# Developing applications in Delphi with FireDac

This chapter will describe the process of developing applications for the Firebird DBMS with the FireDac data access components and the Delphi XE5 environment. FireDac is a standard set of components for accessing various databases starting from Delphi XE3. 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. It 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 supposed to work with data. For example, TFDConnection is 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 makes it possible to specify the path to the client library. If it is not specified, the connection to Firebird will be established via libraries registered in the system, for example, in system32, which may be undesirable in some cases.

### Path to the client library

We will place the necessary library in the fbclient folder located in the application folder. We will 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!**

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. Besides 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, it must be one level up from the folder with the client library, i.e. in the application folder in our case. For Firebird 3, it must be in the same folder as the client library, i.e. in the application folder. For Firebird 3, it must be in the same folder as the client library, i.e. in the fbclient folder. Denis, the last two sentences do not make sense.

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.

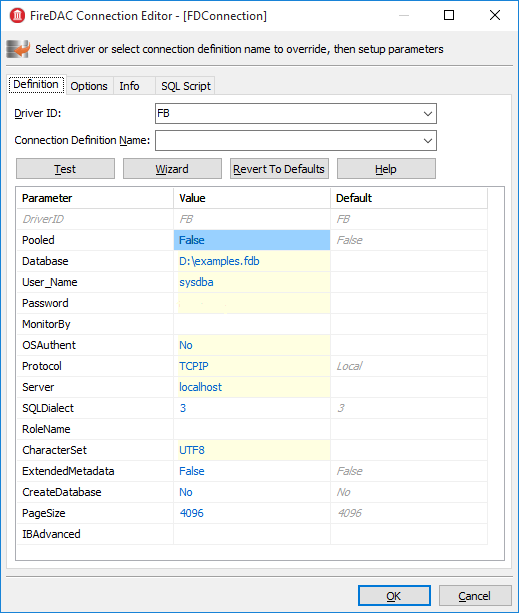
(Denis, Firebird 3 embedded still requires exclusive access if the installed full server is in Super mode.)

### Connection parameters

The Params property of the TFDConnection component contains the database connection parameters (username, password, connection character set, etc.). If you use 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 | Username. |
| Password. | Password. |
| OSAuthent | Whether the 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 | 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. If this property is not specified explicitly, the value from the Transaction property of the dataset? of the connection? will be used. (Denis, please make this explicit. I use IBObjects, which is intelligent about transaction management. I don't use FireDac myself.)

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**;

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, either this file will be virtualized in Program Data or there may be issues with modifying it and reading the new settings later.

### 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. 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 do 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;

## TFDTransaction component

The TFDTransaction component is used to manage transactions explicitly.

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 not to use it like that. For applications to work correctly with databases, it is advisable to manage transactions manually, which means calling the StartTransaction, Commit and Rollback methods of the TFDTransaction component explicitly.

