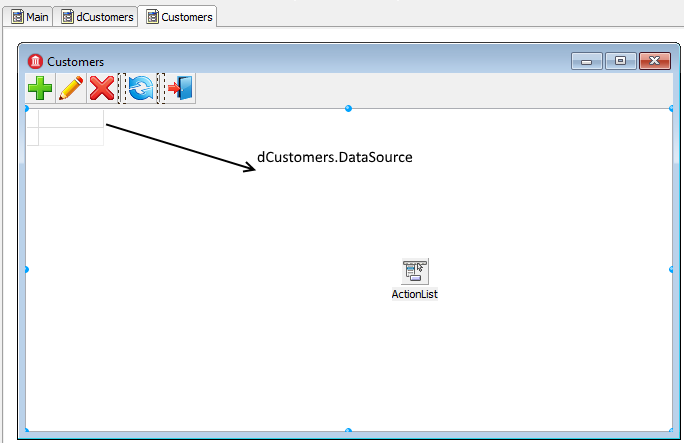
We will create two primary modules in our application: a product module and a customer module. Each primary dataset is displayed on a form by means of a TDBGrid grid and a toolbar with buttons. The business logic of working with the dataset will be located in a separate DataModule that contains a TDataSource data source, a TFDQuery dataset, and two TFDTransaction transactions, one read-only and one read/write.

As our model for creating datasets, we will create the Customer dataset on the dCustomers datamodule:



On tabbing to the Customers form, this is the initial view. The DataSource component

is not visible on the form because it is located in the dCustomers datamodule.



We have placed the TFDQuery component in the dCustomers datamodule and named it qryCustomers. This dataset will be referred to in the DataSet property of the DataSource data source in dCustomers. We specify the read-only transaction trRead in the Transaction property, the trWritetransaction in the UpdateTransaction property and, for the Connection property, the connection located in the main data module. We populate the SQL property with the following query:

**SELECT**  
 customer\_id,  
 **name**,  
 address,  
 zipcode,  
 phone  
**FROM**  
 customer  
**ORDER BY name**

### The Read-only Transaction

The read-only transaction trRead is started when the dataset form is displayed (the OnActivate event) and is ended when the form is closed. READ COMMITTED isolation level (Options.Isolation = xiReadCommitted) is usually used to show data in grids because it allows the transaction to see changes committed in the database by other users by just repeating queries (rereading data) without the transaction being restarted.

Since this transaction is used only to read data, we set the Options.ReadOnly property to True. Thus, our transaction will have the following parameters: read read\_committed rec\_version.

|  |
| --- |
| Why?  A transaction with exactly these parameters can remain open in Firebird as long as necessary (days, weeks, months) without locking other transactions or affecting the accumulation of garbage in the database because, with these parameters, a transaction is started on the server as committed. |

We set the property Options.DisconnectAction to xdCommit, which perfectly fits a read-only transaction. Finally, the read transaction will have the following properties:

Options.AutoStart = False  
Options.AutoCommit = False  
Options.AutoStop = False  
Options.DisconnectAction = xdCommit  
Options.Isolations = xiReadCommitted  
Options.ReadOnly = True

**Important**

Although we do not discuss reporting in this manual, be aware that you should not use such a transaction for reports, especially if they use several queries in sequence. A transaction with READ COMMITTED isolation will see all new committed changes when rereading data.

The recommended configuration for reports is a short read-only transaction with SNAPSHOT isolation (Options.Isolation = xiSnapshot and Options.ReadOnly= True).

### The Read/Write Transaction

The trWrite read/write transaction trWrite that we use for our FDUpdateSQL object must be as short as possible to prevent the oldest active transaction from getting "stuck" and inhibiting garbage collection. High levels of uncollected garbage will lead to lower performance. Since the read/write transaction is very short, we can use the SNAPSHOT isolation level. The default value of the Options.DisconnectAction property, xdCommit, is not appropriate for write transactions, so it should be set to xdRollback. We will not rely on starting and ending transactions automatically. Instead, we will start and end a transaction explicitly. Thus, our transaction will have the following properties:

Options.AutoStart = False  
Options.AutoCommit = False  
Options.AutoStop = False  
Options.DisconnectAction = xdRollback  
Options.Isolations = xiSnapshot  
Options.ReadOnly = False

### SNAPSHOT vs READ COMMITTED Isolation

It is not absolutely necessary to specify SNAPSHOT isolation for simple INSERT/UPDATE/DELETE operations. However, if a table has complex triggers or a stored procedure is executed instead of a simple INSERT/UPDATE/DELETE query, it is advisable to use SNAPSHOT.

The reason is that the READ COMMITTED isolation level does not ensure the read consistency of the statement within one transaction, since the SELECT statement in this isolation can return data that were committed to the database after the transaction began. In principle, the SNAPSHOT isolation level is recommended for short-running transactions.

### Configuring the Customer Module for Editing

In this section, we will configure some properties in the qryCustomer and FDUpdateCustomer objects to make the Customer dataset editable.

