



Firebird 3.0 Developer's Guide PRE-BETA

Release 0.1

Denis Simonov

Megatranslations Ltd, translation of text from
the original Russian to English: Dmitry Borodin

Editor of the translated text: Helen Borrie

3 November 2017, document version 0.101

Firebird 3.0 Developer's Guide PRE-BETA

Release 0.1

3 November 2017, document version 0.101

by Denis Simonov

Megatranslations Ltd, translation of text from the original Russian to English: Dmitry Borodin

Editor of the translated text: Helen Borrie

Copyright © 2017 Firebird Project and all contributing authors, under the [Public Documentation License Version 1.0](#).

Please refer to the [License Notice](#) .

Abstract

...

Table of Contents

1. About the Firebird Developer's Guide	1
Subject Matter	1
About the Author	1
Translation	1
... and More Translation	1
Acknowledgments	2
2. Developing Firebird Applications in Delphi	4
Starting a Project	4
TFDConnection Component	4
Path to the Client Library	4
Developing for Embedded Applications	5
Connection parameters	5
Connection Parameters in a Configuration File	7
Connecting to the database	8
Working with Transactions	9
TFDTransaction Component	9
Datasets	12
TFDQuery Component	12
TFDUpdateSQL component	15
TFDCommand component	18
Types of Command	18
Creating the Primary Modules	19
The Read-only Transaction	21
The Read/Write Transaction	22
Configuring the Customer Module for Editing	23
Implementing the Customer Module	24
Using a RETURNING Clause to Acquire an Autoinc Value	27
Creating a Secondary Module	27
The Transactions for Invoice Data	28
A Filter for the Data	28
Configuring the Module	29
Doing the Work	31
The Invoice Details	35
The Result	40
Conclusion	40
Downloads	41
3. Developing Firebird Applications with Microsoft Entity Framework	42
Methods of Interacting with a Database	42
Setting Up for Firebird in Visual Studio 2015	42
The Installation Process	43
Creating a Project	47
Adding Packages to the Project	47
Creating an Entity Data Model (EDM)	48
The EDM Files	56
Creating a User Interface	59
Getting a Context	59
Working with Data	61
LINQ Extension Methods	61

Secondary Modules	67
Filtering Data	68
Loading the Invoice Data	68
Showing the Invoice Lines	72
Working with Stored Procedures	73
Showing Products for Selection	76
Working with Transactions	77
The Result	80
Source Code	80
Appendix A: License notice	81

List of Figures

2.1. TFDCConnection property editor	6
2.2. TFDUpdateSQL property editor	16
2.3. TFDUpdateSQL SQL command editor	17
2.4. dCustomers datamodule	20
2.5. Customers form, initial view	21
2.6. The Invoice form tab	29
2.7. The Invoice data module tab	30
2.8. The Customer input form	33
2.9. Screenshot of the sample application	40
3.1. Choose data source for testing installation	45
3.2. Locate a database	46
3.3. Test and confirm the connection	46
3.4. Solution Explorer—>select NuGet packages	47
3.5. Select and install packages from NuGet catalogue	48
3.6. Solution Explorer - Add—>New Item	49
3.7. Add New Item wizard - select ADO.NET Entity Data Model	50
3.8. Add New Item wizard - select 'Code First from database'	50
3.9. Add New Item wizard - choose Connection	51
3.10. Add Connection wizard - Connection properties	52
3.11. Add Connection wizard - Advanced connection properties	53
3.12. EDM wizard - connection string storage	54
3.13. EDM wizard - select tables and views	56
3.14. A form for the Customer entity	59
3.15. Customer edit form	66
3.16. Invoice form	68
3.17. Product form	76
3.18. The result of the Entity Framework project	80

List of Tables

2.1. TFDConnection component main properties	6
2.2. TFDTransaction component main properties	9
2.3. TFDQuery component main properties	12
2.4. TFDUpdateSQL component main properties	17
2.5. TFDCommand component main properties	18

Chapter 1

About the Firebird Developer's Guide

for Firebird 3.0

Blurb...

Subject Matter

This volume consists of chapters that walk through the development of a simple application for several language platforms, notably Delphi, Microsoft Entity Framework and MVC.NET (“Model-View-Controller”) for web applications, PHP and Java with the Spring framework. It is hoped that the work will grow in time, with contributions from authors using other stacks with Firebird.

About the Author

Denis Simonov....

Translation

Development of the original Russian version was sponsored by IBSurgeon and Moscow Exchange Bank. A crowd-funding campaign was launched by the Firebird Foundation in 2017 to fund the translation into English to provide this document as the foundation for translation by Firebird Project document writers into other languages.

The campaign succeeded in raising enough to get the process under way.

... and More Translation

Once the DocBook source appears in GitHub, we hope the trusty translators will start making versions in German, Japanese, Italian, French, Portuguese, Spanish, Czech. Certainly, we never have enough translators so please, you Firebirders who have English as a second language, do consider translating some chapters into your first language.

Acknowledgments

We acknowledge these contributions of sponsors and donors with gratitude and thank you all for stepping up.

Sponsors and Other Donors

Sponsors of the Russian Language version of this Guide

Moscow Exchange (Russia)

Moscow Exchange is the largest exchange holding in Russia and Eastern Europe, founded on December 19, 2011, through the consolidation of the MICEX (founded in 1992) and RTS (founded in 1995) exchange groups. Moscow Exchange ranks among the world's top 20 exchanges by trading in bonds and by the total capitalization of shares traded, as well as among the 10 largest exchange platforms for trading derivatives.

IBSurgeon (ibase.ru) (Russia)

Technical support and developer of administrator tools for the Firebird DBMS.

Sponsors of the Translation Project

Firebird Developers' Day contributors, 2017 (Brazil)

Syntess Software BV (Netherlands)

Other Donors

Listed below are the names of companies and individuals whose cash contributions covered the costs for translation into English, editing of the raw, translated text and conversion of the whole into the Firebird Project's standard DocBook 4 documentation source format.

Peter Lee (Australia)

Thomas M. Conrad (U.S.A.)

Doug Chamberlin (U.S.A.)

Francis Moore (U.K.)

Juan Antonio Mendoza Gil

Robert Firl

Aparecido Silva

Deon van Niekerk

Martin Mutiku

Myles Wakeham (U.S.A.)

Pal Lillejord

Chong Ray

Jean-Marc Couret

Guiseppe Minutillo (Italy)

Chris Mathews (U.S.A.)

James Batson

Paolo Sciarrini (Italy)

Arkadiusz Wolanski (Poland)

Michele Denys

Marknadsinformation i Sverige AB (Sweden)

Transdata GmbH (Germany)

Francis Mullan (South Africa)

Laurent Guétin (Burkina Faso)

Massimiliano Coros

Roland van Morckhoven (Netherlands)

Andrew Kipcharsky (Russian Federation)

Hartmuth Prüfer (Germany)

Martin Köditz (Germany)

Mark Rotteveel (Netherlands)

Roknic Dusan

Solucionalia Consultores Auditores, S.L. (Spain)

Gabor Boros

Artur Henneberg

Ivan Cruz

Kjell Rilbe (Sweden)

Antonis Tsourinakis (Greece)

Gerdus van Zyl (South Africa)

Michael Trowe

Ralf Stegemann
Alessandro Marcellini
Alexander K. Bowie
Cserna Zsombor
Juergen Bachsteffel (Germany)
Michele Giordano

Shaymon Gracia Campos
Juan Carlos Ramirez
Alberto Fornes Llodra
Jozo Leko
Jose Antonio Amate Belchi
Vasily Vasilov

Chapter 2

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 in 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 *DriverName* property. 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 `system32`, 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 msucr80.dll (for Firebird 2.5) as well as msvcp100.dll and msucr100.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, for Firebird 2.5 you should replace fbclient.dll with fbembed.dll. Make 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.

Figure 2.1. TFDConnection property editor

FireDAC Connection Editor - [FDConnection]

Select driver or select connection definition name to override, then setup parameters

Definition Options Info SQL Script

Driver ID: FB

Connection Definition Name:

Test Wizard Revert To Defaults Help

Parameter	Value	Default
DriverID	FB	FB
Pooled	False	False
Database	D:\examples.fdb	
User_Name	sysdba	
Password		
MonitorBy		
OSAuthent	No	
Protocol	TCPIP	Local
Server	localhost	
SQLDialect	3	3
RoleName		
CharacterSet	UTF8	
ExtendedMetadata	False	False
CreateDatabase	No	No
PageSize	4096	4096
IBAdvanced		

OK Cancel

Table 2.1. TFDConnection component main properties

Property	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

Property	Purpose
Protocol	<p>Connection protocol. Possible values:</p> <ul style="list-style-type: none"> • 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, e.g., 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 Update-Transaction 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 FDCConnection.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.

The *TFDTransaction* component is used to manage transactions explicitly.

TFDTransaction Component

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

Table 2.2. TFDTransaction component main properties

Property	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 the Autocommit 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	

Property	Purpose
	The action that will be performed when the connection is closed while the transaction is active. The default value is <code>xdCommit</code> —the transaction will be committed. See note (2) below for details of the other options.
<code>Options.EnableNested</code>	Controls nested transactions. The default value is <code>True</code> . Firebird does not support nested transactions as such but FireDac can emulate them using savepoints. For more details, see note(3) below.
<code>Options.Isolation</code>	Specifies the transaction isolation level. It is the most important transaction property. The default value is <code>xiReadCommitted</code> . The other values that Firebird supports are <code>xiSnapshot</code> and <code>xiUnspecified</code> ; also <code>xiSerializable</code> , to some degree. For more details about the available isolation levels, see note (4) below.
<code>Options.Params</code>	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.
<code>Options.ReadOnly</code>	Indicates whether it is a read-only transaction. The default value is <code>False</code> . Setting it to <code>True</code> disables any write activity. Long-running read-only transactions in <code>READ COMMITTED</code> 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`

If the value of *AutoCommit* is set to `True`, FireDAC behaves as follows:

- Starts a transaction (if required) before each SQL command and ends the transaction after the SQL command completes execution
- If the command is successfully executed, the transaction will be ended by `COMMIT`. Otherwise, it will be ended by `ROLLBACK`.
- 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 with Firebird 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 to wait for a conflict to resolve
- `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` are set 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 just variants, 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 *TIBDataSet* component, *TFDQuery* component does not have the properties *RefreshSQL*, *InsertSQL*, *UpdateSQL* and *DeleteSQL*. Instead, a separate *TFDUpdateSQL* object specifies the statement for dataset modifications and the dataset component carries a reference to that component in its *UpdateObject* property.

RequestLive Property

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.

TFDQuery Component

Table 2.3. TFDQuery component main properties

Property	Purpose
Connection	Reference to the <i>FDConnection</i> object
MasterSource	If the dataset is to be used as detail to a master dataset, this property refers to the data source (<i>TDataSource</i>) 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 <i>FDUpdateSQL</i> object providing for the dataset to be editable when the SELECT query does not meet the requirements for automatic generation of modification queries with <i>UpdateOptions.RequestLive=True</i> .
UpdateTransaction	

Property	Purpose
	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 NOT NULL fields. If you keep it True and a field with the 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.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 <i>gpDeferred</i> , 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 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

Property	Purpose
	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 by calling 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 the 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
- *gpDeferred*—the 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*—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. For updating a dataset that is delivered into an *FDStoredProc* object, use of the *TFDUpdateSQL* 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.

TFDUpdateSQL Properties

To specify the SQL DML statements at design time, double-click on the *TFDUpdateSQL* component in your data module to open the property editor.

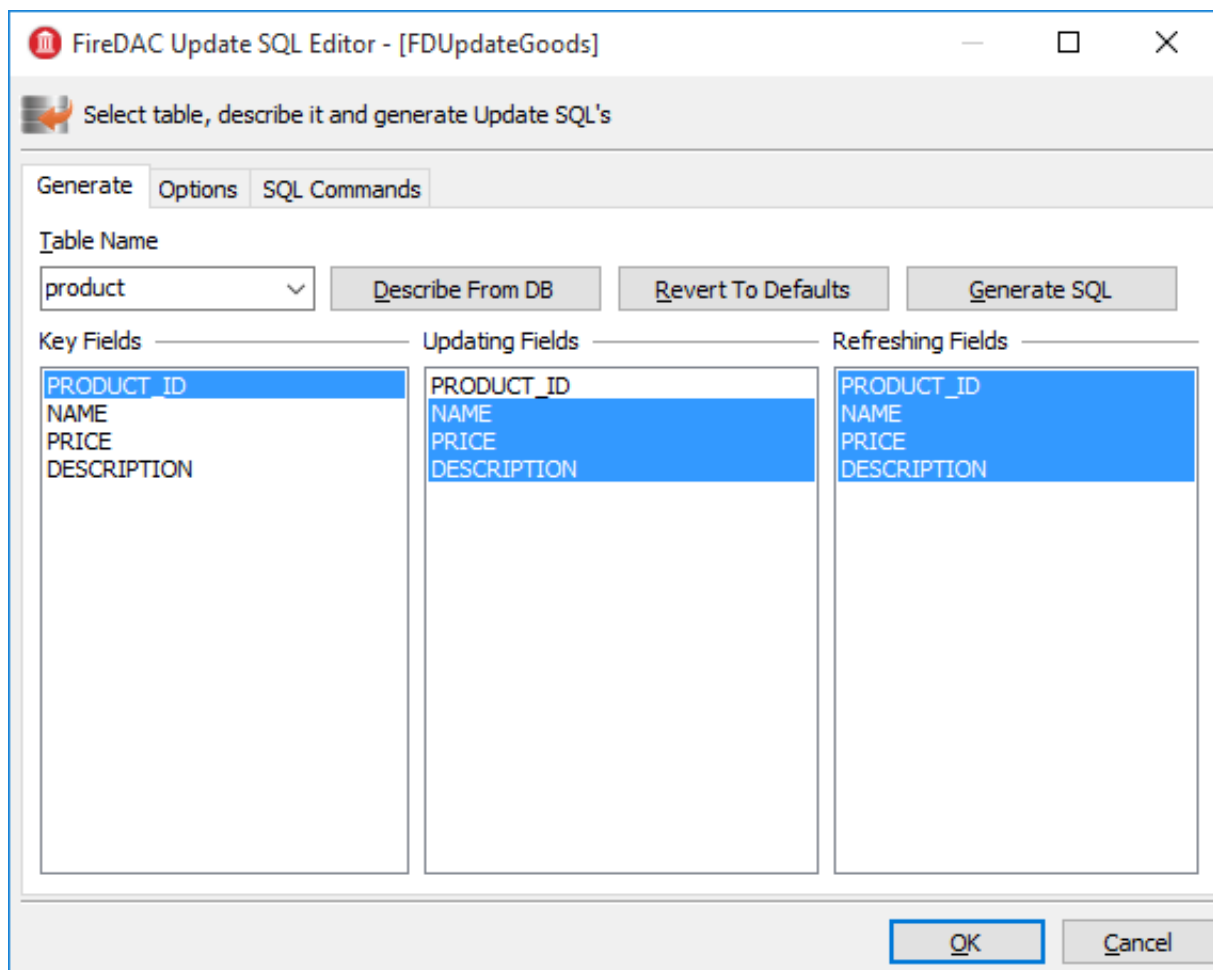
Important

Each component has its own design-time property editor. For multiple data-aware 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*) for each one.