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. The default value is True.  If the value of this property is set to True, FireDAC does the following:  It starts a transaction (if required) before each SQL command and ends the transaction after the SQL command is executed. If the command is successfully executed, the transaction will be ended as COMMIT. Otherwise, it will be ended as ROLLBACK.  If the application calls the StartTransaction method, the automatic transaction management will be disabled until the transaction is ended by Commit or Rollback.  The automatic transaction management in Firebird is emulated by FireDAC components themselves. |
| 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 if the transaction is active. The default value is xdCommit. 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.   The default value for the similar property in other access components is xdRollback. So you need to set this property manually to the value that is actually needed. |
| Options.EnableNested | Controls nested transactions. The default value is True.  When a transaction is active, calling StartTransaction again will create a nested transaction. FireDAC emulates nested transactions using savepoints if the DBMS does not support nested transactions as they are. To disable nested transactions, set EnableNested to False. When False is set, calling StartTransaction inside the transaction will raise an exception.  Firebird does not support nested transactions as such. |
| Options.Isolation | Specifies the transaction isolation level. It is the most important transaction property. The default value is xiReadCommitted. The following values are possible:   * xiUnspecified – the default isolation level of your DBMS is used (it is SNAPSHOT in Firebird, i.e. with the following parameters: read write concurrency wait); * xiDirtyRead – there is no such isolation level in Firebird so READ COMMITTED will be used instead; * xiReadCommitted – the READ COMMITTED isolation level. FireDac starts ReadCommitted transactions in Firebird with the following parameters: read/write read\_committed rec\_version nowait; * xiRepeatableRead – there is no such isolation level in Firebird so SNAPSHOT will be used instead; * xiSnapshot – the SNAPSHOT isolation level. FireDac starts Snapshot transactions in Firebird with the following parameters: read/write concurrency wait; * xiSerializable – the SERIALIZABLE isolation level. Firebird does not support serializable isolation, but FireDac emulates it by starting a transaction with the following parameters: read write consistency wait. |
| Options.Params | DBMS-specific transaction parameters. It is currently used for Firebird and Interbase only. Possible values:   * read * write * read\_committed * concurrency * consistency * wait * nowait * rec\_version * no rec\_version |
| Options.ReadOnly | Specifies whether it is a read-only transaction. The default value is False. If it is set to True, any changes within the current transaction are not possible, there is no read/write value in the transaction parameters in Firebird in this case.  Setting this property to True allows the DBMS to optimize the usage of resources. |

Unlike most other DBMSs, Firebird allows as many TFDTransaction components 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 write transaction for each dataset dataset.

We will not rely on starting and ending transactions automatically and that is why Options.AutoCommit = False, Options.AutoStart = False and Options.AutoStop = False in all transactions.

## Datasets

It is possible to work with data in FireDac with the help of the FDQuery, FDTable, FDStoredProc, FDCommand components, but FDCommand is not a dataset.

TFDQuery, TFDTable and TFDStoredProc are inherited from TFDRdbmsDataSet. Besides datasets for working with the database directly, FireDac also has the TFDMemTable component for working with in-memory datasets which is analogous to TClientDataSet.

The main component for working with datasets is TFDQuery. This component can be used for practically any purpose. The TFDTable and TFDStoredProc components are just its modifications, either a bit expanded or truncated. We will neither dwell upon them nor use them in our application. If you wish, you can learn about them in the FireDac documentation.

The purpose of the component is to buffer records retrieved by the SELECT statement in order to show them in a grid and to provide for the current record in the buffer (grid) to be editable. Unlike the IBX.IBDataSet component, the FDQuery component contains no RefreshSQL, InsertSQL, UpdateSQL and DeleteSQL properties. Instead, editability is provided by the FDUpdateSQL component that is specified in the UpdateObject property.

**Note**

Sometimes it is possible to make the FDQuery component editable without specifying the UpdateObject property and specifying Insert/Update/Delete queries by just setting the property UpdateOptions.RequestLive = True. The modification queries will be generated automatically in this case. However, this approach has a lot of limitations regarding the main SELECT query so it is not wise to rely on it.