#### The TFDUpdateSQL Settings

To make the dataset editable, the InsertSQL, ModifySQL, DeleteSQL and FetchRowSQL properties should be specified in the FDUpdateSQL object that is linked to the dataset. The wizard can generate these statements but it may be necessary to correct some things afterwards. For instance, you can add a RETURNING clause, remove some columns from the update list or cancel an automatically generated stored procedure call entirely.

*InsertSQL:*

**INSERT INTO** customer (

customer\_id,  
 **name**,  
 address,  
 zipcode,  
 phone)  
**VALUES** (:new\_customer\_id,  
 :new\_name,  
 :new\_address,  
 :new\_zipcode,  
 :new\_phone)

*ModifySQL:*

**UPDATE** customer  
**SET name** = :new\_name,  
 address = :new\_address,  
 zipcode = :new\_zipcode,  
 phone = :new\_phone  
**WHERE** (customer\_id = :old\_customer\_id)

*DeleteSQL:*

**DELETE FROM** customer  
**WHERE** (customer\_id = :old\_customer\_id)

*FetchRowSQL:*

**SELECT**  
 customer\_id,  
 **name**,  
 address,  
 zipcode,  
 phone  
**FROM**  
 customer  
**WHERE** customer\_id = :old\_customer\_id

#### Getting a Generator Value

In this project, we will get the value from the generator before making an insert into the table. To enable that, specify the following values for the properties of the TFDQuery component: UpdateOptions.GeneratorName = GEN\_CUSTOMER\_ID and UpdateOptions.AutoIncFields = CUSTOMER\_ID.

|  |
| --- |
| Note  This method works only for autoinc fields that are populated by explicit generators (sequences). It is not applicable to the IDENTITY type of autoinc key introduced in Firebird 3.0. |

Another way to get the value from the generator is to return it after the INSERT is executed by means of a RETURNING clause. This method, which works for IDENTITY fields as well, will be shown later.

### Implementing the Customer Module

Modal forms are often used to add a new record or to edit an existing one. Once the modal form is closed by the mrOK result, the changes are posted to the database. Database-aware visual components are usually used to create this kind of form. These components enable you to display the values of some fields from the current record and immediately accept the user's changes in the corresponding fields if the dataset is in Insert/Edit mode, i.e. before Post.

The only way to switch the dataset to Insert/Edit mode is by starting a write transaction. So, if somebody opens a form for adding a new record and leaves for a lunch break, we will have an active transaction hanging until the user comes back from lunch and closes the form. This uncommitted edit can inhibit garbage collection, which will reduce performance. There are two ways to solve this problem:

1. Use the CachedUpdates mode, which enables the transaction to be active just for a very short period (to be exact, only for the time it takes for the changes to be applied to the database).
2. Give up using visual components that are data-aware. This approach requires some additional effort from you to activate the data source and pass user input to it.

We will show how both methods are used. The first method is much more convenient to use. Let's examine the code for editing a customer record:

**procedure** TCustomerForm.actEditRecordExecute(Sender: TObject);  
**var**  
 xEditorForm: TEditCustomerForm;  
**begin**  
 xEditorForm := TEditCustomerForm.Create(Self);  
 **try**  
 xEditorForm.OnClose := CustomerEditorClose;  
 xEditorForm.DataSource := Customers.DataSource;

xEditorForm.Caption := 'Edit customer';  
 Customers.Edit;  
 xEditorForm.ShowModal;  
 **finally**  
 xEditorForm.Free;  
 **end**;  
**end**;

The Customers property is initiated in the OnCreate event:

**procedure** TCustomerForm.FormCreate(Sender: TObject);  
**begin**  
 FCustomers := TDMCustomers.Create(Self);  
 DBGrid.DataSource := Customers.DataSource;  
**end**;

We set the CachedUpdates mode for the dataset in the Edit method of the dCustomers module before switching it to the edit mode:

**procedure** TdmCustomers.Edit;  
**begin**  
 qryCustomer.CachedUpdates := True;  
 qryCustomer.Edit;  
**end**;

The logic of handling the process of editing and adding a record is implemented in the OnClose event handler for the modal edit form:

**procedure** TCustomerForm.CustomerEditorClose(Sender: TObject;  
 **var** Action: TCloseAction);  
**begin**  
 **if** TEditCustomerForm(Sender).ModalResult <> mrOK **then**  
 **begin**  
 Customers.Cancel;  
 Action := caFree;  
 Exit;  
 **end**;  
 **try**  
 Customers.Post;  
 Customers.Save;  
 Action := caFree;  
 **except**  
 **on** E: Exception **do**  
 **begin**  
 Application.ShowException(E);  
 *// It does not close the window give the user correct the error*  
 Action := caNone;  
 **end**;  
 **end**;

**end**;

To understand the internal processes, we can study the code for the Cancel, Post and Save methods of the dCustomer data module.