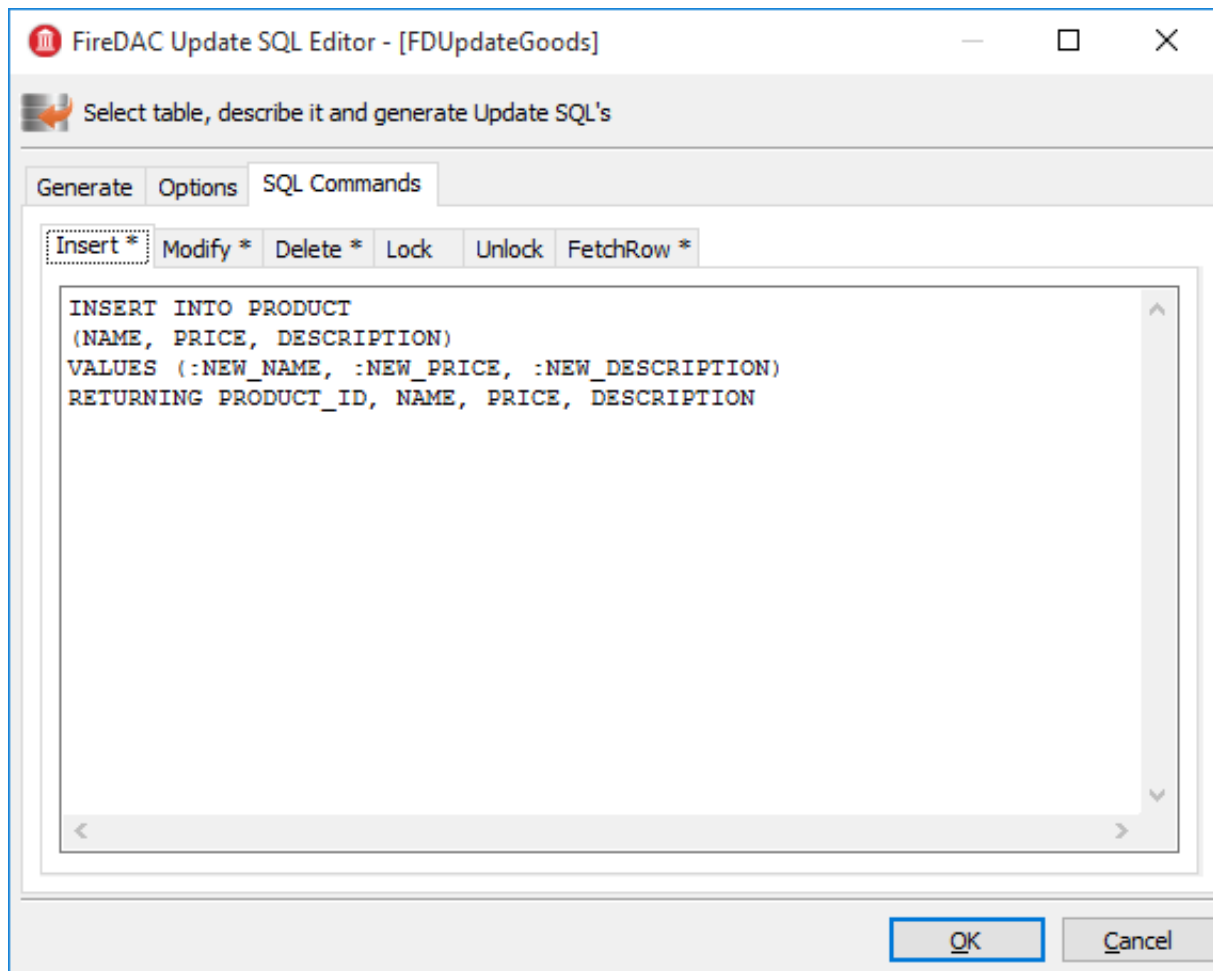
Design-time settings could interfere with the way the application is intended to work. For instance, the user is supposed to log in to the program using his username, but the *FDConnection* object connects to the database as *SYSDBA*.

It is advisable to check the *Connected* property of the *FDConnection* object and reset it each time you use the data-aware editors. *AutoStart* will have to be enabled and disabled for a read-only transaction as well.

Figure 2.2. TFDUpdateSQL property editor



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.

Figure 2.3. TFDUpdateSQL SQL command editor**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, 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 convenience, they are references to the *UpdateOptions* properties of the dataset that has the current *TFDUpdateSQL* specified in its *UpdateObject* property.

Table 2.4. TFDUpdateSQL component main properties

Property	Purpose
Connection	Reference to the TFDConnection component
DeleteSQL	The SQL query for deleting a record

Property	Purpose
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 2.5. TFDCommand component main properties

Property	Purpose
Connection	Reference to the <i>TFDConnection</i> 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:

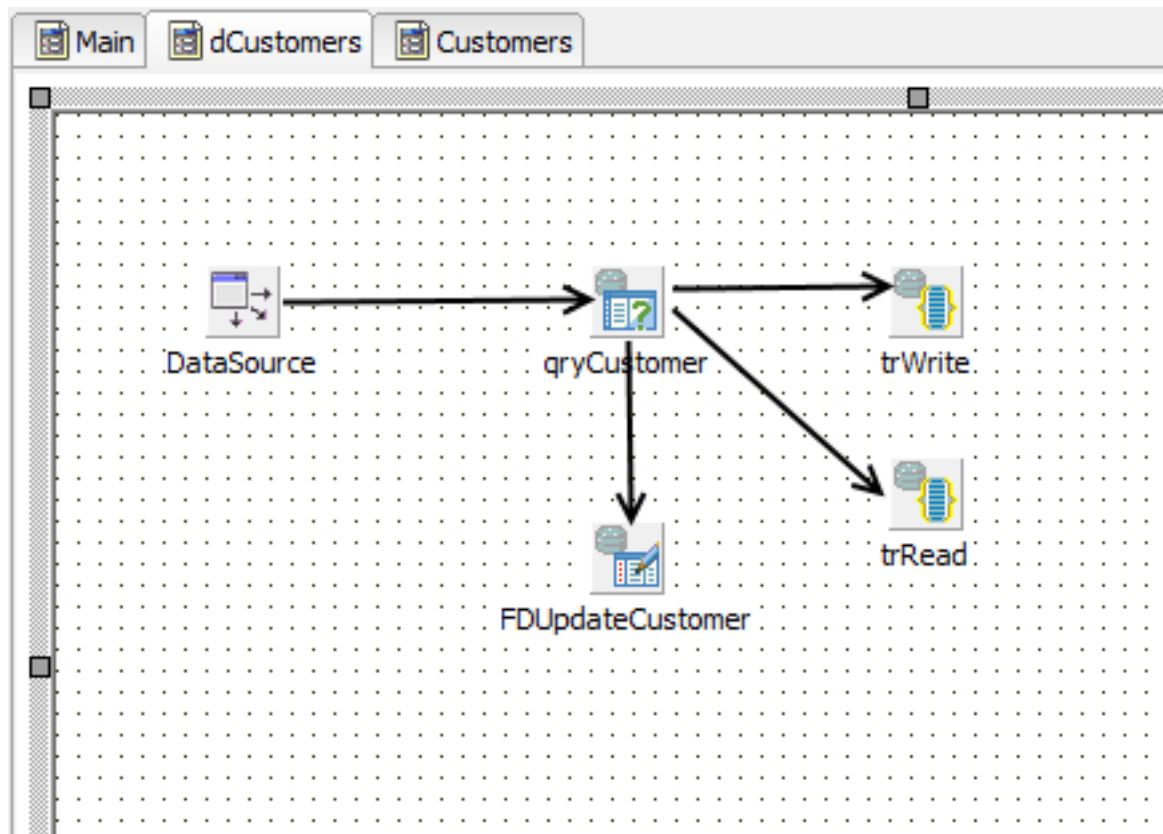
- *skUnknown*—unknown. Tells the internal parser to determine the command type automatically from its analysis of the text of the command
- *skStartTransaction*—a command for starting a transaction

- `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 `EXECUTE BLOCK` command
- `skStoredProc`—a stored procedure call
- `skStoredProcNoCrs`—a call to a stored procedure that does not return a cursor
- `skStoredProcWithCrs`—a call to a stored procedure that returns a cursor

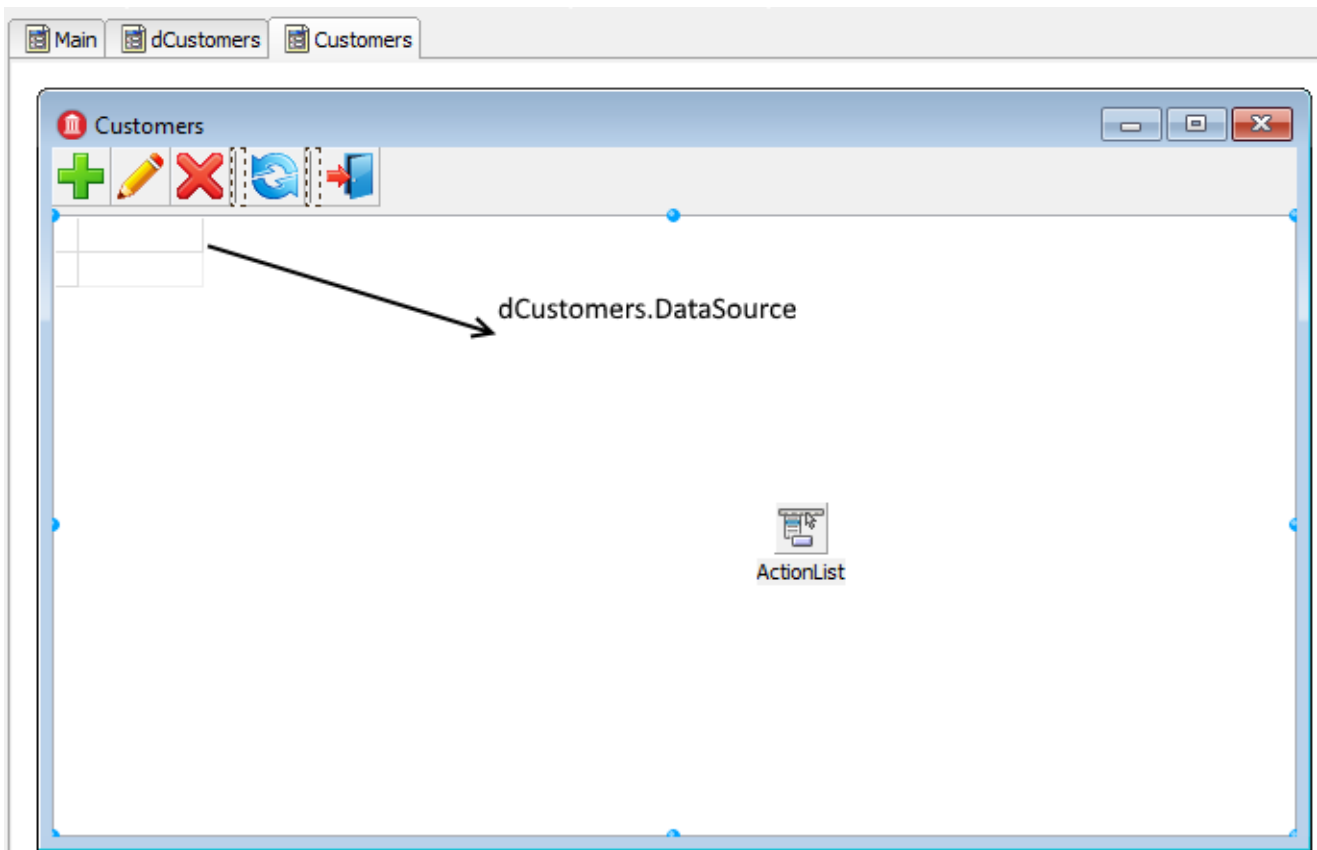
Creating the Primary Modules

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:

Figure 2.4. 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.

Figure 2.5. Customers form, initial view

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 *trWrite* transaction 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 *trRead* read transaction 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 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 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 should 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/UP-

DATE/DELETE query, it is advisable to use SNAPSHOT. The reason is that READ COMMITTED isolation 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, SNAPSHOT isolation 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 example, 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, in the topic [Using a RETURNING Clause to Acquire an Autoinc Value](#).

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 the 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, just 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 implemented. 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;
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;
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 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.

To acquire 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, their quantities, prices, etc. It is convenient to have two grids for such documents: the main one (master) showing the data 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 one 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

Figure 2.6. The Invoice form tab

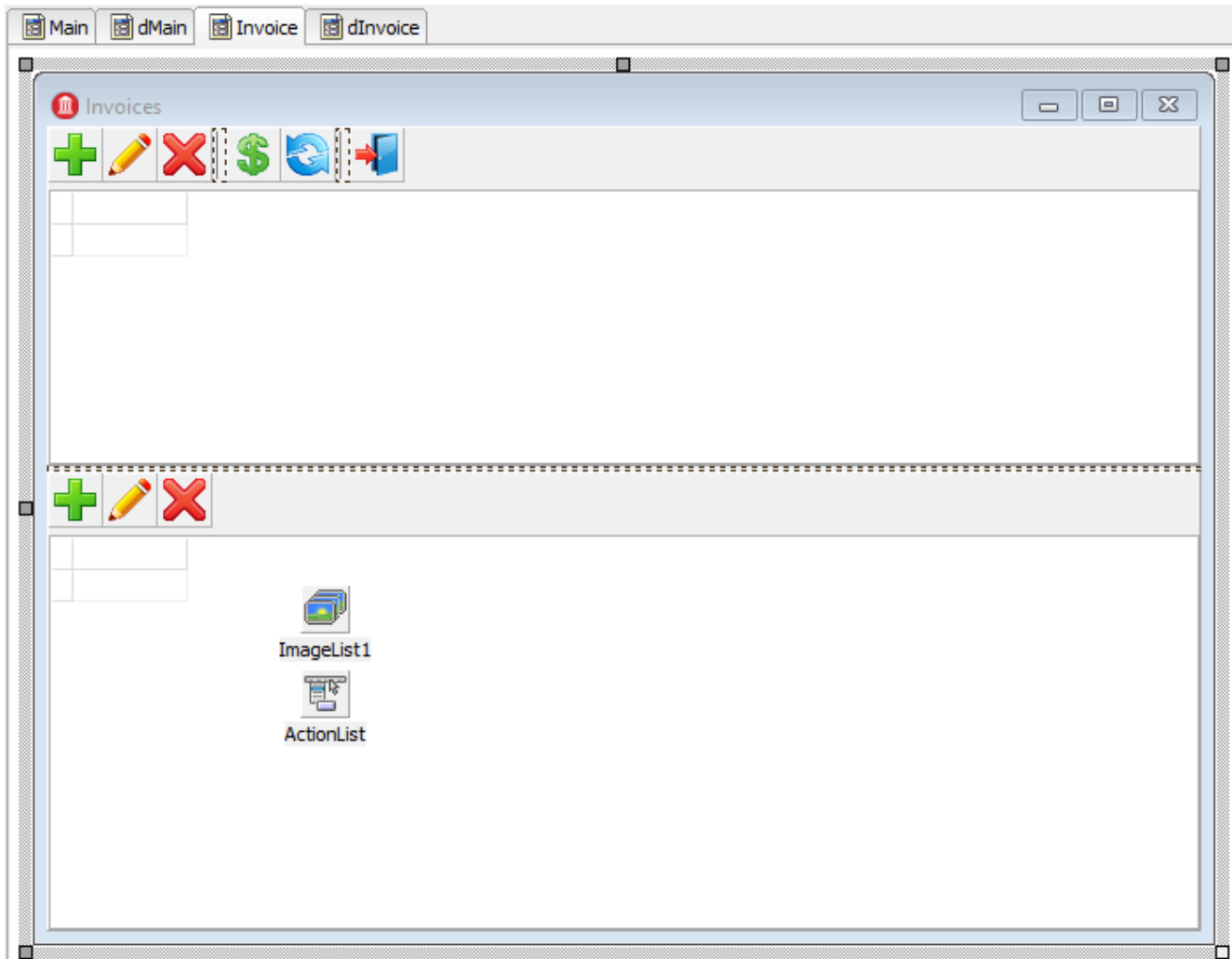
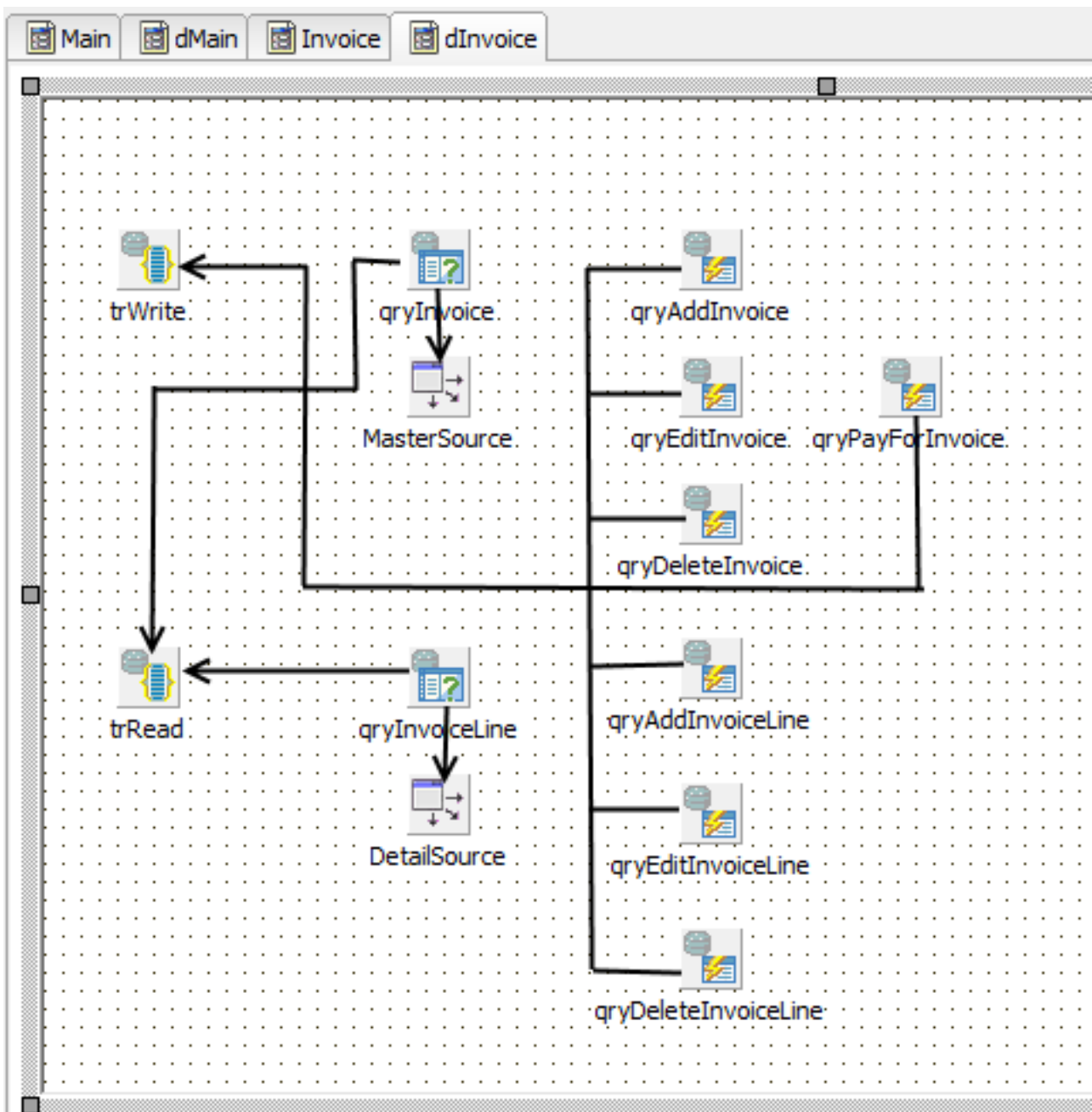