Table 3. TFDQuery component main properties

|  |  |
| --- | --- |
| **Parameter** | **Purpose** |
| Connection | Reference to the FDConnection component. |
| MasterSource | Reference to the master data source (TDataSource) for FDQuery used as Detail. |
| Transaction | The transaction within which the query specified in the SQL property will be executed. If the property is not specified, the default transaction for the connection will be used. |
| UpdateObject | Reference to the FDUpdateSQL component that provides that the dataset is editable when the SELECT query does not meet the requirements for the 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 dataset?) will be used. |
| UpdateOptions.CheckRequired | If the CheckRequired property is set to True, FireDac controls the Required property of the corresponding fields, i.e. those fields that have the NOT NULL restriction. By default, it is set to True.  If CheckRequired=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. It may be undesirable 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.FetchGeneratorsPoint | 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. It 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; * gpDeferred– the next value is fetched from the generator before a new record is posted in the database, i.e. while calling the Post or ApplyUpdates method.   The default value isgpDeferred. |
| 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 "live", i.e. editable if possible. The Insert/Update/Delete queries will be generated automatically in this case. This setting imposes a lot of limitations on the SELECT query. It is supported for backward compatibility with the BDE and is not recommended. |
| UpdateOptions.UpdateMode | Controls how to check whether a record has been modified. This property makes it possible to control possible overwrites of updates in cases where one user is taking a long time to edit a record while another user is simultaneously editing the same record and completes the update earlier. The first user might not even know that the record has been updated (maybe more than once) during his editing session, which could cause his own changes to overwrite these updates:   * upWhereAll – check whether a record exists by its primary key + check all columns for old values. For instance   update table set ...  where pkfield = :old\_ pkfield and  client\_name = :old\_client\_name and  info = :old\_info ...  With this setting, the 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 — for instance, minimum and maximum wages, etc.   * upWhereChanged – check whether a record exists by its primary key + check only the columns being edited for old values.   update table set ...  where pkfield = :old\_pkfield and  client\_name = :old\_client   * upWhereKeyOnly (default) – check whether a record exists by its primary key.   This check corresponds to the automatically generated query for UpdateSQL. To avoid (or handle) update conflicts in a multi-user environment, it is typically necessary to add WHERE conditions manually. And, of course, while implementing the analog of upWhereChanged, a similar tactic is needed to remove the unused column modifications from the update table set, leaving in the update list only the columns that are actually modified. Otherwise, the query will overwrite someone else's updates of this record. As you can see, it means that it is necessary to dynamically create the UpdateSQL query.  If you want to specify the settings for detecting update conflicts individually for each field, you can use the ProviderFlags property for each field. |
| CachedUpdates | Specifies whether the dataset cache changes without immediately applying them to the database. If this property is set to True, any changes (Insert/Post, Update/Post, Delete) are not applied to the database at once, but saved to a special log. The application must explicitly apply the changes by calling the ApplyUpdates method. In this case, 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 SQL query. If this property contains the SELECT statement, execute it by calling the Open method. Otherwise, the Execute or ExecSQL method should be used.  (Note to Denis: In my opinion, the Open method for datasets is another legacy from the BDE and just mimics Paradox and other table-based systems. I would prefer to recommend calling Execute/ExecSQL always with Firebird.) |

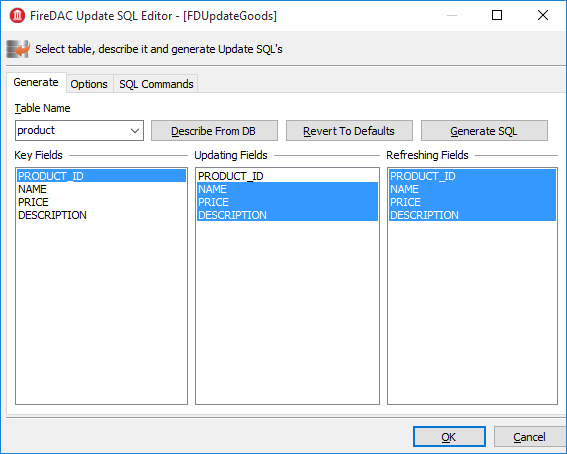
## TFDUpdateSQL component

The TFDUpdateSQL component makes it possible to redefine the SQL command that Delphi generates for updating a dataset automatically. It can be used to update the TFDQuery, TFDTable and TFDStoredProc components. Using TFDUpdateSQL is optional for the TFDQuery and TFDTable components because these components can automatically generate commands for posting updates from a dataset to the DBMS. Using TFDUpdateSQL is required for updating a TFDStoredProc dataset. We recommend that you always use it, even in the most simple cases, in order to have 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.