**procedure** TdmCustomers.Cancel;  
**begin**  
 qryCustomer.Cancel;  
 qryCustomer.CancelUpdates;  
 qryCustomer.CachedUpdates := False;  
**end**;

**procedure** TdmCustomers.Post;  
**begin**  
 qryCustomer.Post;  
**end**;

**procedure** TdmCustomers.Save;  
**begin**  
 *// We do everything in a short transaction*  
 *// In CachedUpdates mode an error does not interrupt the running code.*  
 *// The ApplyUpdates method returns the number of errors.*  
 *// The error can be obtained from the property RowError*  
 **try**  
 trWrite.StartTransaction;  
 **if** (qryCustomer.ApplyUpdates = 0) **then**  
 **begin**  
 qryCustomer.CommitUpdates;  
 trWrite.Commit;  
 **end**  
 **else**  
 **raise** Exception.Create(qryCustomer.RowError.**Message**);  
 qryCustomer.CachedUpdates := False;  
 **except**  
 **on** E: Exception **do**  
 **begin**  
 **if** trWrite.Active **then**  
 trWrite.Rollback;  
 **raise**;  
 **end**;  
 **end**;  
**end**;

Observe that the write transaction is not started at all until the OK button is clicked. Thus, the write transaction is active only while the data are being transferred from the dataset buffer to the database. Since we access not more than one record in the buffer, the transaction will be active for a very short time, which is exactly what we want.

### Using a RETURNING Clause to Acquire an Autoinc Value

Creating the product module is similar to creating the customer one. We will use it to demonstrate the method of getting an auto-incremented value by means of a RETURNING clause.

The main query:

**SELECT**  
 product\_id,  
 **name**,  
 price,  
 description  
**FROM** product  
**ORDER BY name**

The TFDUpdateSQL.InsertSQL property will contain the following statement:

**INSERT INTO** PRODUCT (**NAME**, PRICE, DESCRIPTION)  
**VALUES** (:NEW\_NAME, :NEW\_PRICE, :NEW\_DESCRIPTION)  
**RETURNING** PRODUCT\_ID

The RETURNING clause in this statement will return the value of the PRODUCT\_ID field after it has been populated by the BEFORE INSERT trigger. The client side in this case has no need to know the name of the generator, since it all happens on the server. Leave the UpdateOptions.GeneratorName property as nil.

Acquiring the autoinc value by this method also requires filling a couple of properties for the PRODUCT\_ID field because the value is being entered indirectly: Required = False and ReadOnly = True. Everything else is set up similarly to the way it was done for the Customer module.

## Creating a secondary module

Secondary datasets typically contain larger numbers of records than primary datasets and new records are added frequently. Our application will have only one secondary module, named "Invoices".An invoice consists of a header where some general attributes are described (number, date, customer …) and invoice lines with the list of products sold, their quantities, prices, etc. It is convenient to have two grids for such documents: the main one (master) showing the invoice header data and the detail one showing the invoice lines. We want to place two TDBGrid components on the invoice form and link a separate TDataSource to each of them that will be linked to its respective TFDQuery. In our project. The dataset with the invoice headers (the master set) will be called qryInvoice, and the dataset with the invoice lines (the detail set) will be called qryInvoiceLine.

### The Transactions for Invoice Data

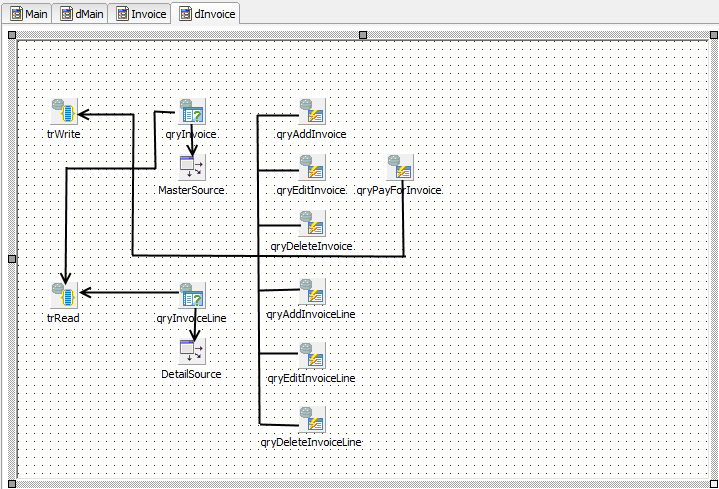
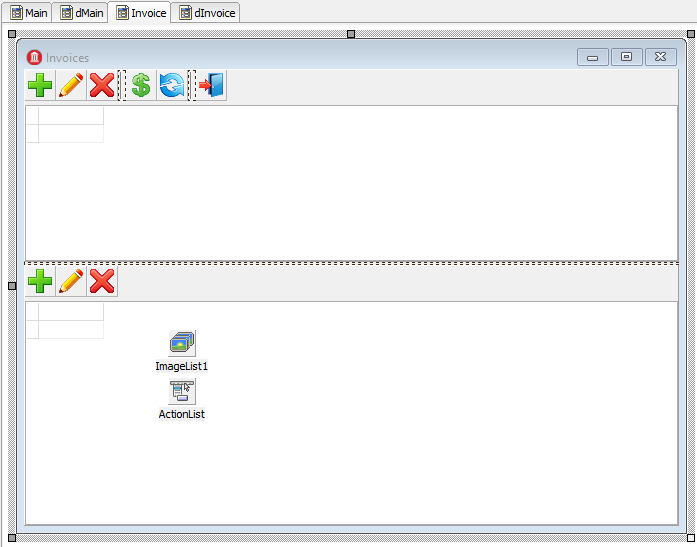
The Transaction property of each dataset will specify the read-only transaction trRead that is located in the dmInvoicedata module. Use the UpdateTransaction property to specify the trWrite transaction and the Connection property to specify the connection located in the main data module.