Figure 2.7. The Invoice data module tab

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 paid
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 *TButtonEdit* 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 *TDBLookup-Combobox*, 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.

Figure 2.8. The Customer input form

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 *TButtonEdit* 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 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;
end;

```

```

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;
end;

```

The Invoice Details

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 *TButtonEdit* component again. The code for the on-click handler for the button on the *TButtonEdit* 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;
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 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;
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;

```

```

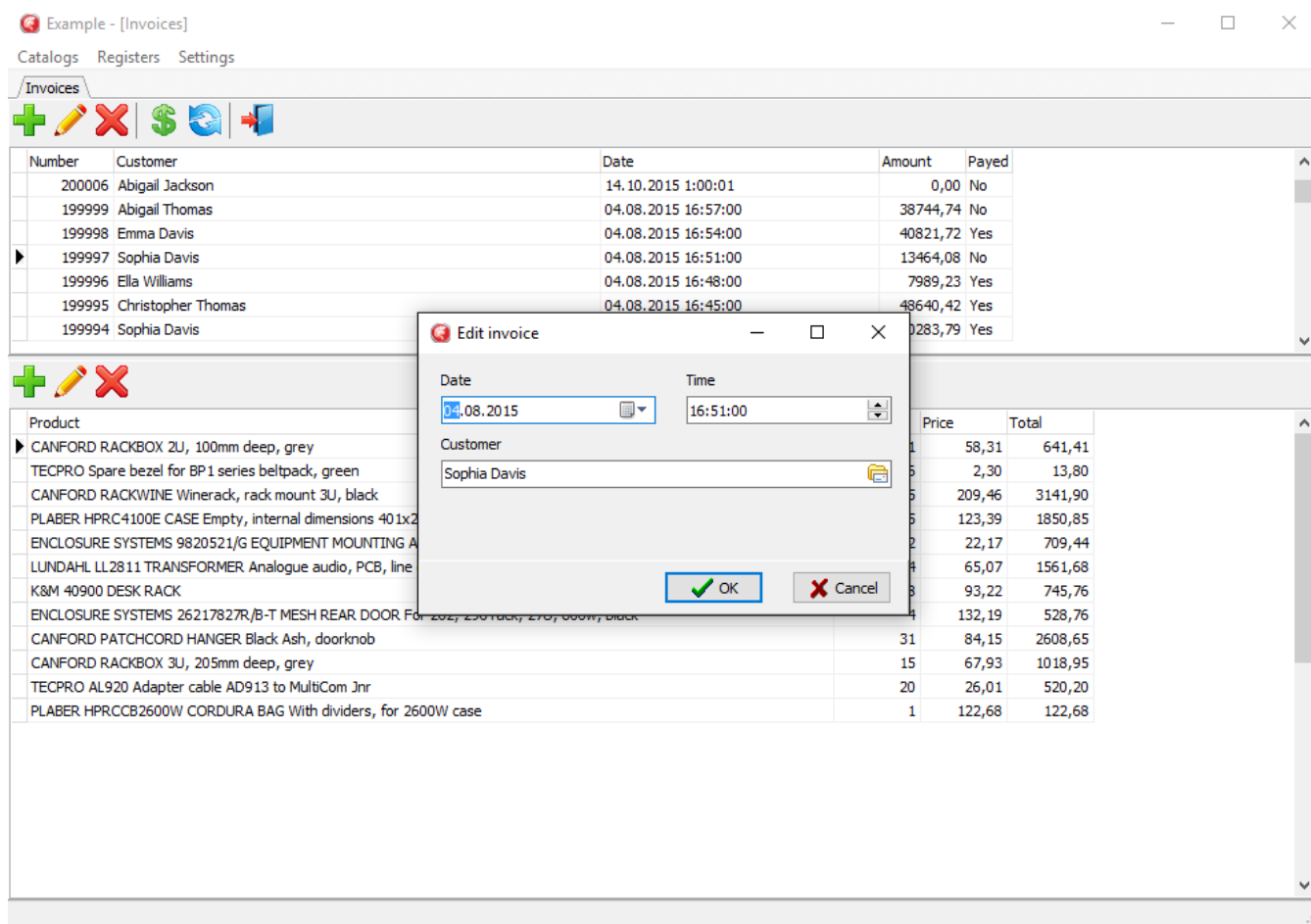
procedure TdmInvoice.EditInvoiceLine(AQuantity: Integer);
begin
  // We do everything in a short transaction
  trWrite.StartTransaction;
  try
    qryEditInvoiceLine.ParamByName('INVOICE_LINE_ID').AsInteger :=
      InvoiceLine.INVOICE_LINE_ID.Value;
    qryEditInvoiceLine.ParamByName('QUANTITY').AsInteger := 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

Figure 2.9. Screenshot of the sample application



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 free-ware. They include FibPlus, IBOjects, 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.

To avoid holding an uncommitted transaction during an editing session, either use visual components that are not data-aware or use the *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 changes 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* component. 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

.

Downloads

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

Chapter 3

Developing Firebird Applications with Microsoft Entity Framework

This chapter will describe the process of creating applications with a Firebird database using the Microsoft™ Entity Framework™ access components in the Visual Studio 2015™ environment.

ADO.NET Entity Framework (EF) combines an object-oriented data access technology with an object-relational mapping (ORM) solution for the Microsoft .NET Framework. It enables interaction with objects by means of both LINQ in the form of **LINQ to Entities** and with Entity SQL.

Methods of Interacting with a Database

Entity Framework assumes three possible methods for interacting with a database:

Database first:

Entity Framework creates a set of classes that reflect the model of an existing database.

Model first:

the developer creates a database model that Entity Framework later uses to create an actual database on the server.

Code first:

the developer creates a class for the model of the data that will be stored in a database and then Entity Framework uses this model to generate the database and its tables

Our sample application will use the **Code first** approach, but you could use one of the others just as easily.

Note

As we already have a database, we will just write the code that would result in creating that database.

Setting Up for Firebird in Visual Studio 2015

To prepare for working with Firebird, you will need to install the following:

- FirebirdSql.Data.FirebirdClient.dll
- the Firebird DDEX Provider for Visual Studio

- EntityFramework.Firebird.dll

There is nothing difficult in installing the first two. They are currently distributed and installed into a project by means of the [NuGet package manager](#). The DDEX Provider library, designed for operating Visual Studio wizards, is not so easy to install and may take more time and effort.

Efforts have been made to automate the installation process and include all components in a [single installer package](#). However, you might need to install all of the components manually under some conditions. If so, you can download the following:

- [FirebirdSql.Data.FirebirdClient-4.10.0.0.msi](#)
- [EntityFramework.Firebird-4.10.0.0-NET45.7z](#)
- [DDEXProvider-3.0.2.0.7z](#)
- [DDEXProvider-3.0.2.0-src.7z](#)

The Installation Process

Important!

Because the installation involves operations in protected directories, you will need administrator privileges to do it.

Steps

1. Install FirebirdSql.Data.FirebirdClient-4.10.0.0.msi
2. Unpack EntityFramework.Firebird-4.10.0.0-NET45.7z to the folder with the installed Firebird client. In my case, it is the folder `c:\Program Files (x86)\FirebirdClient\`.
3. You need to install a Firebird build into the GAC. For your convenience, specify the path to the *gacutil* utility for .NET Framework 4.5 in the environment variable `%PATH%`. In my case, the path is `c:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6.1 Tools\`
4. Run the command shell `cmd.exe` as administrator and go to the directory with the installed client, e.g.,

```
chdir "c:\Program Files (x86)\FirebirdClient"
```

5. Now make sure that `FirebirdSql.Data.FirebirdClient` is installed into the GAC by typing the following command:

```
gacutil /l FirebirdSql.Data.FirebirdClient
```

If `FirebirdSql.Data.FirebirdClient` has not been installed into the GAC, use the following command to do it now:

```
gacutil /i FirebirdSql.Data.FirebirdClient.dll
```

6. Now install EntityFramework.Firebird into the GAC

```
gacutil /i EntityFramework.Firebird.dll
```

7. Unpack DDEXProvider-3.0.2.0.7z to a directory convenient for you. Mine was unpacked to `c:\Program Files (x86)\FirebirdDDEX\`.
8. Unpack the contents of the `/reg_files/Vs2015` subdirectory from the archive `DDEXProvider-3.0.2.0-src.7z` there as well.

Author's remark

For some strange reason these files are absent from the archive with the compiled dll libraries, but they are present in the source code archive.

9. Open the `FirebirdDDEXProvider64.reg` file in Notepad. Find the line that contains `%path%` and change it to the full path to the file `FirebirdSql.VisualStudio.DataTools.dll`, e.g.,

```
"CodeBase"="c:\\Program Files (x86)\\FirebirdDDEX\\FirebirdSql.VisualStudio.DataTools.d
```

10. Save this Registry file and run it. Click YES to the question about adding the information to the Registry.
11. Now you need to edit the `machine.config` file. In my installation, the path is as follows:

```
C:\Windows\Microsoft.NET\Framework\v4.0.30319\Config
```

Open this file in Notepad. Find the following section:

```
<system.data>
  <DbProviderFactories>
```

Add the following lines to this section:

```
<add name="FirebirdClient Data Provider"
      invariant="FirebirdSql.Data.FirebirdClient"
      description=".Net Framework Data Provider for Firebird"
      type="FirebirdSql.Data.FirebirdClient.FirebirdClientFactory,
           FirebirdSql.Data.FirebirdClient, Version=4.10.0.0, Culture=neutral,
           PublicKeyToken=3750abcc3150b00c" />
```

Note

The settings we have configured here are valid for version 4.10.0.

Do the same for `machine.config` located at `c:\Windows\Microsoft.NET\Framework64\v4.0.30319\Config\`

This completes the installation.

Testing the Installation

To make sure that everything has been installed successfully, start Visual Studio 2015. Find the Server Explorer and try to connect to an existing Firebird database.

Figure 3.1. Choose data source for testing installation

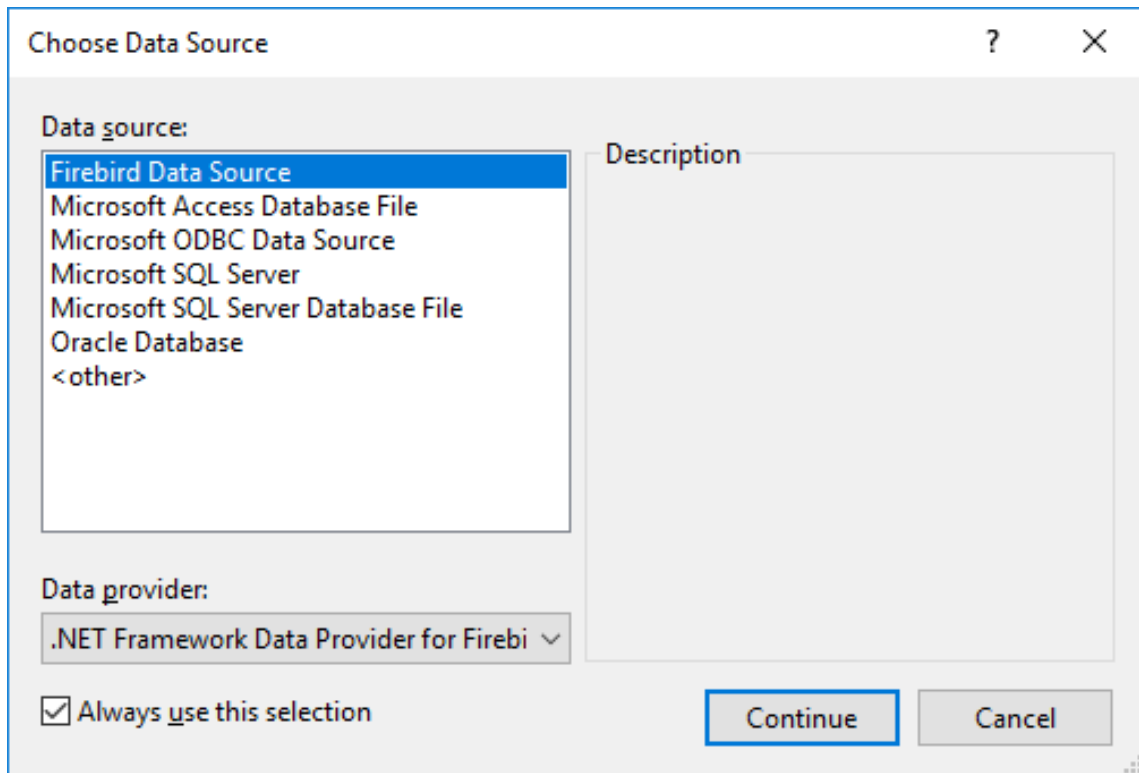


Figure 3.2. Locate a database

Add Connection ? X

Enter information to connect to the selected data source or click "Change" to choose a different data source and/or provider.

Data source:
Firebird Data Source (.NET Framework Data Provider for Firebird) Change...

Data Source	Data Source Port	Dialect	Charset
localhost	3050	3	UTF8

Database
examples ...

Login

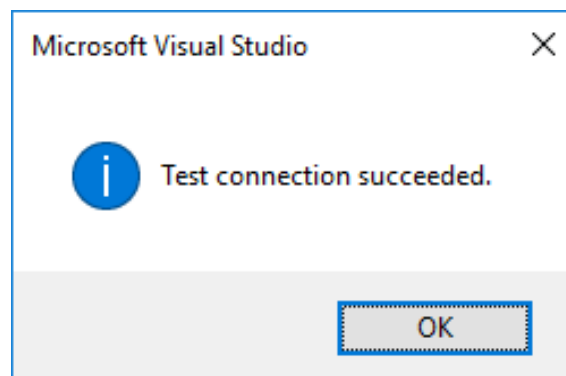
User: SYSDBA
Password: *****
Role:

Connection Settings

Server Type: Standalone Server

Advanced...

Test Connection OK Cancel

Figure 3.3. Test and confirm the connection

Creating a Project

For our example in this chapter, we will create a *Windows Forms* application. Other types of applications differ from it, but the principles of working with Firebird via Entity Framework remain the same.

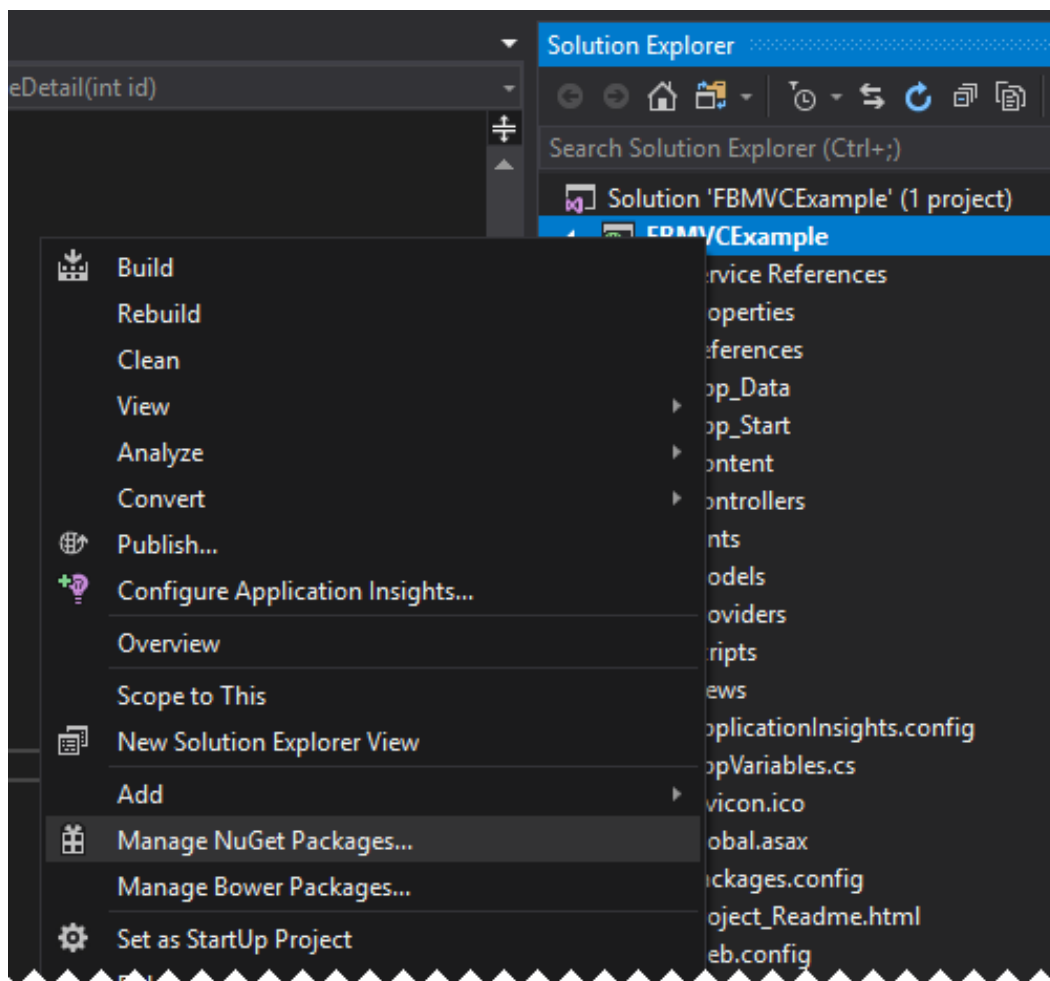
Adding Packages to the Project

The first task after creating a Windows Forms project is to add the following packages to it, using the NuGet package manager:

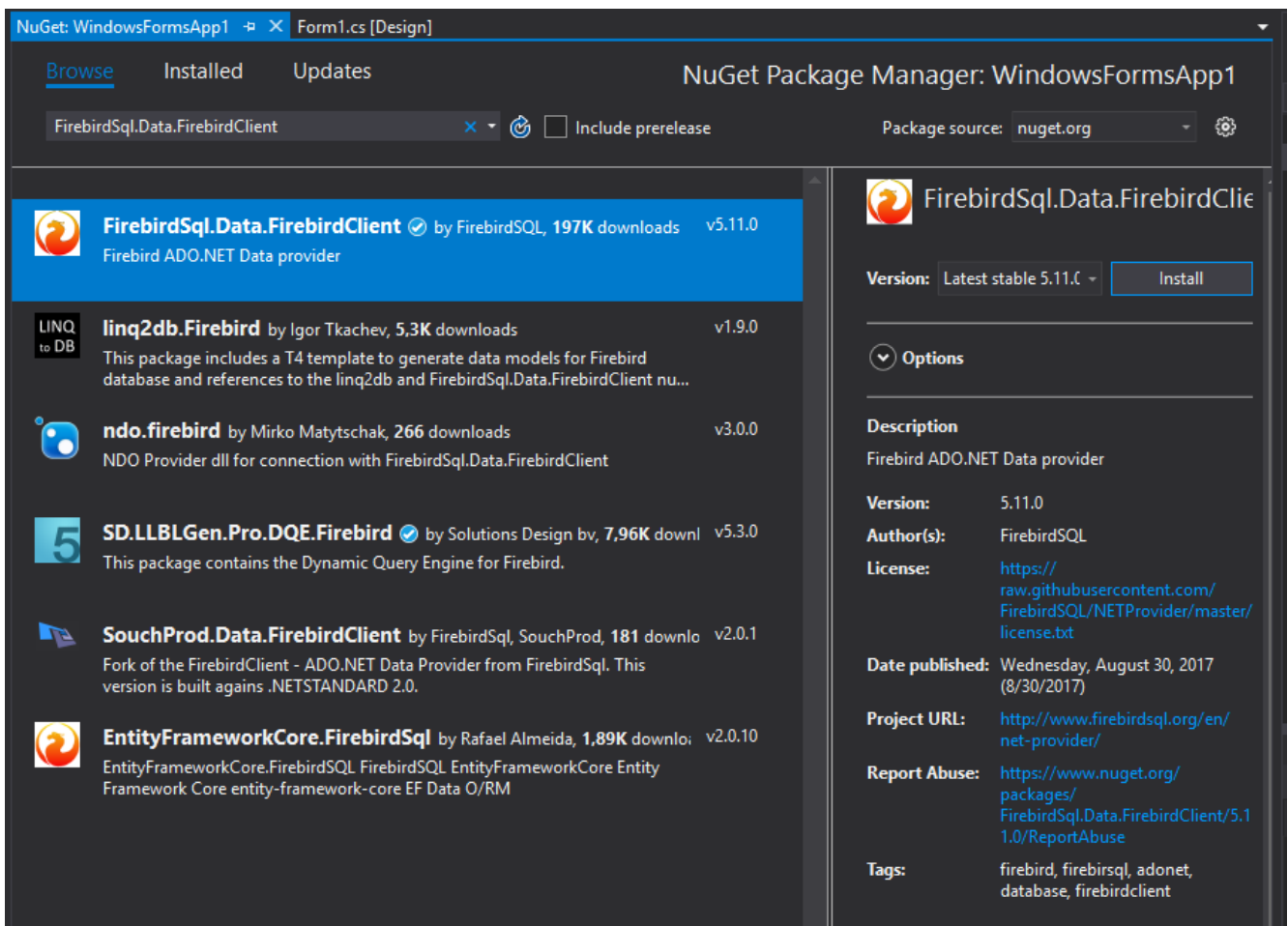
- `FirebirdSql.Data.FirebirdClient`
- `EntityFramework`
- `EntityFramework.Firebird`

Right-click the project name in Solution Explorer and select *Manage NuGet Packages* from the drop-down list.

Figure 3.4. Solution Explorer—>select NuGet packages



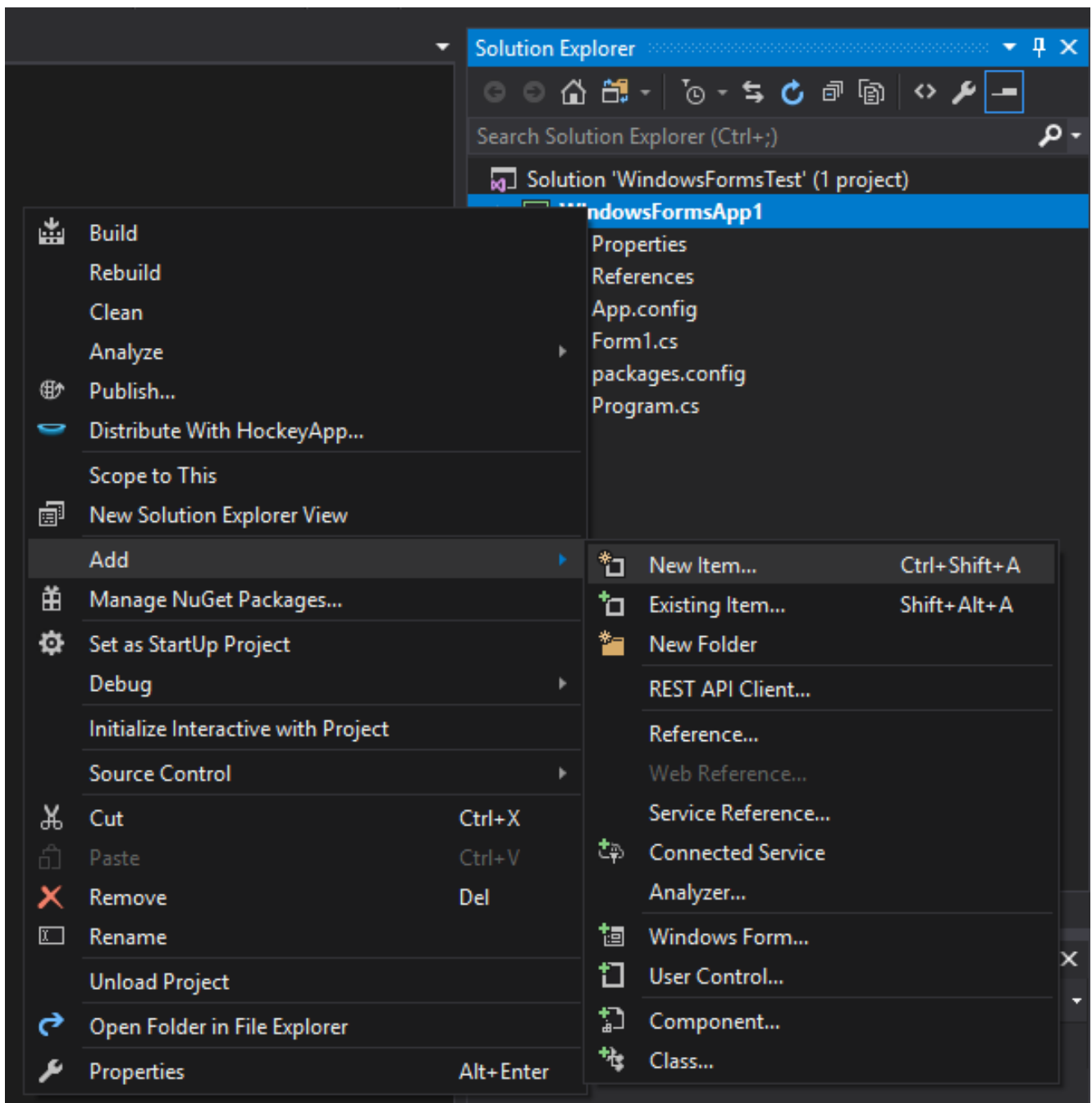
Find the packages listed above in the Nuget catalogue and install them in the package manager.

Figure 3.5. Select and install packages from NuGet catalogue

Creating an Entity Data Model (EDM)

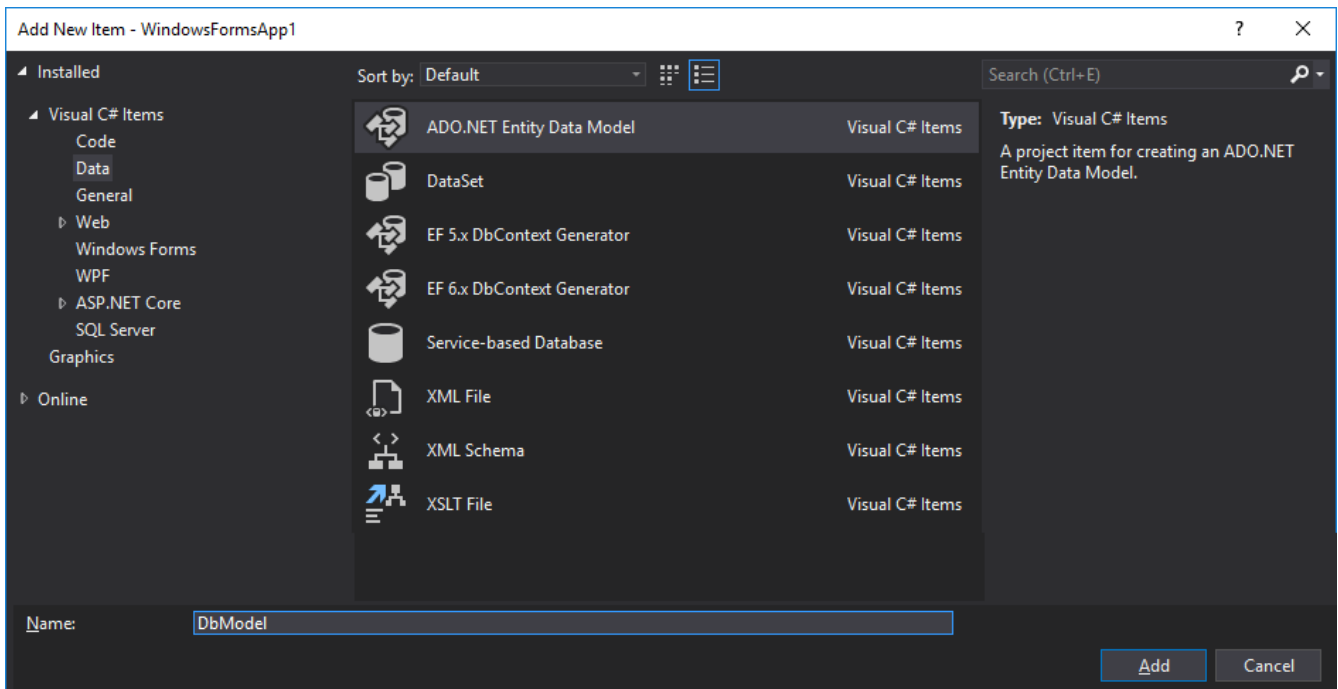
In our application, we will use the *Code First* approach.

To create an EDM, right-click the project name in Solution Explorer and select *Add—>New Item* from the menu.

Figure 3.6. Solution Explorer - Add—>New Item

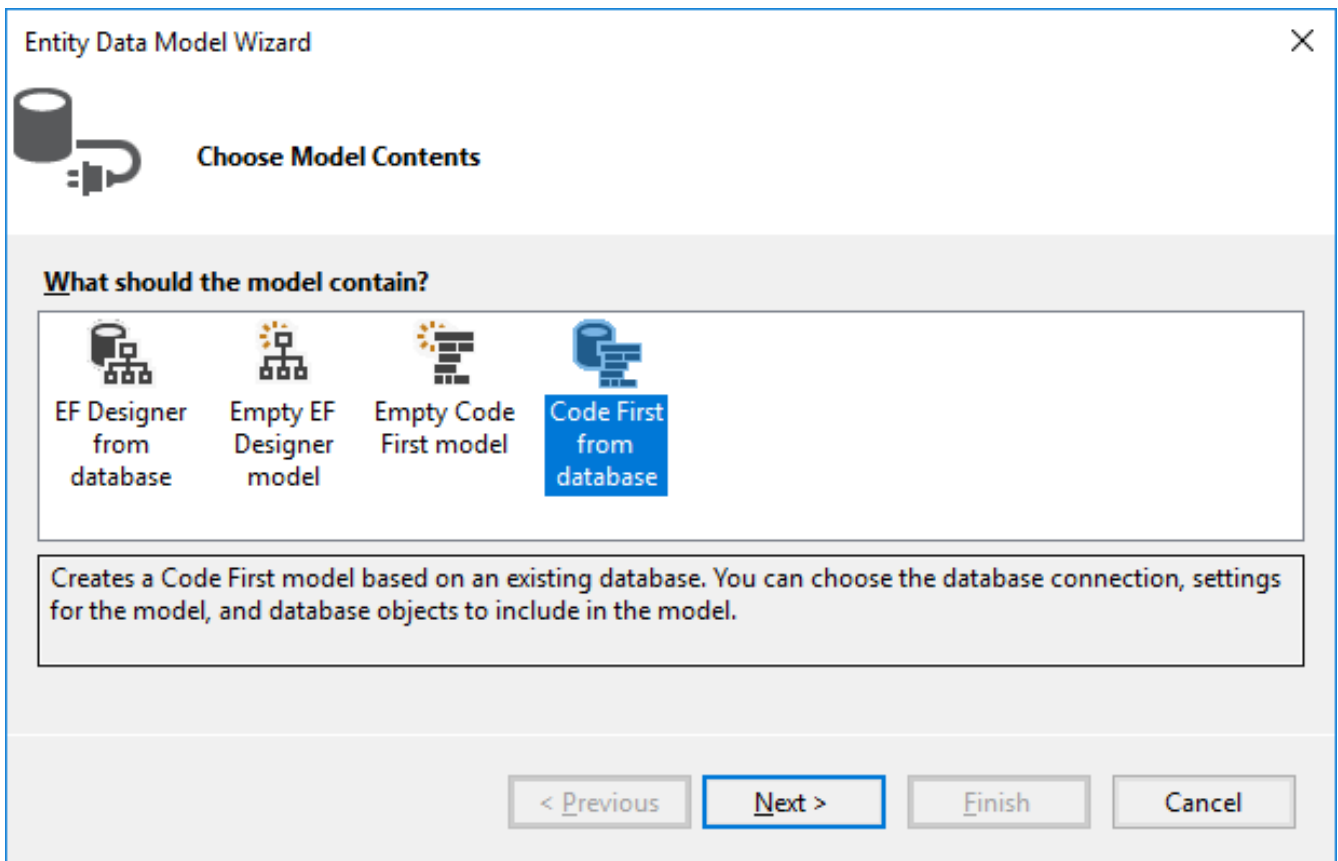
Next, in the *Add New Item* wizard, select *ADO.NET Entity Data Model*.

Figure 3.7. Add New Item wizard - select ADO.NET Entity Data Model



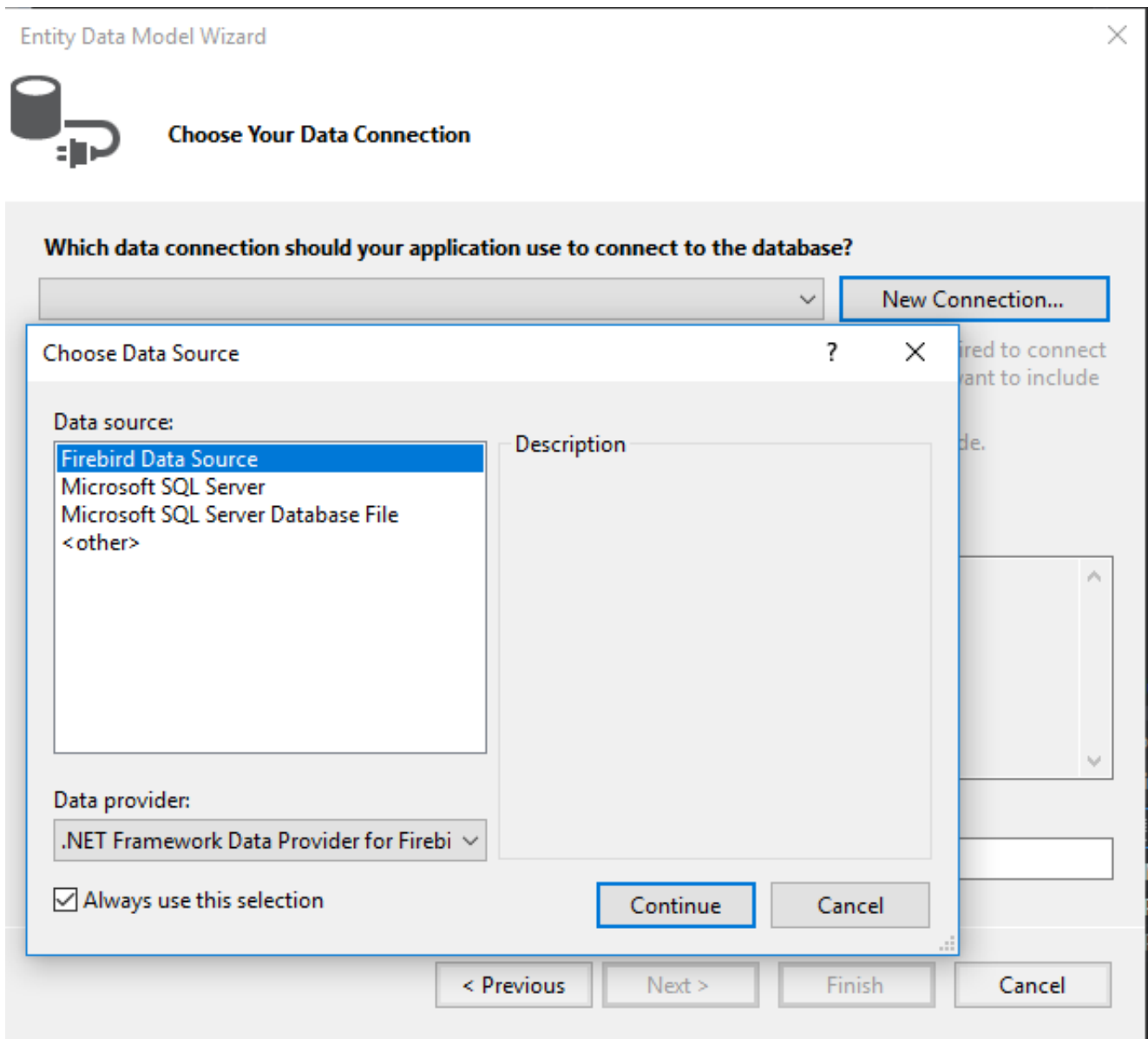
Since we already have a database, we will generate the EDM from the database. Select the icon captioned *Code First from database*.

Figure 3.8. Add New Item wizard - select 'Code First from database'



Now we need to select the connection the model will be created from. If the connection does not exist, it will have to be created.

Figure 3.9. Add New Item wizard - choose Connection



You might need to specify some advanced properties in addition to the main connection properties. You might want to set the transaction isolation, for example, to a level different from the default *Read Committed*, or to specify connection pooling, or something else that differs from defaults.

Figure 3.10. Add Connection wizard - Connection properties

The screenshot shows the 'Add Connection' wizard window. The title bar says 'Add Connection' with a help icon and a close button. The main text says: 'Enter information to connect to the selected data source or click "Change" to choose a different data source and/or provider.'

Data source:

Firebird Data Source (.NET Framework Data Provider for Firebird) [Change...]

Data Source **Data Source Port** **Dialect** **Charset**

localhost 3050 3 UTF8

Database

examples [...]

Login

User: SYSDBA

Password: *****

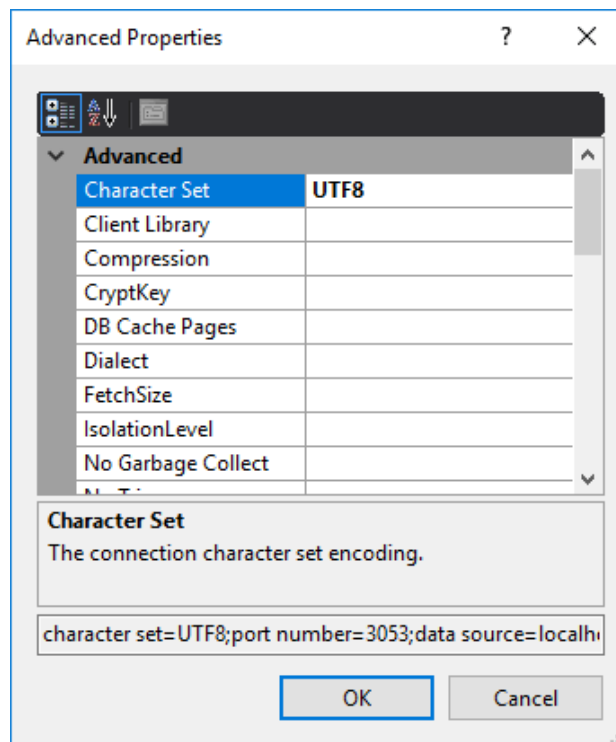
Role:

Connection Settings

Server Type: Standalone Server

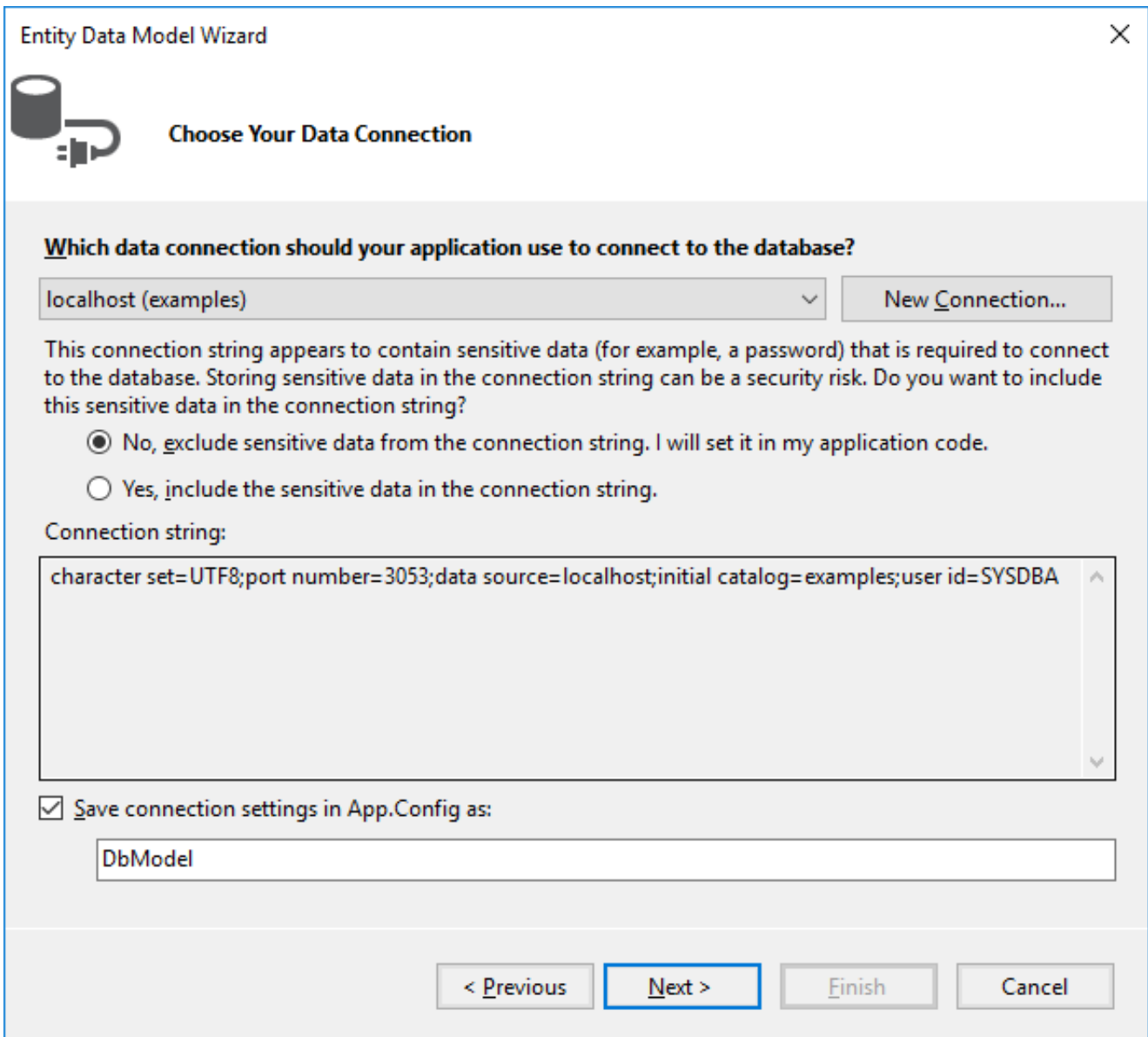
[Advanced...]

[Test Connection] [OK] [Cancel]

Figure 3.11. Add Connection wizard - Advanced connection properties**Tip**

Snapshot is the recommended isolation level because Entity Framework and ADO.NET both use disconnected data access—where each connection and each transaction is active only for a very short time.

Next, the Entity Data Model wizard will ask you how to store the connection string.

Figure 3.12. EDM wizard - connection string storage

The image shows the 'Entity Data Model Wizard' dialog box, specifically the 'Choose Your Data Connection' step. The title bar reads 'Entity Data Model Wizard' with a close button. Below the title bar is a database icon and the text 'Choose Your Data Connection'. The main area contains the question 'Which data connection should your application use to connect to the database?'. Below this is a dropdown menu showing 'localhost (examples)' and a 'New Connection...' button. A text block explains that sensitive data like passwords in connection strings is a security risk and asks if the user wants to include it. There are two radio buttons: 'No, exclude sensitive data from the connection string. I will set it in my application code.' (selected) and 'Yes, include the sensitive data in the connection string.' Below this is a 'Connection string:' label and a text box containing 'character set=UTF8;port number=3053;data source=localhost;initial catalog=examples;user id=SYSDBA'. At the bottom, there is a checked checkbox 'Save connection settings in App.Config as:' followed by a text box containing 'DbModel'. Navigation buttons at the bottom are '< Previous', 'Next >' (highlighted with a blue border), 'Finish', and 'Cancel'.

Entity Data Model Wizard

Choose Your Data Connection

Which data connection should your application use to connect to the database?

localhost (examples) New Connection...

This connection string appears to contain sensitive data (for example, a password) that is required to connect to the database. Storing sensitive data in the connection string can be a security risk. Do you want to include this sensitive data in the connection string?

☒ No, exclude sensitive data from the connection string. I will set it in my application code.

☐ Yes, include the sensitive data in the connection string.

Connection string:

character set=UTF8;port number=3053;data source=localhost;initial catalog=examples;user id=SYSDBA

☒ Save connection settings in App.Config as:

DbModel

< Previous Next > Finish Cancel

For a web application or another three-tier architecture, where all users will be working with the database using a single account, select **Yes**. If your application is going to request authentication for connecting to the database, select **No**.

Tip

It is much more convenient to work with wizards if you select Yes for each property. You can always change the isolation level in the application when it is ready for testing and deployment by just editing the connection string in the `<AppName>.exe.conf` application configuration file. The connection string will be stored in the *connectionStrings* section and will look approximately like this:

```
<add name="DbModel"
  connectionString="character set=UTF8; data source=localhost;
  initial catalog=examples; port number=3050;
  user id=sysdba; dialect=3; isolationlevel=Snapshot;
  pooling=True; password=masterkey;"
  providerName="FirebirdSql.Data.FirebirdClient" />
```

For the configuration file to stop storing the confidential information, just delete this parameter from the connection string: `password=masterkey;`

Firebird 3.0 Notes

Unfortunately, the current ADO.Net provider for Firebird (version 5.9.0.0) does not support network traffic encryption, which is enabled by default in Firebird 3.0 and higher versions. If you want to work with Firebird 3.0, you need to change some settings in `firebird.conf` (or in `databases.conf` for a specific database) to make Firebird to work without trying to use network encryption.

To do it, change the setting from the default

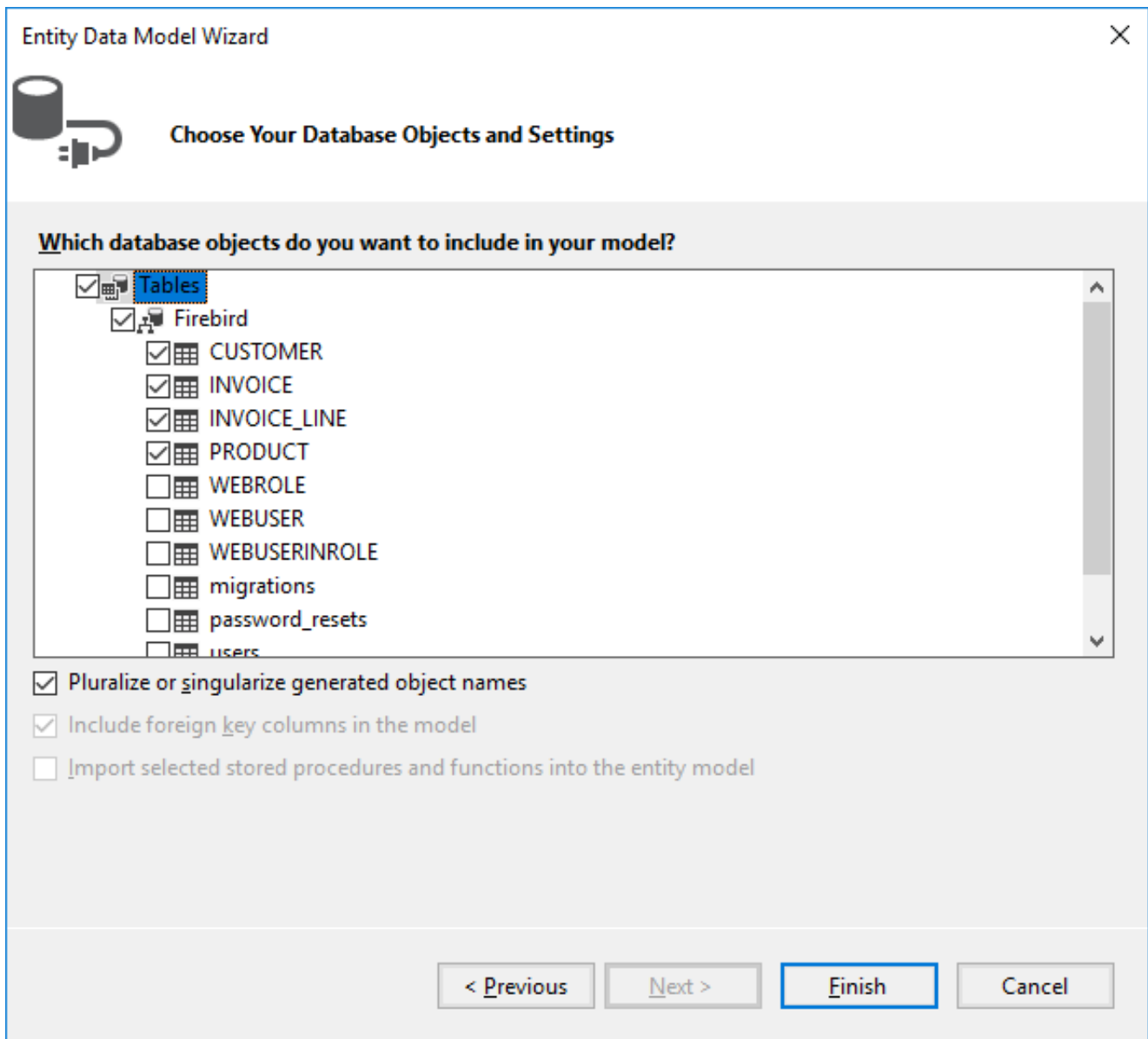
```
# WireCrypt = Enabled
```

to