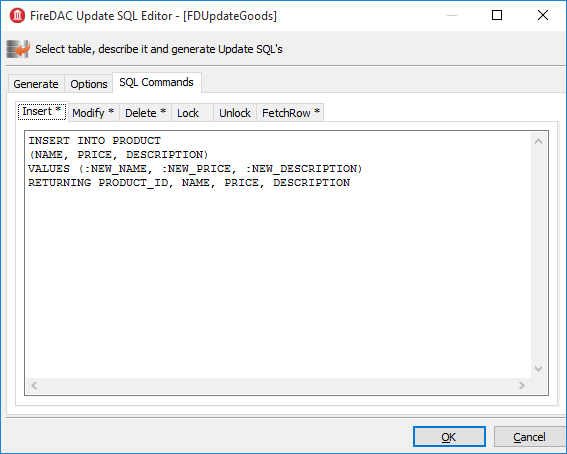
**Note**

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. To do it, 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.

The queries will be generated automatically after that and 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 the trigger (with a generator) or it is an IDENTITY column (starting from Firebird 3.0). When a value is fetched from the generator for this column from the server, it is recommended to manually add the PRODUCT\_ID column to the RETURNING clause of the INSERT statement.

The Options tab contains some properties that can affect the process of query generation. These properties are not related to the TFDUpdateSQL component itself. Instead, they are references to the UpdateOptions properties of the dataset that has the current TFDUpdateSQL specified for it in the UpdateObject property. It is implemented like this only for convenience.

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 one current (updated, inserted) record. (RefreshSQL) |
| InsertSQL | The SQL query for inserting a record. |
| LockSQL | The SQL query for locking one current record. (FOR UPDATE WITH LOCK). |
| ModifySQL | The SQL query for modifying a record. |
| UnlockSQL | The SQL query for unlocking the current record. It is not used in Firebird. |

As you can notice, the TFDUpdateSQL component has no Transaction property. It is because the component does not make modification queries directly. Instead, it just replaces queries automatically generated in the dataset parent to TFDRdbmsDataSet.

## TFDCommand component

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

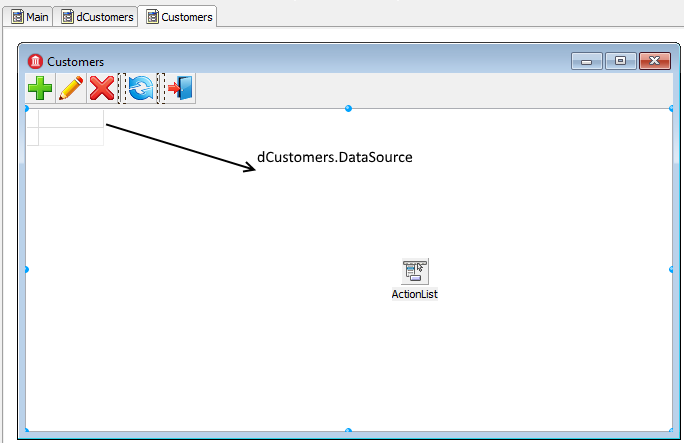
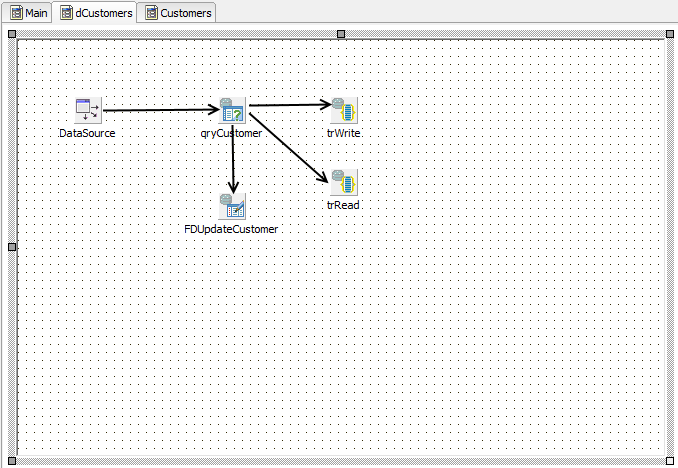
Table 5. TFDCommand component main properties