A Filter for the Data

Secondary datasets usually contain a field with the record creation date. In order to reduce the amount of retrieved data, a notion such as a "work period" is commonly incorporated in the application to filter the set of data sent to the client. A work period is a range of dates for which the records are required.

Since the application could have more than one secondary dataset, it makes sense to add variables containing the start and end dates of a work period to the global dmMain data module that is used by all modules working with the database in one way or another. Once the application is started, the work period could be defined by the start and end dates of the current quarter, or some other appropriate start/end date pair. The application could allow the user to change the work period while working with the application.

Configuring the Module



Since the latest invoices are the most requested ones, it makes sense to sort them by date in reverse order. The query will look like this in the SQL property of the qryInvoice dataset:

**SELECT**  
 invoice.invoice\_id **AS** invoice\_id,  
 invoice.customer\_id **AS** customer\_id,  
 customer.**NAME AS** customer\_name,  
 invoice.invoice\_date **AS** invoice\_date,  
 invoice.total\_sale **AS** total\_sale,  
 **IIF**(invoice.payed=1, 'Yes', 'No') **AS** payed  
**FROM**  
 invoice  
 **JOIN** customer **ON** customer.customer\_id = invoice.customer\_id  
**WHERE** invoice.invoice\_date **BETWEEN** :date\_begin **AND** :date\_end  
**ORDER BY** invoice.invoice\_date **DESC**

To open this dataset, it will be necessary to initialise the query parameters:

qryInvoice.ParamByName('date\_begin').AsSqlTimeStamp := dmMain.BeginDateSt;  
qryInvoice.ParamByName('date\_end').AsSqlTimeStamp := dmMain.EndDateSt;  
qryInvoice.Open;

For the purpose of illustration, we will use stored procedures to perform all operations on an invoice. Regular INSERT/UPDATE/DELETE queries can be used when operations are simple and involve writing to only one table in the database. We will execute each stored procedure as a separate query in TFDCommand objects. This component is not descended from TFDRdbmsDataSet, does not buffer data and returns not more than one result row. We are using it because it consumes fewer resources for queries that do not return data.

Since our stored procedures modify data, it is necessary to point the Transaction property of each TFDCommand object to the trWrite transaction.

**Tip**

Another alternative is to place the stored procedure calls for inserting, editing and adding a record in the corresponding properties of a TFDUpdateSQL object.

### Doing the Work

Four operations are provided for working with the invoice header: adding, editing, deleting and setting the "paid" attribute. Once an invoice is paid, we prevent any modifications to either the header or the lines. The rule is implemented at stored procedure level. Let's examine the query strings in the CommandText property for calling the stored procedures.

*qryAddInvoice.CommandText:*

**EXECUTE PROCEDURE** sp\_add\_invoice(  
 **NEXT VALUE FOR** gen\_invoice\_id,  
 :CUSTOMER\_ID,  
 :INVOICE\_DATE  
)

*qryEditInvoice.CommandText:*

**EXECUTE PROCEDURE** sp\_edit\_invoice(  
 :INVOICE\_ID,  
 :CUSTOMER\_ID,  
 :INVOICE\_DATE  
)

*qryDeleteInvoice.CommandText:*

**EXECUTE PROCEDURE** sp\_delete\_invoice(:INVOICE\_ID)

*qryPayForInvoice.CommandText:*

**EXECUTE PROCEDURE** sp\_pay\_for\_invoice(:invoice\_id)

Since our stored procedures are not called from a TFDUpdateSQL object, we need to call qryInvoice.Refresh after they are executed in order to update the data in the grid.

Stored procedures that do not require input data from the user are called as follows:

**procedure** TdmInvoice.DeleteInvoice;  
**begin**  
 *// We do everything in a short transaction*  
 trWrite.StartTransaction;  
 **try**  
 qryDeleteInvoice.ParamByName('INVOICE\_ID').AsInteger :=  
 Invoice.INVOICE\_ID.Value;  
 qryDeleteInvoice.Execute;  
 trWrite.Commit;  
 qryInvoice.Refresh;  
 **except**  
 **on** E: Exception **do**  
 **begin**  
 **if** trWrite.Active **then**  
 trWrite.Rollback;  
 **raise**;  
 **end**;  
 **end**;  
**end**;