```
WireCrypt = Disabled
```

making sure to delete the '#' comment marker. Remember that you must restart the server for configuration changes to take effect.

Next, you will be asked which tables and views should be included in the model.

Figure 3.13. EDM wizard - select tables and views

For our project, select the four tables that are checked in the screenshot.

The basic EDM is now ready.

The EDM Files

When the wizard's work is finished, you should have five new files: a model file and four files each describing an entity in the model.

An Entity File

Let's take a look at the generated file describing the INVOICE entity:



```
[Table("Firebird.INVOICE")]
public partial class INVOICE
{
    [System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Usage",
"CA2214:DoNotCallOverridableMethodsInConstructors")]
    public INVOICE()
    {
        INVOICE_LINES = new HashSet<INVOICE_LINE>();
    }

    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.None)]
    public int INVOICE_ID { get; set; }

    public int CUSTOMER_ID { get; set; }

    public DateTime? INVOICE_DATE { get; set; }

    public decimal? TOTAL_SALE { get; set; }

    public short PAYED { get; set; }

    public virtual CUSTOMER CUSTOMER { get; set; }

    [System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Usage",
"CA2227:CollectionPropertiesShouldBeReadOnly")]
    public virtual ICollection<INVOICE_LINE> INVOICE_LINES { get; set; }
}
```

The class contains properties for each field of the INVOICE table. Each of these properties has attributes that describe constraints. You can study the details of the various attributes in the Microsoft document, [Code First Data Annotations](#).

Navigation Properties and “Lazy Loading”

Two navigation properties are generated: CUSTOMER and INVOICE_LINES. The first one contains a reference to the customer entity. The second contains a collection of invoice lines. It is generated because the INVOICE_LINE table has a foreign key to the INVOICE table. Of course, you can remove this property from the INVOICE entity, but it is not really necessary. The CUSTOMER and INVOICE_LINES properties use “lazy loading” which means that loading is not performed until the first access to an object. That way, the loading of related data is avoided unless it is actually needed. Once the data are accessed via the navigation property, they will be loaded from the database automatically.

Important

If lazy loading is in effect, classes that use it must be public and their properties must have the keywords public and virtual.

The DbModel File

Next, we examine the DbModel.cs file that describes the overall model.

```
public partial class DbModel : DbContext
```



```
{
    public DbModel()
        : base("name=DbModel")
    {
    }

    public virtual DbSet<CUSTOMER> CUSTOMERS { get; set; }
    public virtual DbSet<INVOICE> INVOICES { get; set; }
    public virtual DbSet<INVOICE_LINE> INVOICE_LINES { get; set; }
    public virtual DbSet<PRODUCT> PRODUCTS { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Entity<CUSTOMER>()
            .Property(e => e.ZIPCODE)
            .IsFixedLength();

        modelBuilder.Entity<CUSTOMER>()
            .HasMany(e => e.INVOICES)
            .WithRequired(e => e.CUSTOMER)
            .WillCascadeOnDelete(false);

        modelBuilder.Entity<PRODUCT>()
            .HasMany(e => e.INVOICE_LINES)
            .WithRequired(e => e.PRODUCT)
            .WillCascadeOnDelete(false);

        modelBuilder.Entity<INVOICE>()
            .HasMany(e => e.INVOICE_LINES)
            .WithRequired(e => e.INVOICE)
            .WillCascadeOnDelete(false);
    }
}
```

The properties coded here describe a dataset for each entity, along with advanced properties that are specified for creating a model with Fluent API. A complete description of the Fluent API can be found in the Microsoft document entitled [Configuring/Mapping Properties and Types with the Fluent API](#).

We will use the Fluent API to specify precision and scale for properties of type DECIMAL in the *OnModelCreating* method, by adding the following lines:

```
modelBuilder.Entity<PRODUCT>()
    .Property(p => p.PRICE)
    .HasPrecision(15, 2);
modelBuilder.Entity<INVOICE>()
    .Property(p => p.TOTAL_SALE)
    .HasPrecision(15, 2);

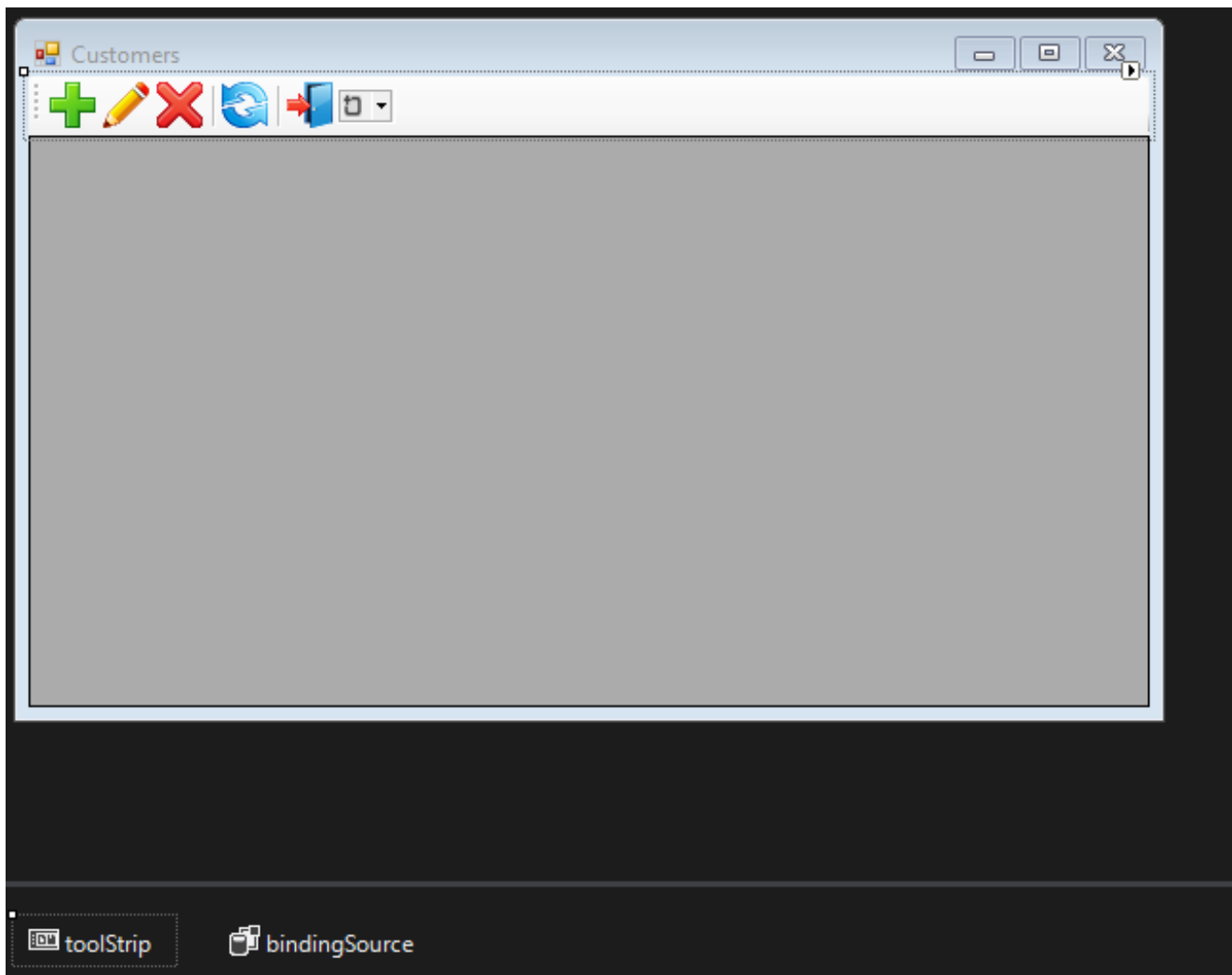
modelBuilder.Entity<INVOICE_LINE>()
    .Property(p => p.SALE_PRICE)
    .HasPrecision(15, 2);

modelBuilder.Entity<INVOICE_LINE>()
    .Property(p => p.QUANTITY)
    .HasPrecision(15, 0);
```

Creating a User Interface

In our application, we will create interfaces for two primary entities: a form each for the product and the customer entities. Each form contains a DataGridView grid, a ToolStrip toolbar with buttons and also a BindingSource component that is used to bind data to the controls on the form.

Figure 3.14. A form for the Customer entity



Since both forms are similar in function and implementation, we will describe just one.

Getting a Context

To work with our model, we will need the method for getting a context (or a model). The following statement is sufficient for that purpose:

```
DbModel dbContext = new DbModel();
```

If no confidential data are stored in the connection string—for example, the password is absent because it will be captured during the authentication process when the application is started—we will need a special method for storing and recovering the connection string or for storing the previously created context. For that, we will create a special class containing some application-level global variables, along with a method for getting a context.

A context might be the start and end dates of a work period, for example.

```
static class AppVariables
{
    private static DbModel dbContext = null;

    /// <summary>
    /// Start date of the working period
    /// </summary>
    public static DateTime StartDate { get; set; }

    /// <summary>
    /// End date of the working period
    /// </summary>
    public static DateTime FinishDate { get; set; }

    /// <summary>
    /// Returns an instance of the model (context)
    /// </summary>
    /// <returns>Model</returns>
    public static DbModel CreateDbContext() {
        dbContext = dbContext ?? new DbModel();
        return dbContext;
    }
}
```

The connection string itself is applied after the authentication process completes successfully during the application launch. We will add the following code to the *Load* event handler of the main form for that.

```
private void MainForm_Load(object sender, EventArgs e) {
    var dialog = new LoginForm();
    if (dialog.ShowDialog() == DialogResult.OK)
    {
        var dbContext = AppVariables.GetDbContext();
        try
        {
            string s = dbContext.Database.Connection.ConnectionString;
            var builder = new FbConnectionStringBuilder(s);
            builder.UserID = dialog.UserName;
            builder.Password = dialog.Password;
            dbContext.Database.Connection.ConnectionString = builder.ConnectionString;
            // try connect
            dbContext.Database.Connection.Open();
        }
        catch (Exception ex)
        {
            // display error
            MessageBox.Show(ex.Message, "Error");
            Application.Exit();
        }
    }
}
```

```
    }  
  }  
  else  
    Application.Exit();  
}
```

Now, to get a context, we use the static *CreateDbContext* method:

```
var dbContext = AppVariables.getDbContext();
```

Working with Data

The entities in the model definition contain no data. The easiest way to load data is to call the *Load* method. For example,

```
private void LoadCustomersData()  
{  
    dbContext.CUSTOMERS.Load();  
    var customers = dbContext.CUSTOMERS.Local;  
    bindingSource.DataSource = customers.ToBindingList();  
}  
  
private void CustomerForm_Load(object sender, EventArgs e)  
{  
    LoadCustomersData();  
    dataGridView.DataSource = bindingSource;  
    dataGridView.Columns["CUSTOMER_ID"].Visible = false;  
}
```

However, this approach has a few drawbacks:

1. The *Load* method loads all data from the CUSTOMER table to memory at once
2. Although lazy properties (INVOICES) are not loaded immediately, but only once they are accessed, they will be loaded anyway when the records are shown in the grid and it will happen each time a group of records is shown
3. Record ordering is not defined

To get around these drawbacks, we will use a feature of the LINQ (Language Integrated Query) technology, *LINQ to Entities*. LINQ to Entities offers a simple and intuitive approach to getting data using C# statements that are syntactically similar to SQL query statements. You can read about the LINQ syntax in [LINQ to Entities](#).

LINQ Extension Methods

The LINQ extension methods can return two objects: *IEnumerable* and *IQueryable*. The *IQueryable* interface is inherited from *IEnumerable* so, theoretically, an *IQueryable* object is also an *IEnumerable*. In reality, they are distinctly different.

The *IEnumerable* interface is in the *System.Collections* namespace. An *IEnumerable* object is a collection of data in memory that can be addressed only in a forward direction. During the query execution, *IEnumerable* loads all data. Filtering, if required, is done on the client side.

The *IQueryable* interface is in the *System.Linq* namespace. It provides remote access to the database and movement through the data can be bi-directional. During the process of creating a query that returns an *IQueryable* object, the query is optimized to minimise memory usage and network bandwidth.

The *Local* property returns the *IEnumerable* interface, through which we can create LINQ queries.