|  |  |
| --- | --- |
| **Parameter** | **Purpose** |
| Connection | Reference to the FDConnection component. |
| Transaction | The transaction the SQL command will be executed within. |
| CommandKind | Command type.   * skUnknown – unknown. In this case, the internal parser will automatically determine the command type by the text of the command; * skStartTransaction – the command for starting the transaction; * skCommit – the command for ending and coimmitting the transaction; * skRollback – the command for ending and rolling back the transaction; * skCreate – the CREATE … command for creating a new new metadata object; * skAlter – the ALTER … command for altering a metadata object; * skDrop – the DROP … command for deleting a metadata object; * skSelect – the SELECT command for retrieving data; * skSelectForLock – the SELECT … WITH LOCK command for locking the selected rows; * skInsert – the INSERT … command for inserting a new record; * skUpdate – the UPDATE … command for modifying records; * skDelete – the DELETE … command for deleting records; * skMerge – the MERGE INTO … command * skExecute – the EXECUTE PROCEDURE or EXECUTE BLOCK command; * skStoredProc – calling a stored procedure; * skStoredProcNoCrs – calling a stored procedure that returns no cursor; * skStoredProcWithCrs – calling a stored procedure that returns a cursor.   Usually, the command type is determined automatically by the text of the SQL query. |
| CommandText | SQL query text. |

## 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 with a TDBGrid grid and a toolbar with buttons. The business logic of working with a dataset will be located in a separate DataModule that contains a TDataSource data source, a TFDQuery dataset, read and write TFDTransaction transactions.

Let us create the customer dataset as our example for creating datasets.



**Note**

The DataSource component is not visible because it is not on the form, but in the dCustomers module.

Let us place the TFDQuery component in the dCustomers module and name it qryCustomers. This dataset will be specified in the DataSet property of the DataSource data source. We specify the trRead read-only transaction in the Transaction property, the trWritetransaction in the UpdateTransaction property and the connection located in the main data module in the Connection property. We specify the following query in the SQL property:

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

The trRead read transaction is started when the dataset form is displayed (the OnActivate event) and is ended when the form is closed. The 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. This way our transaction will have the following parameters: read read\_committed rec\_version. A transaction with 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 such a transaction is started on the server as committed). We set the Options.DisconnectAction property to xdCommit,which perfectly fits a read-only transaction. So, 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

**Note**

You cannot use such a transaction for reports (especially if they use several sequential queries) because a transaction with the READ COMMITTED isolation level will see all new committed changes while rereading data.

It is recommended to use a short read-only transaction with the SNAPSHOT isolation level (Options.Isolation = xiSnapshot and Options.ReadOnly= True) for reports. We do not dwell upon working with reports in this example.

The trWrite write transaction must be as short as possible in order 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 write transaction is very short, we can use the SNAPSHOT isolation level. Thus, our write transaction will have the following parameters: Options.ReadOnly=False and Options.Isolation = xiSnapshot. The default value of the Options.DisconnectAction property is no good for write transactions, its value 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 must have the following properties:

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

Actually, it is not necessary to specify the SNAPSHOT isolation level for simple INSERT/UPDATE/DELETE. However, if a table has complex triggers or a stored procedure is executed instead of simple INSERT/UPDATE/DELETE queries, it is advisable to use the SNAPSHOT isolation level.

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

To be able to edit a dataset, you should specify the InsertSQL, ModifySQL, DeleteSQL and FetchRowSQL properties. It is possible to have the wizard generate these statements but it may be necessary to correct some things after that. 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

In this sample, we will get the value from the generator before making an insert into the table. To do it, you should specify the following values for the properties of the TFDQuery component: UpdateOptions.GeneratorName = GEN\_CUSTOMER\_ID and UpdateOptions.AutoIncFields = CUSTOMER\_ID. Another way to get the value of the generator (auto-incremental field) is to return it after the INSERT is executed with the use of a RETURNING clause. This method will be shown later.

Modal forms are usually used to add a new record or to edit an existing one. Once the modal form is closed with 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 the Insert/Edit mode, i.e. before Post. But it is possible switch the dataset to the Insert/Edit mode only 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 after the 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 allows you to keep the transaction active only for a very short period of time (to be exact, for the time the changes are being made).
2. Give up using database-aware visual components. However, this way will require 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 us see the code for editing a supplier 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**;

In order to understand the internal processes, we can look at 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**;

You can see from the code 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 accumulate not more than one record in the buffer, the transaction will be active for a very short time, which is exactly what we need.