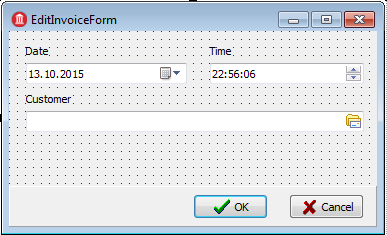
Getting User Confirmation

Before performing some operations, such as deleting an invoice, we want to get confirmation from the user:**procedure** TInvoiceForm.actDeleteInvoiceExecute(Sender: TObject);  
**begin**  
 **if** MessageDlg('Are you sure you want to delete an invoice?',

mtConfirmation,  
 [mbYes, mbNo], 0) = mrYes **then**  
 **begin**  
 Invoices.DeleteInvoice;  
 **end**;  
**end**;

### Adding or Editing Records

As with the primary modules, we will use modal forms to add a new record or edit an existing one. We will not use data-aware visual components in this implementation. As another variation, we will use a TButtonedEdit component to select a customer. It will display the name of the current customer and open a modal form with a grid for selecting a customer on the click of the embedded button. We could use something like TDBLookupCombobox, of course, but it has drawbacks: first, the customer list may be too large for scrolling comfortably through the drop-down list; secondly, the name alone may not be enough to find the customer you want.



As the window for selecting a customer, we will use the same modal form that was created for adding customers. The code for the button click handler for the TButtonedEdit component is as follows:

**procedure** TEditInvoiceForm.edtCustomerRightButtonClick(Sender: TObject);  
**var**  
 xSelectForm: TCustomerForm;  
**begin**  
 xSelectForm := TCustomerForm.Create(Self);  
 **try**  
 xSelectForm.Visible := False;  
 **if** xSelectForm.ShowModal = mrOK **then**

**begin**  
 FCustomerId := xSelectForm.Customers.Customer.CUSTOMER\_ID.Value;  
 edtCustomer.Text := xSelectForm.Customers.Customer.**NAME**.Value;  
 **end**;  
 **finally**  
 xSelectForm.Free;  
 **end**;  
**end**;

Since we are not using data-aware visual components, we need to initialize the customer code and name for displaying during the call to the edit form:

**procedure** TInvoiceForm.actEditInvoiceExecute(Sender: TObject);  
**var**  
 xEditorForm: TEditInvoiceForm;  
**begin**  
 xEditorForm := TEditInvoiceForm.Create(Self);  
 **try**  
 xEditorForm.OnClose := EditInvoiceEditorClose;  
 xEditorForm.Caption := 'Edit invoice';  
 xEditorForm.InvoiceId := Invoices.Invoice.INVOICE\_ID.Value;  
 xEditorForm.SetCustomer(  
 Invoices.Invoice.CUSTOMER\_ID.Value,  
 Invoices.Invoice.CUSTOMER\_NAME.Value);  
 xEditorForm.InvoiceDate := Invoices.Invoice.INVOICE\_DATE.AsDateTime;  
 xEditorForm.ShowModal;  
 **finally**  
 xEditorForm.Free;  
 **end**;  
**end**;

**procedure** TEditInvoiceForm.SetCustomer(ACustomerId: Integer;  
 **const** ACustomerName: **string**);  
**begin**  
 FCustomerId := ACustomerId;  
 edtCustomer.Text := ACustomerName;  
**end**;

Adding a new invoice and editing an existing one will be handled in the Close event of the modal form as it is for the primary modules. However, we will not switch the dataset to CachedUpdates mode for these because the updates are carried out by stored procedures and we are not using data-aware visual components to capture input.

**procedure** TInvoiceForm.actAddInvoiceExecute(Sender: TObject);  
**var**  
 xEditorForm: TEditInvoiceForm;  
**begin**  
 xEditorForm := TEditInvoiceForm.Create(Self);

**try**  
 xEditorForm.Caption := 'Add invoice';  
 xEditorForm.OnClose := AddInvoiceEditorClose;  
 xEditorForm.InvoiceDate := Now;  
 xEditorForm.ShowModal;  
 **finally**  
 xEditorForm.Free;  
 **end**;  
**end**;

**procedure** TInvoiceForm.AddInvoiceEditorClose(Sender: TObject;  
 **var** Action: TCloseAction);  
**var**  
 xEditorForm: TEditInvoiceForm;  
**begin**  
 xEditorForm := TEditInvoiceForm(Sender);  
 **if** xEditorForm.ModalResult <> mrOK **then**  
 **begin**  
 Action := caFree;  
 Exit;  
 **end**;  
 **try**  
 Invoices.AddInvoice(xEditorForm.CustomerId, xEditorForm.InvoiceDate);  
 Action := caFree;  
 **except**  
 **on** E: Exception **do**  
 **begin**  
 Application.ShowException(E);  
*// It does not close the window give the user correct the error*  
 Action := caNone;  
 **end**;  
 **end**;  
**end**;