```
private void LoadCustomersData()
{
    var dbContext = AppVariables.getDbContext();
    dbContext.CUSTOMERS.Load();
    var customers =
        from customer in dbContext.CUSTOMERS.Local
        orderby customer.NAME
        select new customer;
    bindingSource.DataSource = customers.ToBindingList();
}
```

However, as this query will be executed on the data in memory, it is really useful only for small tables that do not need to be filtered beforehand.

For a LINQ query to be converted into SQL and executed on the server, we need to access the *dbContext.CUSTOMERS* directly instead of accessing the *dbContext.CUSTOMERS.Local* property in the LINQ query. The prior call to *dbContext.CUSTOMERS.Load()*; to load the collection to memory is not required.

IQueryable and BindingList

IQueryable objects present a small problem: they cannot return *BindingList*. *BindingList* is a base class for creating a two-way data-binding mechanism. We can use the *IQueryable* interface to get a regular list by calling *ToList* but, this way, we lose handy features such as sorting in the grid and several more. The deficiency was fixed in .NET Framework 5 by creating a special extension. To do the same thing in FW4, we will create our own solution.

```
public static class DbExtensions
{
    // Internal class for map generator values to it
    private class IdResult
    {
        public int Id { get; set; }
    }

    // Cast IQueryable to BindingList
    public static BindingList<T> ToBindingList<T>
        (this IQueryable<T> source) where T : class
    {
        return (new ObservableCollection<T>(source)).ToBindingList();
    }
}
```

```
// Get the next value of the sequence
public static int NextValueFor(this DbModel dbContext, string genName)
{
    string sql = String.Format(
        "SELECT NEXT VALUE FOR {0} AS Id FROM RDB$DATABASE", genName);
    return dbContext.Database.SqlQuery<IdResult>(sql).First().Id;
}

// Disconnect all objects from the DbSet collection from the context
// Useful for updating the cache
public static void DetachAll<T>(this DbModel dbContext, DbSet<T> dbSet)
    where T : class
{
    foreach (var obj in dbSet.Local.ToList())
    {
        dbContext.Entry(obj).State = EntityState.Detached;
    }
}

// Update all changed objects in the collection
public static void Refresh(this DbModel dbContext, RefreshMode mode,
    IEnumerable collection)
{
    var objectContext = ((IObjectContextAdapter)dbContext).ObjectContext;
    objectContext.Refresh(mode, collection);
}

// Update the object
public static void Refresh(this DbModel dbContext, RefreshMode mode,
    object entity)
{
    var objectContext = ((IObjectContextAdapter)dbContext).ObjectContext;
    objectContext.Refresh(mode, entity);
}
}
```

Other Extensions

There are several more extensions in the *IQueryable* interface:

NextValueFor

is used to get the next value from the generator.

dbContext.Database.SqlQuery

allows SQL queries to be executed directly and their results to be displayed on some entity (projection).

DetachAll

is used to detach all objects of the DbSet collection from the context. It is necessary to update the internal cache, because all retrieved data are cached and are not retrieved from the database again. However, that is not always useful because it makes it more difficult to get the latest version of records that were modified in another context.

Note

In web applications, a context usually exists for a very short period. A new context has an empty cache.

Refresh

is used to update the properties of an entity object. It is useful for updating the properties of an object after it has been edited or added.

Code for Loading the Data

Our code for loading data will look like this:

```
private void LoadCustomersData()
{
    var dbContext = AppVariables.getDbContext();
    // disconnect all loaded objects
    // this is necessary to update the internal cache
    // for the second and subsequent calls of this method
    dbContext.DetachAll(dbContext.CUSTOMERS);
    var customers =
        from customer in dbContext.CUSTOMERS
        orderby customer.NAME
        select customer;
    bindingSource.DataSource = customers.ToBindingList();
}

private void CustomerForm_Load(object sender, EventArgs e)
{
    LoadCustomersData();
    dataGridView.DataSource = bindingSource;
    dataGridView.Columns["INVOICES"].Visible = false;
    dataGridView.Columns["CUSTOMER_ID"].Visible = false;
    dataGridView.Columns["NAME"].HeaderText = "Name";
    dataGridView.Columns["ADDRESS"].HeaderText = "Address";
    dataGridView.Columns["ZIPCODE"].HeaderText = "ZipCode";
    dataGridView.Columns["PHONE"].HeaderText = "Phone";
}
```

Adding a Customer

This is the code of the event handler for clicking the Add button:

```
private void btnAdd_Click(object sender, EventArgs e) {
    var dbContext = AppVariables.getDbContext();
    // creating a new entity instance
    var customer = (CUSTOMER)bindingSource.AddNew();
    // create an editing form
    using (CustomerEditorForm editor = new CustomerEditorForm()) {
        editor.Text = "Add customer";
        editor.Customer = customer;
        // Form Close Handler
    }
}
```

```

editor.FormClosing += delegate (object fSender,
    FormClosingEventArgs fe) {
    if (editor.DialogResult == DialogResult.OK) {
        try {
            // get next sequence value
            // and assign it
            customer.CUSTOMER_ID = dbContext.NextValueFor("GEN_CUSTOMER_ID");
            // add a new customer
            dbContext.CUSTOMERS.Add(customer);
            // trying to save the changes
            dbContext.SaveChanges();
            // and update the current record
            dbContext.Refresh(RefreshMode.StoreWins, customer);
        }
        catch (Exception ex) {
            // display error
            MessageBox.Show(ex.Message, "Error");
            // Do not close the form to correct the error
            fe.Cancel = true;
        }
    }
    else
        bindingSource.CancelEdit();
};
// show the modal form
editor.ShowDialog(this);
}
}

```

While adding the new record, we used the generator to get the value of the next identifier. We could have done it without applying the value of the identifier, leaving the BEFORE INSERT trigger to fetch the next value of the generator and apply it. However, that would leave us unable to update the added record.

Editing a Customer

The code of the event handler for clicking the Edit button is as follows:

```

private void btnEdit_Click(object sender, EventArgs e) {
    var dbContext = AppVariables.getDbContext();
    // get instance
    var customer = (CUSTOMER)bindingSource.Current;
    // create an editing form
    using (CustomerEditorForm editor = new CustomerEditorForm()) {
        editor.Text = "Edit customer";
        editor.Customer = customer;
        // Form Close Handler
        editor.FormClosing += delegate (object fSender, FormClosingEventArgs fe) {
            if (editor.DialogResult == DialogResult.OK) {
                try {
                    // trying to save the changes
                    dbContext.SaveChanges();
                    dbContext.Refresh(RefreshMode.StoreWins, customer);
                    // update all related controls
                    bindingSource.ResetCurrentItem();
                }
                catch (Exception ex) {

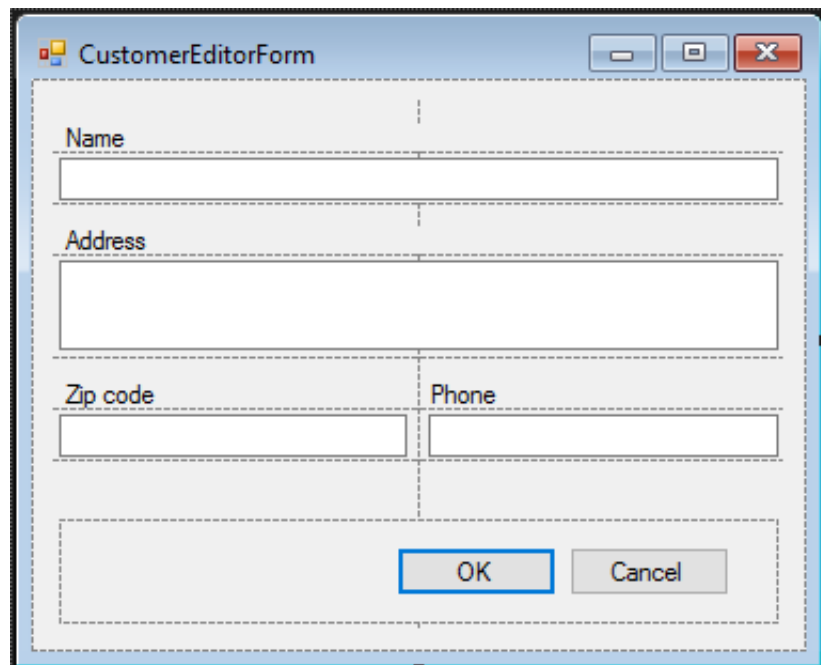
```



```
// display error
MessageBox.Show(ex.Message, "Error");
// Do not close the form to correct the error
fe.Cancel = true;
    }
}
else
    bindingSource.CancelEdit();
};
// show the modal form
editor.ShowDialog(this);
}
}
```

The form for editing the customer looks like this:

Figure 3.15. Customer edit form



The code for binding to data is very simple.

```
public CUSTOMER Customer { get; set; }

private void CustomerEditorForm_Load(object sender, EventArgs e)
{
    edtName.DataBindings.Add("Text", this.Customer, "NAME");
    edtAddress.DataBindings.Add("Text", this.Customer, "ADDRESS");
    edtZipCode.DataBindings.Add("Text", this.Customer, "ZIPCODE");
    edtPhone.DataBindings.Add("Text", this.Customer, "PHONE");
}
```

Deleting a Customer

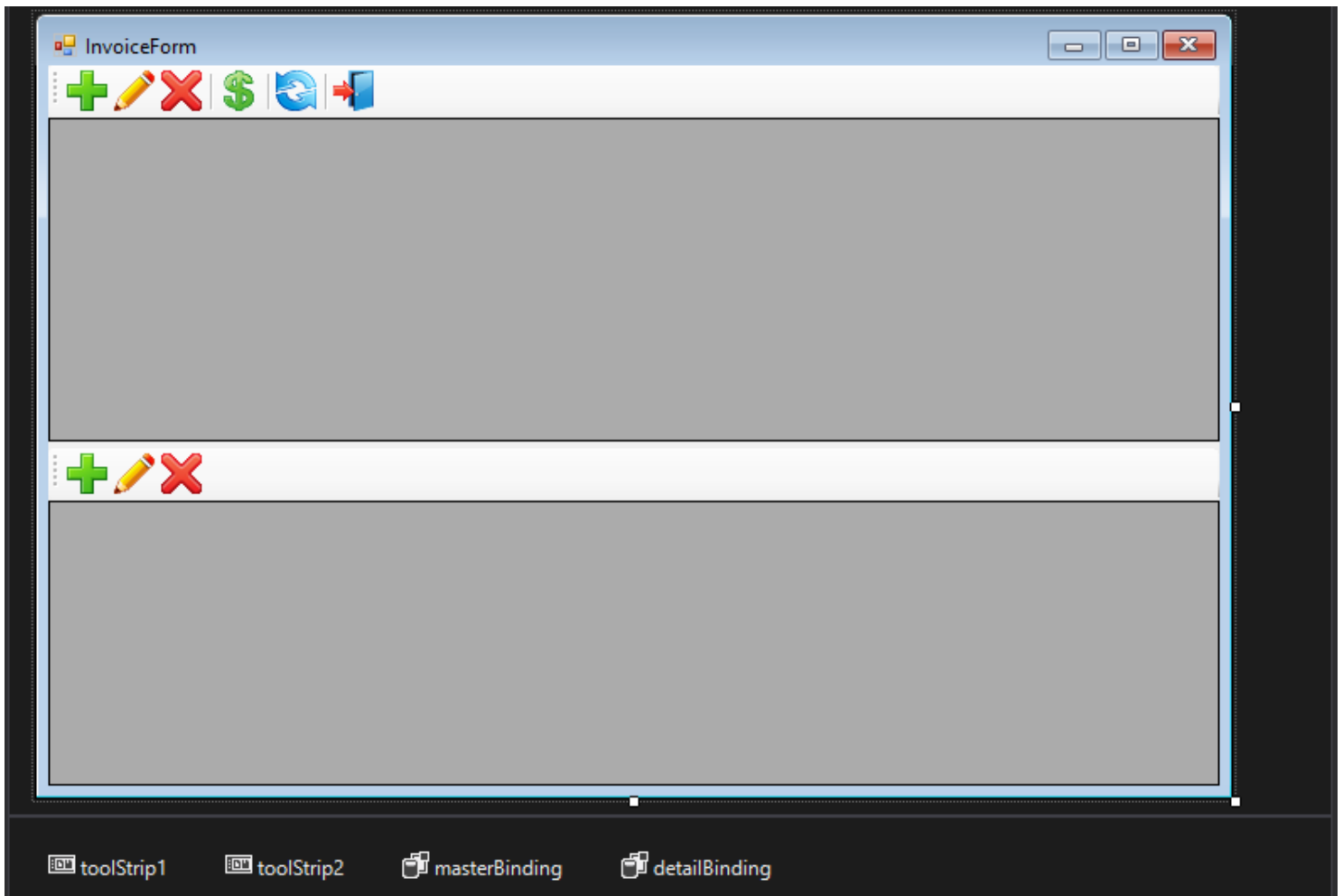
The code of the event handler for clicking the Delete button is as follows:

```
private void btnDelete_Click(object sender, EventArgs e) {
    var dbContext = AppVariables.getDbContext();
    var result = MessageBox.Show("Are you sure you want to delete the customer?",
        "Confirmation",
        MessageBoxButtons.YesNo,
        MessageBoxIcon.Question);
    if (result == DialogResult.Yes) {
        // get the entity
        var customer = (CUSTOMER)bindingSource.Current;
        try {
            dbContext.CUSTOMERS.Remove(customer);
            // trying to save the changes
            dbContext.SaveChanges();
            // remove from the linked list
            bindingSource.RemoveCurrent();
        }
        catch (Exception ex) {
            // display error
            MessageBox.Show(ex.Message, "Error");
        }
    }
}
```

Secondary Modules

Our application will have only one secondary module, named “Invoices”. Secondary modules typically contain larger numbers of records than primary ones and new records are added to them 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 showing the invoice header data and the detail one for the list of products sold. We will need one DataGridView component for each entity on the document form, binding the appropriate BindingSource to each.

Figure 3.16. Invoice form

Filtering Data

Most secondary entities contain a field with the document creation date. To reduce the amount of retrieved data, the concept of a work period is usually introduced to filter the 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 entity, it makes sense to add variables containing the start and end dates of a work period to the global *AppVariables* data module (see [Getting a Context](#) that is used by all modules working with the database in one way or another. Once the application is started, the work period is usually defined by the dates when the current quarter starts and ends, although of course, other options are possible. While working with the application, the user can change the work period.

Since the most recent records are the most requested, it makes sense to sort them by date in reverse order. As with the primary modules, we will use LINQ to retrieve data.

Loading the Invoice Data

The following method loads the invoice headers:

```
public void LoadInvoicesData() {
    var dbContext = AppVariables.getDbContext();

    var invoices =
        from invoice in dbContext.INVOICES
        where (invoice.INVOICE_DATE >= AppVariables.StartDate) &&
            (invoice.INVOICE_DATE <= AppVariables.FinishDate)
        orderby invoice.INVOICE_DATE descending
        select new InvoiceView
        {
            Id = invoice.INVOICE_ID,
            Customer_Id = invoice.CUSTOMER_ID,
            Customer = invoice.CUSTOMER.NAME,
            Date = invoice.INVOICE_DATE,
            Amount = invoice.TOTAL_SALE,
            Payed = (invoice.PAYED == 1) ? "Yes" : "No"
        };
    masterBinding.DataSource = invoices.ToBindingList();
}
```

To simplify type casting, we define an *InvoiceView* class, rather than use some anonymous type. The definition is as follows:

```
public class InvoiceView {
    public int Id { get; set; }

    public int Customer_Id { get; set; }

    public string Customer { get; set; }

    public DateTime? Date { get; set; }

    public decimal? Amount { get; set; }

    public string Payed { get; set; }

    public void Load(int Id) {
        var dbContext = AppVariables.getDbContext();
        var invoices =
            from invoice in dbContext.INVOICES
            where invoice.INVOICE_ID == Id
            select new InvoiceView
            {
                Id = invoice.INVOICE_ID,
                Customer_Id = invoice.CUSTOMER_ID,
                Customer = invoice.CUSTOMER.NAME,
                Date = invoice.INVOICE_DATE,
                Amount = invoice.TOTAL_SALE,
                Payed = (invoice.PAYED == 1) ? "Yes" : "No"
            };

        InvoiceView invoiceView = invoices.ToList().First();
        this.Id = invoiceView.Id;
        this.Customer_Id = invoiceView.Customer_Id;
        this.Customer = invoiceView.Customer;
    }
}
```