The product dataset is created similarly to the customer one. However, we will use it to demonstrate the other method of getting auto-incremental values.

The main query will look like this:

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

The TFDUpdateSQL.InsertSQL component property will contain the following query:

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

This query contains the RETURNING clause that will return the value of the PRODUCT\_ID field after it is changed in the BEFORE INSERT trigger. In this case, it makes no sense to specify the value of the UpdateOptions.GeneratorName property. Besides, it is necessary to specify the following properties for the PRODUCT\_ID field because the value of this field is not entered directly: Required = False and ReadOnly = True. Everything else is set up similarly to way we did it for the Customer dataset .

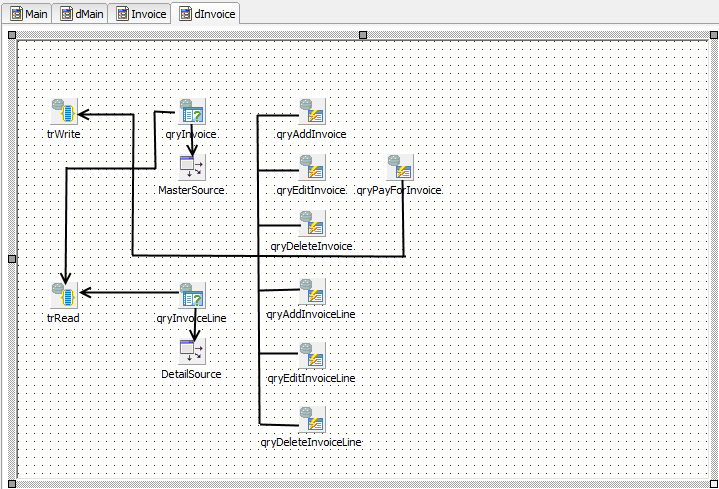
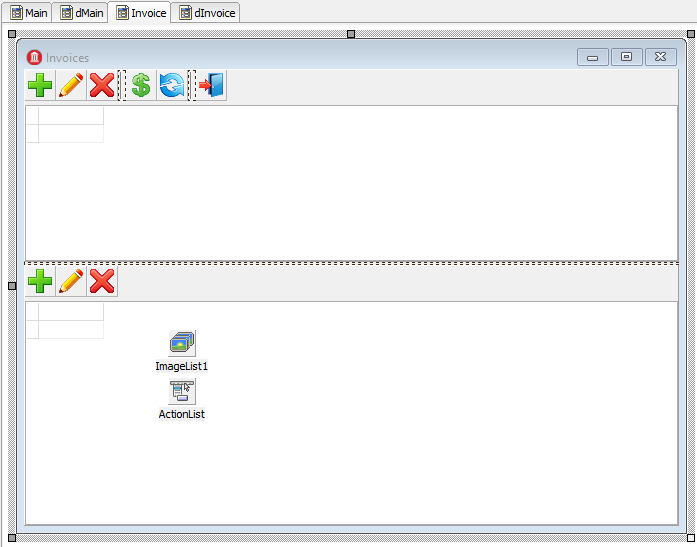
## Creating a secondary module

There will be only one secondary module in our application, named "Invoices". Unlike primary datasets, secondary datasets typically contain larger numbers of records and new records are added frequently.

An invoice consists of a title where some general attributes are described (number, date, customer …) and invoice lines with the list of products, their quantities, prices, etc. It is convenient to have two grids for such documents: the main one (master) shows the data about the header of the invoice and the detailed one shows the list of products. That is why we need to place two TDBGrid components on the document form and link a separate TDataSource to each of them that will be linked to its respective TFDQuery. In our sample, the dataset with document headers (the master set) will be called qryInvoice, and the dataset with document lines (the detail set) will be called qryInvoiceLine.