**procedure** TdmInvoice.AddInvoice(ACustomerId: Integer; AInvoiceDate: TDateTime);  
**begin**  
 *// We do everything in a short transaction*  
 trWrite.StartTransaction;  
 **try**  
 qryAddInvoice.ParamByName('CUSTOMER\_ID').AsInteger := ACustomerId;  
 qryAddInvoice.ParamByName('INVOICE\_DATE').AsSqlTimeStamp :=  
 DateTimeToSQLTimeStamp(AInvoiceDate);  
 qryAddInvoice.Execute();  
 trWrite.Commit;  
 qryInvoice.Refresh;  
 **except**  
 **on** E: Exception **do**  
 **begin**  
 **if** trWrite.Active **then**  
 trWrite.Rollback;

**raise**;  
 **end**;  
 **end**;  
**end**;

Next, we move on to the details of an invoice. For the qryInvoiceLine dataset, we set the MasterSource property to the datasource that is linked to qryInvoice and the MasterFields property to INVOICE\_ID. We specify the following query in the SQL property:

**SELECT**  
 invoice\_line.invoice\_line\_id **AS** invoice\_line\_id,  
 invoice\_line.invoice\_id **AS** invoice\_id,  
 invoice\_line.product\_id **AS** product\_id,  
 product.**name AS** productname,  
 invoice\_line.quantity **AS** quantity,  
 invoice\_line.sale\_price **AS** sale\_price,  
 invoice\_line.quantity \* invoice\_line.sale\_price **AS** total  
**FROM**  
 invoice\_line  
 **JOIN** product **ON** product.product\_id = invoice\_line.product\_id  
**WHERE** invoice\_line.invoice\_id = :invoice\_id

As with the invoice header, we will use stored procedures to perform all modifications. Let's examine the query strings in the CommandText property of the commands that call the stored procedures.

*qryAddInvoiceLine.CommandText:*

**EXECUTE PROCEDURE** sp\_add\_invoice\_line(  
 :invoice\_id,  
 :product\_id,  
 :quantity  
)

*qryEditInvoiceLine.CommandText:*

**EXECUTE PROCEDURE** sp\_edit\_invoice\_line(  
 :invoice\_line\_id,  
 :quantity  
)

*qryDeleteInvoiceLine.CommandText:*

**EXECUTE PROCEDURE** sp\_delete\_invoice\_line(  
 :invoice\_line\_id  
)

As with the header, the form for adding a new record and editing an existing one does not use data-aware visual components. To select a product, we use the TButtonedEdit component again. The code for the on-click handler for the button in the TButtonedEdit object is as follows:

**procedure** TEditInvoiceLineForm.edtProductRightButtonClick(Sender: TObject);  
**var**  
 xSelectForm: TGoodsForm;  
**begin**  
 **if** FEditMode = emInvoiceLineEdit **then**  
 Exit;  
 xSelectForm := TGoodsForm.Create(Self);  
 **try**  
 xSelectForm.Visible := False;  
 **if** xSelectForm.ShowModal = mrOK **then**  
 **begin**  
 FProductId := xSelectForm.Goods.Product.PRODUCT\_ID.Value;  
 edtProduct.Text := xSelectForm.Goods.Product.**NAME**.Value;  
 edtPrice.Text := xSelectForm.Goods.Product.PRICE.AsString;  
 **end**;  
 **finally**  
 xSelectForm.Free;  
 **end**;  
**end**;

Since we are not using data-aware visual components, again we will need to initialize the product code and name and its price for displaying on the edit form.

**procedure** TInvoiceForm.actEditInvoiceLineExecute(Sender: TObject);  
**var**  
 xEditorForm: TEditInvoiceLineForm;  
**begin**  
 xEditorForm := TEditInvoiceLineForm.Create(Self);  
 **try**  
 xEditorForm.EditMode := emInvoiceLineEdit;  
 xEditorForm.OnClose := EditInvoiceLineEditorClose;  
 xEditorForm.Caption := 'Edit invoice line';  
 xEditorForm.InvoiceLineId := Invoices.InvoiceLine.INVOICE\_LINE\_ID.Value;  
 xEditorForm.SetProduct(  
 Invoices.InvoiceLine.PRODUCT\_ID.Value,  
 Invoices.InvoiceLine.PRODUCTNAME.Value,  
 Invoices.InvoiceLine.SALE\_PRICE.AsCurrency);  
 xEditorForm.Quantity := Invoices.InvoiceLine.QUANTITY.Value;  
 xEditorForm.ShowModal;  
 **finally**  
 xEditorForm.Free;  
 **end**;  
**end**;

**procedure** TEditInvoiceLineForm.SetProduct(AProductId: Integer;

AProductName: **string**; APrice: Currency);  
**begin**  
 FProductId := AProductId;  
 edtProduct.Text := AProductName;  
 edtPrice.Text := CurrToStr(APrice);  
**end**;

We will handle adding a new item and editing an existing one in the Close event of the modal form.