We will use the Transaction property of both datasets to specify the trRead read-only transaction 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.

Secondary datasets usually contain a field with the document creation date. In order to reduce the amount of retrieved data, a notion such as a work period is commonly implemented in the application to reduce 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 can 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 dates when the current quarter starts and ends, or some other appropriate start/end date pair. The application could include the capability for the user to change the work period while working with the application.



Since the latest documents are the most requested ones, it makes sense to sort them by date in reverse order. Taking that into account, 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**

While opening 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 our sample application, 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 every stored procedure as a separate query in TFDCommand components. This component is not parent to TFDRdbmsDataSet, does not buffer data and returns not more than one result row so using it consumes fewer resources for queries that do not return data. Since our stored procedures modify data, it is necessary to specify the trWrite transaction in the Transaction property of the TFDCommand components.

**Note**

It is also possible to place stored procedures for inserting, editing and adding a record in the corresponding properties of the TFDUpdateSQL component.

Four operations are provided for working with the header of an invoice: adding, editing, deleting and settings the "paid" attribute. As soon as an invoice is paid, we forbid any modifications to it, either in its header or in its lines. The rule is implemented at the stored procedure level. Let us look at the query strings in the CommandText property for calling 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\_inovice(:invoice\_id)

Since our stored procedures are called not from the TFDUpdateSQL component, it is necessary to call qryInvoice.Refresh after they are executed in order to update the data in the grid.

Stored procedures that do not require data to be entered are called in the following way:

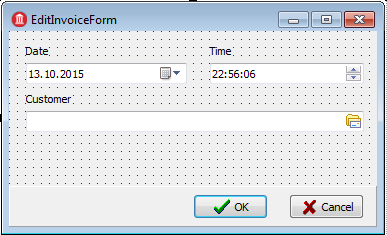
**procedure** TdmInvoice.DeleteInvoice;  
**begin**  
 *// We do everything in a short transaction*  
 trWrite.StartTransaction;  
 **try**  
 qryDeleteInvoice.ParamByName('INVOICE\_ID').AsInteger :=  
 Invoice.INVOICE\_ID.Value;  
 qryDeleteInvoice.Execute;  
 trWrite.Commit;  
 qryInvoice.Refresh;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

Before performing some operations, it is necessary to get confirmation from the user, for instance, when an invoice is deleted:

**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**;

As with the primary modules, we will use modal forms to add a new record or edit an existing one. We will not use database-aware visual components in this case. Another peculiarity is that we will use the 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 once the button is clicked. Of course, we could use something like TDBLookupCombobox, but, first, there may be very many customers and it will be inconvenient to scroll such a drop-down list and, second, it may be not enough to specify only a name in order to find the customer you need.



As a window for selecting a customer, we will use the same modal form that was created for adding customers. The code for the handler of a click on the button in the TButtonedEdit component will look like this:

**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 database-aware visual components, we will need to initialize the code of the customer and its name for displaying it while calling 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 we did for the primary modules. However, we will not switch the dataset to the CachedUpdates mode in this case because the modification is carried out by stored procedures and we are not using database-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**;

Now let us proceed 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 header of an invoice, we will use stored procedures to perform all modifications. Let us look at the query strings in the CommandText property of the queries for calling 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 database-aware visual components. To select a product, we will use the TButtonedEdit component. The code for the handler of a click on the button in the TButtonedEdit component will look like this:

**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 database-aware visual components , we will need to initialize the product's code, name and 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**;