**procedure** TInvoiceForm.actAddInvoiceLineExecute(Sender: TObject);  
**var**  
 xEditorForm: TEditInvoiceLineForm;  
**begin**  
 xEditorForm := TEditInvoiceLineForm.Create(Self);  
 **try**  
xEditorForm.EditMode := emInvoiceLineAdd;  
xEditorForm.OnClose := AddInvoiceLineEditorClose;  
 xEditorForm.Caption := 'Add invoice line';  
 xEditorForm.Quantity := 1;  
 xEditorForm.InvoiceId := Invoices.Invoice.INVOICE\_ID.Value;  
 xEditorForm.ShowModal;  
 **finally**  
 xEditorForm.Free;  
 **end**;  
**end**;

**procedure** TInvoiceForm.actEditInvoiceLineExecute(Sender: TObject);  
**var**  
 xEditorForm: TEditInvoiceLineForm;  
**begin**  
 xEditorForm := TEditInvoiceLineForm.Create(Self);  
 **try**  
 xEditorForm.EditMode := emInvoiceLineEdit;  
 xEditorForm.OnClose := EditInvoiceLineEditorClose;  
 xEditorForm.Caption := 'Edit invoice line';  
 xEditorForm.InvoiceLineId := Invoices.InvoiceLine.INVOICE\_LINE\_ID.Value;  
 xEditorForm.SetProduct(  
 Invoices.InvoiceLine.PRODUCT\_ID.Value,  
 Invoices.InvoiceLine.PRODUCTNAME.Value,  
 Invoices.InvoiceLine.SALE\_PRICE.AsCurrency);  
 xEditorForm.Quantity := Invoices.InvoiceLine.QUANTITY.Value;  
 xEditorForm.ShowModal;  
 **finally**  
 xEditorForm.Free;  
 **end**;  
**end**;

**procedure** TInvoiceForm.AddInvoiceLineEditorClose(Sender: TObject;  
 **var** Action: TCloseAction);  
**var**  
 xEditorForm: TEditInvoiceLineForm;

xCustomerId: Integer;  
**begin**  
 xEditorForm := TEditInvoiceLineForm(Sender);  
 **if** xEditorForm.ModalResult <> mrOK **then**  
 **begin**  
 Action := caFree;  
 Exit;  
 **end**;  
 **try**  
 Invoices.AddInvoiceLine(xEditorForm.ProductId, xEditorForm.Quantity);  
 Action := caFree;  
 **except**  
 **on** E: Exception **do**  
 **begin**  
 Application.ShowException(E);  
 *// It does not close the window give the user correct the error*  
 Action := caNone;  
 **end**;  
 **end**;  
**end**;

**procedure** TInvoiceForm.EditInvoiceLineEditorClose(Sender: TObject;  
 **var** Action: TCloseAction);  
**var**  
 xCustomerId: Integer;  
 xEditorForm: TEditInvoiceLineForm;  
**begin**  
 xEditorForm := TEditInvoiceLineForm(Sender);  
 **if** xEditorForm.ModalResult <> mrOK **then**  
 **begin**  
 Action := caFree;  
 Exit;  
 **end**;  
 **try**  
 Invoices.EditInvoiceLine(xEditorForm.Quantity);  
 Action := caFree;  
 **except**  
 **on** E: Exception **do**  
 **begin**  
 Application.ShowException(E);  
 *// It does not close the window give the user* correct the error  
 Action := caNone;  
 **end**;  
 **end**;  
**end**;

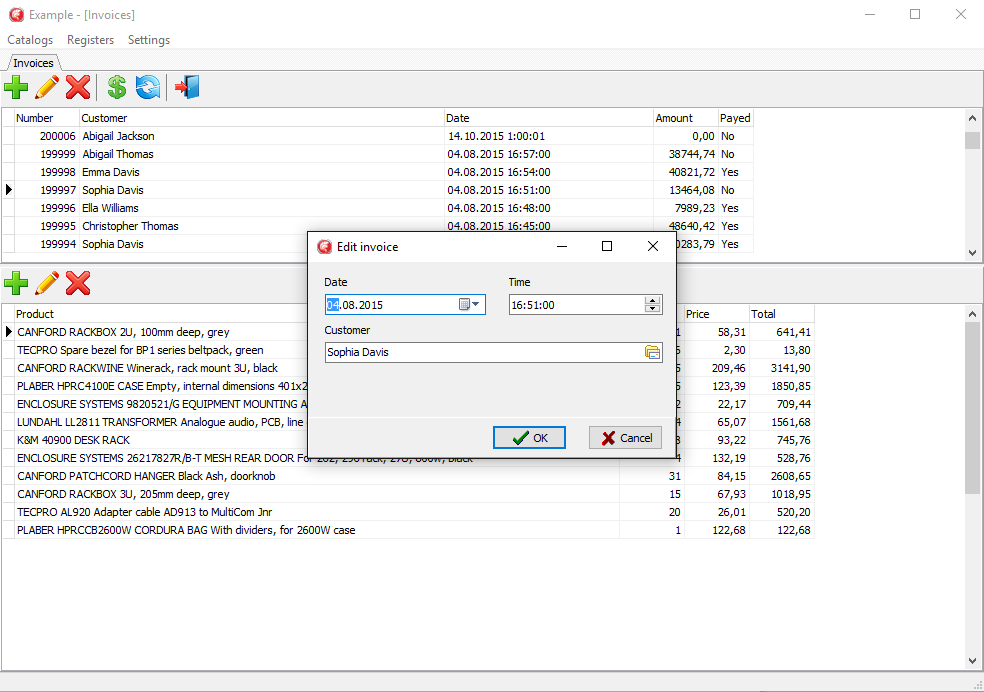
Now let's take a look at the code for the AddInvoiceLine and EditInvoiceLine procedures of the dmInvoice data module:

**procedure** TdmInvoice.AddInvoiceLine(AProductId: Integer; AQuantity: Integer);  
begin  
 // We do everything in a short transaction  
 trWrite.StartTransaction;  
 **try**  
 qryAddInvoiceLine.ParamByName('INVOICE\_ID').AsInteger :=  
 Invoice.INVOICE\_ID.Value;  
 **if** AProductId = 0 **then**  
 **raise** Exception.Create('Not selected product');  
 qryAddInvoiceLine.ParamByName('PRODUCT\_ID').AsInteger := AProductId;  
 qryAddInvoiceLine.ParamByName('QUANTITY').AsInteger := AQuantity;  
 qryAddInvoiceLine.Execute();  
 trWrite.Commit;  
 qryInvoice.Refresh;  
 qryInvoiceLine.Refresh;  
 **except**  
 **on** E: Exception **do**  
 **begin**  
 **if** trWrite.Active **then**  
 trWrite.Rollback;  
 **raise**;  
 **end**;  
 **end**;  
**end**;

**procedure** TdmInvoice.EditInvoiceLine(AQuantity: Integer);  
**begin**  
 // We do everything in a short transaction  
 trWrite.StartTransaction;  
 **try**  
 qryEditInvoiceLine.ParamByName('INVOICE\_LINE\_ID').AsInteger :=  
 InvoiceLine.INVOICE\_LINE\_ID.Value;  
 qryEditInvoiceLine.ParamByName('QUANTITY').AsInteger := AQuantity;  
 qryEditInvoiceLine.Execute();  
 trWrite.Commit;  
 qryInvoice.Refresh;  
 qryInvoiceLine.Refresh;  
 **except**  
 **on** E: Exception **do**  
 **begin**  
 **if** trWrite.Active **then**  
 trWrite.Rollback;  
 **raise**;  
 **end**;  
 **end**;  
**end**;

## The Result

As a result, we have an application that looks like this:



## Conclusion

FireDac is a standard set of data-access and data-aware visual components for developing with various database systems, including Firebird, starting from Delphi™; XE3. FireDac™; ships with the higher-end versions of Delphi. Many independent sets of data access and data-aware visual components are available for working with Firebird, some commercial, others distributed under a variety of licences, including open source and freeware. They include FibPlus, IBObjects, UIB, UniDAC, IBDac, Interbase Express (IBX) and more. The principles for developing Firebird applications in Delphi™ are the same, regardless of the components you choose.

All queries to a database are executed within a transaction. To guarantee that applications will work correctly and efficiently with Firebird databases, it is advisable to manage transactions manually, by explicit calls to the StartTransaction, Commit and Rollback methods of the TFDTransaction component. Transactions should be as short as possible and you can use as many as the logic of your application requires. .

The recommended configuration for a long-running, read-only transaction to view datasets is to use READ\_COMMITTED isolation with REC\_VERSION for conflict resolution. An application can run many datasets in one such transaction or one for each dataset, according to the requirements of the design.

Read/write transactions must be as short as possible. If you use complex logic while editing data--change several tables, calculate stored aggregates, and so on--the SNAPSHOT isolation level is recommended. SNAPSHOT is recommended also for report forms, especially when they involve executing large numbers of queries.

To avoid holding an uncommitted transaction during an editing session, either use visual components that are not data-aware or use CachedUpdates mode. With CachedUpdates you can restrict writes to short bursts of activity, keeping the read/write transaction active only for as long as it takes to post the most recent change to the database.

The TFDUpdateSQL component is necessary for editing most datasets. Update queries are governed by its InsertSQL, ModifySQL, DeleteSQL and FetchRowSQL properties. The queries for those properties can be generated automatically by a wizard but manual corrections or adjustments are often required.Acquiring values for Auto-incrementing primary keys can be handled in one of two ways:

* Getting the value from the generator beforehand by specifying the UpdateOptions.GeneratorName and UpdateOptions.AutoIncFields properties for the TFDQuery object. This method cannot be used for auto-incrementing fields of the IDENTITY type that was introduced in Firebird 3.
* Getting the value by adding a RETURNING clause to the InsertSQL query. For this method you need to specify Required = False and ReadOnly = True for the field because the value is not entered directly.

It is convenient and sometimes necessary to implement more complex business logic with stored procedures. Using the TFDCommand component to execute stored procedures that do not return data reduces resource consumption.

## Source Code

ObjectPascal source code for the sample project is available for download using the following link: <https://github.com/sim1984/FireDacEx>.

For links to the database scripts and ready-to-use databases, please refer to the final sections of the database chapter.