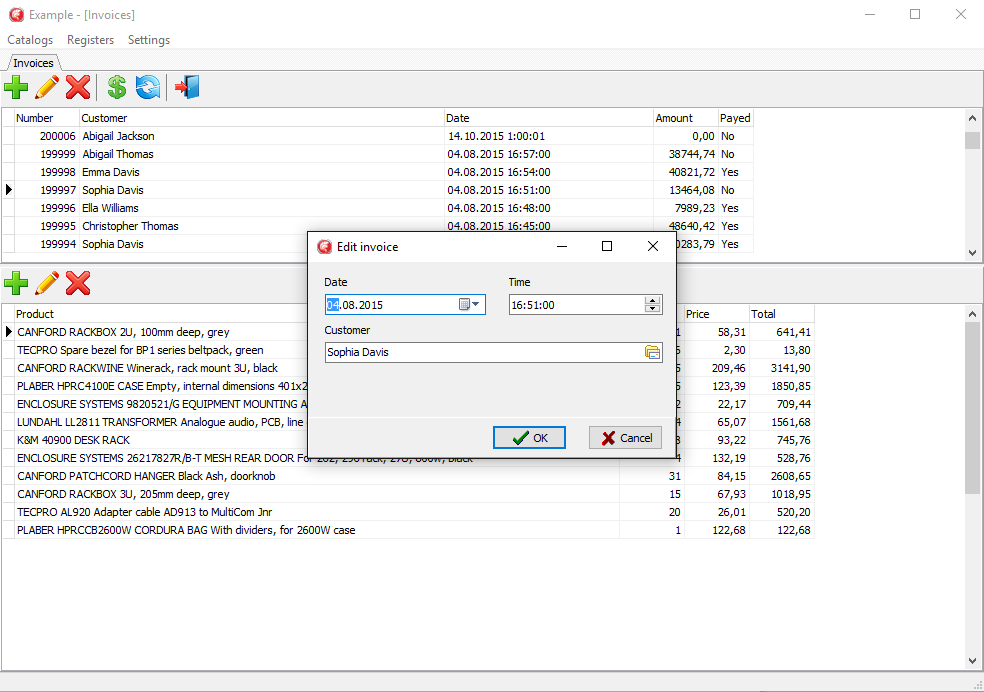
**procedure** TInvoiceForm.EditInvoiceLineEditorClose(Sender: TObject;  
 **var** Action: TCloseAction);  
**var**  
 xCustomerId: Integer;  
 xEditorForm: TEditInvoiceLineForm;  
**begin**  
 xEditorForm := TEditInvoiceLineForm(Sender);  
 **if** xEditorForm.ModalResult <> mrOK **then**  
 **begin**  
 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 us 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**;

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



**Conclusion**

There are a lot of access components in Delphi for working with the Firebird DBMS (Interbase Express (IBX), FibPlus, UIB, UniDAC, IBDac, FireDac). FireDac is a standard set of components for accessing various databases starting from Delphi XE3.

Denis, you forgot about IBObjects!

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

If you need a long read transaction to view datasets, it is advisable to start such a transaction with the following parameters: READ READ\_COMMITTED REC\_VERSION. You can start either one such transaction for all datasets or one transaction for each dataset.

Write transactions must be as short as possible. If you use a complex logic while editing data (change several tables, calculate stored aggregates, etc.), you need to use the SNAPSHOT isolation level. In order not to hold a transaction during the process of editing in the edit forms, you need to avoid using database-aware visual components or use the CachedUpdates mode. The CachedUpdates mode allows you to keep the transaction active only for a very short period of time (to be exact, for the time it takes to post the changes to the database).

For report forms, especially when a lot queries are executed, it is necessary to use a transaction with the SNAPSHOT isolation level.

To be able to edit a dataset, it is necessary to use the TFDUpdateSQL component and fill in its InsertSQL, ModifySQL, DeleteSQL and FetchRowSQL properties. It is possible to generate these properties by the wizard, but it may be necessary to correct some things after that.

There are two ways to work with auto-incremental primary keys:

* Getting the value from the generator beforehand. To do it, you need to specify the UpdateOptions.GeneratorName and UpdateOptions.AutoIncFields properties for the TFDQuery component. This method cannot be used for auto-incrementing fields of the IDENTITY type that was introduced in Firebird 3.
* Getting the values of the primary key with the help of the RETURNING clause that must be added to the InsertSQL query. In this case, it is necessary to specify the following properties for the field because the value of this field is not entered directly: Required = False and ReadOnly = True.

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.

You can get the source code of the sample application using the following link <https://github.com/sim1984/FireDacEx>