```

        this.Date = invoiceView.Date;
        this.Amount = invoiceView.Amount;
        this.Payed = invoiceView.Payed;
    }
}

```

The *Load* method allows us to update one added or updated record in the grid quickly, instead of completely reloading all records. Here is the code of the event handler for clicking the Add button:

```

private void btnAddInvoice_Click(object sender, EventArgs e) {
    var dbContext = AppVariables.getDbContext();
    var invoice = dbContext.INVOICES.Create();
    using (InvoiceEditorForm editor = new InvoiceEditorForm()) {
        editor.Text = "Add invoice";
        editor.Invoice = invoice;
        // Form Close Handler
        editor.FormClosing += delegate (object fSender, FormClosingEventArgs fe) {
            if (editor.DialogResult == DialogResult.OK) {
                try {
                    // get next sequence value
                    invoice.INVOICE_ID = dbContext.NextValueFor("GEN_INVOICE_ID");
                    // add a record
                    dbContext.INVOICES.Add(invoice);
                    // trying to save the changes
                    dbContext.SaveChanges();
                    // add the projection to the grid list
                    ((InvoiceView)masterBinding.AddNew()).Load(invoice.INVOICE_ID);
                }
                catch (Exception ex) {
                    // display error
                    MessageBox.Show(ex.Message, "Error");
                    // Do not close the form to correct the error
                    fe.Cancel = true;
                }
            }
        };
        // show the modal form
        editor.ShowDialog(this);
    }
}

```

In our primary modules, the similarly-named method called *dbContext.Refresh* but, here, a record is updated by by calling the *Load* method of the *InvoiceView* class. The reason for the difference is that *dbContext.Refresh* is used to update entity objects, not the objects that can be produced by complex LINQ queries.

The code of the event handler for clicking the Edit button:

```

private void btnEditInvoice_Click(object sender, EventArgs e) {
    var dbContext = AppVariables.getDbContext();
    // find entity by id
    var invoice = dbContext.INVOICES.Find(this.CurrentInvoice.Id);
    if (invoice.PAYED == 1) {
        MessageBox.Show("The change is not possible, the invoice has already been paid.",
            "Error");
        return;
    }
}

```

```
}
using (InvoiceEditorForm editor = new InvoiceEditorForm()) {
    editor.Text = "Edit invoice";
    editor.Invoice = invoice;
    // Form Close Handler
    editor.FormClosing += delegate (object fSender, FormClosingEventArgs fe) {
        if (editor.DialogResult == DialogResult.OK) {
            try {
                // trying to save the changes
                dbContext.SaveChanges();
                // refresh
                CurrentInvoice.Load(invoice.INVOICE_ID);
                masterBinding.ResetCurrentItem();
            }
            catch (Exception ex) {
                // display error
                MessageBox.Show(ex.Message, "Error");
                // Do not close the form to correct the error
                fe.Cancel = true;
            }
        }
    };
    editor.ShowDialog(this);
}
```

Here we needed to find an entity by the identifier provided in the current record. The `CurrentInvoice` is used to retrieve the invoice selected in the grid. This is how we code it:

```
public InvoiceView CurrentInvoice {
    get {
        return (InvoiceView)masterBinding.Current;
    }
}
```

Using the same approach, you can implement deleting the invoice header yourself.

Paying an Invoice

Besides adding, editing and deleting, we want one more operation for invoices: payment. Here is code for a method implementing this operation:

```
private void btnInvoicePay_Click(object sender, EventArgs e) {
    var dbContext = AppVariables.getDbContext();
    var invoice = dbContext.INVOICES.Find(this.CurrentInvoice.Id);
    try {
        if (invoice.PAYED == 1)
            throw new Exception("The change is not possible, the invoice has already been paid.");
        invoice.PAYED = 1;
        // trying to save the changes
        dbContext.SaveChanges();
        // refresh record
        CurrentInvoice.Load(invoice.INVOICE_ID);
    }
```

```
        masterBinding.ResetCurrentItem();
    }
    catch (Exception ex) {
        // display error
        MessageBox.Show(ex.Message, "Error");
    }
}
```

Showing the Invoice Lines

We have two choices for displaying the invoice lines:

1. Getting data for each invoice from the INVOICE_LINE navigation property and displaying the contents of this complex property in the detail grid, probably with LINQ transformations
2. Getting the data for each invoice with a separate LINQ query that will be re-executed when the cursor moves to another record in the master grid

Either way has its advantages and drawbacks.

The first one assumes that we want to retrieve all invoices at once for the specified period together with the bound data from the invoice lines when the invoice form is opened. Although it is done with one SQL query, it may take quite a while and requires a large amount of random-access memory. It is better suited to web applications where records are usually displayed page by page.

The second one is a bit more difficult to implement, but it allows the invoice form to be opened quickly and requires less resource. However, each time the cursor in the master grid moves, an SQL query will be executed, generating network traffic, albeit with only a small volume of data.

For our application we will use the second approach. We need an event handler for the BindingSource component for editing the current record:

```
private void masterBinding_CurrentChanged(object sender, EventArgs e) {
    LoadInvoiceLineData(this.CurrentInvoice.Id);
    detailGridView.DataSource = detailBinding;
}
```

Now, the method for loading the invoice data:

```
private void LoadInvoiceLineData(int? id) {
    var dbContext = AppVariables.getDbContext();
    var lines =
        from line in dbContext.INVOICE_LINES
        where line.INVOICE_ID == id
        select new InvoiceLineView
        {
            Id = line.INVOICE_LINE_ID,
            Invoice_Id = line.INVOICE_ID,
            Product_Id = line.PRODUCT_ID,
            Product = line.PRODUCT.NAME,
            Quantity = line.QUANTITY,
        }
}
```

```

        Price = line.SALE_PRICE,
        Total = Math.Round(line.QUANTITY * line.SALE_PRICE, 2)
    };
    detailBinding.DataSource = lines.ToBindingList();
}

```

We use the InvoiceLineView class as an extension:

```

public class InvoiceLineView {
    public int Id { get; set; }
    public int Invoice_Id { get; set; }
    public int Product_Id { get; set; }
    public string Product { get; set; }
    public decimal Quantity { get; set; }
    public decimal Price { get; set; }
    public decimal Total { get; set; }
}

```

Note

Unlike the InvoiceView class, this one has no method for loading one current record. In our example, the speed of reloading the detail grid it is not crucial, because one document does not contain thousands of items. Implementing this method is optional.

Now we will add a special property for retrieving the current line of the document selected in the detail grid.

```

public InvoiceLineView CurrentInvoiceLine {
    get {
        return (InvoiceLineView)detailBinding.Current;
    }
}

```

Working with Stored Procedures

The methods we will use for adding, editing and deleting illustrate how to work with stored procedures in Entity Framework. As an example, this is the method for adding a new record:

```

private void btnAddInvoiceLine_Click(object sender, EventArgs e) {
    var dbContext = AppVariables.getDbContext();
    // get current invoice
    var invoice = dbContext.INVOICES.Find(this.CurrentInvoice.Id);
    if (invoice.PAYED == 1) {
        MessageBox.Show("The change is not possible, the invoice has already been paid.", "Error");
        return;
    }
    // create invoice position
    var invoiceLine = dbContext.INVOICE_LINES.Create();
    invoiceLine.INVOICE_ID = invoice.INVOICE_ID;
    // create the position editor of the invoice
}

```



```

using (InvoiceLineEditorForm editor = new InvoiceLineEditorForm()) {
    editor.Text = "Add invoice line";
    editor.InvoiceLine = invoiceLine;
    // Form Close Handler
    editor.FormClosing += delegate (object fSender, FormClosingEventArgs fe) {
        if (editor.DialogResult == DialogResult.OK) {
            try {
                // create SP parameters
                var invoiceIdParam = new FbParameter("INVOICE_ID",
                                                    FbDbType.Integer);
                var productIdParam = new FbParameter("PRODUCT_ID",
                                                    FbDbType.Integer);
                var quantityParam = new FbParameter("QUANTITY", FbDbType.Integer);
                // initial parameters values
                invoiceIdParam.Value = invoiceLine.INVOICE_ID;
                productIdParam.Value = invoiceLine.PRODUCT_ID;
                quantityParam.Value = invoiceLine.QUANTITY;
                // execute stored procedure
                dbContext.Database.ExecuteSqlCommand(
                    "EXECUTE PROCEDURE SP_ADD_INVOICE_LINE("
                    + "@INVOICE_ID, @PRODUCT_ID, @QUANTITY)",
                    invoiceIdParam,
                    productIdParam,
                    quantityParam);
                // refresh grids
                // reload current invoice record
                CurrentInvoice.Load(invoice.INVOICE_ID);
                // reload all record in detail grid
                LoadInvoiceLineData(invoice.INVOICE_ID);
                // refresh all related data
                masterBinding.ResetCurrentItem();
            }
            catch (Exception ex) {
                // display error
                MessageBox.Show(ex.Message, "Error");
                // Do not close the form to correct the error
                fe.Cancel = true;
            }
        }
    };
    editor.ShowDialog(this);
}

```

With our example, an update of the master grid record will be needed because one of its fields (TotalSale) contains aggregated information derived from the detail lines of the document. This is how we do that:

```

private void btnEditInvoiceLine_Click(object sender, EventArgs e) {
    var dbContext = AppVariables.getDbContext();
    // get current invoice
    var invoice = dbContext.INVOICES.Find(this.CurrentInvoice.Id);
    if (invoice.PAYED == 1) {
        MessageBox.Show("The change is not possible, the invoice has already been paid.",
                        "Error");
        return;
    }
}

```

```
// get current invoice position
var invoiceLine = invoice.INVOICE_LINES
    .Where(p => p.INVOICE_LINE_ID == this.CurrentInvoiceLine.Id)
    .First();
// create invoice position editor
using (InvoiceLineEditorForm editor = new InvoiceLineEditorForm()) {
    editor.Text = "Edit invoice line";
    editor.InvoiceLine = invoiceLine;
    // form close handler
    editor.FormClosing += delegate (object fSender, FormClosingEventArgs fe) {
        if (editor.DialogResult == DialogResult.OK) {
            try {
                // create parameters
                var idParam = new FbParameter("INVOICE_LINE_ID", FbDbType.Integer);
                var quantityParam = new FbParameter("QUANTITY", FbDbType.Integer);
                // initial parameters values
                idParam.Value = invoiceLine.INVOICE_LINE_ID;
                quantityParam.Value = invoiceLine.QUANTITY;
                // execute stored procedure
                dbContext.Database.ExecuteSqlCommand(
                    "EXECUTE PROCEDURE SP_EDIT_INVOICE_LINE("
                    + "@INVOICE_LINE_ID, @QUANTITY)",
                    idParam,
                    quantityParam);
                // refresh grids
                // reload current invoice record
                CurrentInvoice.Load(invoice.INVOICE_ID);
                // reload all records in detail grid
                LoadInvoiceLineData(invoice.INVOICE_ID);
                // refresh all related controls
                masterBinding.ResetCurrentItem();
            }
            catch (Exception ex) {
                // display error
                MessageBox.Show(ex.Message, "Error");
                // Do not close the form to correct the error
                fe.Cancel = true;
            }
        }
    };
    editor.ShowDialog(this);
}
```

Deleting an Invoice Detail Line

The method for deleting a detail record is implemented as follows:

```
private void btnDeleteInvoiceLine_Click(object sender, EventArgs e) {
    var result = MessageBox.Show(
        " Are you sure you want to delete the invoice item?",
        "Confirmation",
        MessageBoxButtons.YesNo,
        MessageBoxIcon.Question);
    if (result == DialogResult.Yes) {
```

```

var dbContext = AppVariables.getDbContext();
// get current invoice
var invoice = dbContext.INVOICES.Find(this.CurrentInvoice.Id);
try {
    if (invoice.PAYED == 1)
        throw new Exception("It is not possible to delete the entry, the invoice is paid.");
    // create parameters
    var idParam = new FbParameter("INVOICE_LINE_ID", FbDbType.Integer);
    // initialize parameters values
    idParam.Value = this.CurrentInvoiceLine.Id;
    // execute stored procedure
    dbContext.Database.ExecuteSqlCommand(
        "EXECUTE PROCEDURE SP_DELETE_INVOICE_LINE(@INVOICE_LINE_ID)",
        idParam);
    // update grids
    // reload current invoice
    CurrentInvoice.Load(invoice.INVOICE_ID);
    // reload all records in detail grids
    LoadInvoiceLineData(invoice.INVOICE_ID);
    // refresh related controls
    masterBinding.ResetCurrentItem();
}
catch (Exception ex) {
    // display error
    MessageBox.Show(ex.Message, "Error");
}
}
}

```

Showing Products for Selection

In the methods for adding and editing invoice lines we used the form. For displaying products, we will use a TextBox control.

Figure 3.17. Product form

The image shows a Windows-style dialog box titled "InvoiceLineEditorForm". It contains three input fields arranged in a grid-like fashion. The first field is labeled "Product" and is a text box with a small folder icon to its right. Below it, there are two more fields: "Cost" and "Quantity". The "Cost" field is a text box, and the "Quantity" field is a spinner box (up and down arrows) with the number "1" displayed. At the bottom of the dialog, there are two buttons: "OK" and "Cancel".

A click on the button next to the TextBox will open a modal form with a grid for selecting products. The same modal form created for displaying the products is used for selecting them. The click handler code for the embedded button that initiates the form is:

```
public partial class InvoiceLineEditorForm : Form {
    public InvoiceLineEditorForm() {
        InitializeComponent();
    }

    public INVOICE_LINE InvoiceLine { get; set; }

    private void InvoiceLineEditorForm_Load(object sender, EventArgs e) {
        if (this.InvoiceLine.PRODUCT != null) {
            edtProduct.Text = this.InvoiceLine.PRODUCT.NAME;
            edtPrice.Text = this.InvoiceLine.PRODUCT.PRICE.ToString("F2");
            btnChooseProduct.Click -= this.btnChooseProduct_Click;
        }
        if (this.InvoiceLine.QUANTITY == 0)
            this.InvoiceLine.QUANTITY = 1;
        edtQuantity.DataBindings.Add("Value", this.InvoiceLine, "QUANTITY");
    }

    private void btnChooseProduct_Click(object sender, EventArgs e) {
        GoodsForm goodsForm = new GoodsForm();
        if (goodsForm.ShowDialog() == DialogResult.OK) {
            InvoiceLine.PRODUCT_ID = goodsForm.CurrentProduct.Id;
            edtProduct.Text = goodsForm.CurrentProduct.Name;
            edtPrice.Text = goodsForm.CurrentProduct.Price.ToString("F2");
        }
    }
}
```

Working with Transactions

Whenever we call the *SaveChanges()* method while adding, updating or deleting, Entity Framework starts and ends an implicit transaction. Since we use disconnected data access, all operations are carried out within one transaction. Entity Framework starts and ends a transaction automatically for each data retrieval. We will take the following example to illustrate how automatic transactions work.

Suppose we need to make a discount on goods selected in the grid. Without explicit transaction management, the code would be as follows:

```
var dbContext = AppVariables.getDbContext();
foreach (DataGridViewRow gridRows in dataGridView.SelectedRows) {
    int id = (int)gridRows.Cells["Id"].Value;
    // here there is an implicit start and the completion of the transaction
    var product = dbContext.PRODUCTS.Find(id);
    // discount 10%
    decimal discount = 10.0m;
    product.PRICE = product.PRICE * (100 - discount) /100;
}
```

```
// here there is an implicit start and the completion of the transaction
// all changes occur in one transaction
dbContext.SaveChanges();
```

Let's say we select 10 products. Ten implicit transactions will be used for finding the products by their identifiers. One more transaction will be used to save the changes.

If we control transactions explicitly, we can use just one transaction for the same piece of work. For example:

```
var dbContext = AppVariables.getDbContext();
// explicit start of a default transaction
using (var dbTransaction = dbContext.Database.BeginTransaction()) {
    string sql =
        "UPDATE PRODUCT " +
        "SET PRICE = PRICE * ROUND((100 - @DISCOUNT)/100, 2) " +
        "WHERE PRODUCT_ID = @PRODUCT_ID";
    try {
        // create query parameters
        var idParam = new FbParameter("PRODUCT_ID", FbDbType.Integer);
        var discountParam = new FbParameter("DISCOUNT", FbDbType.Decimal);
        // create a SQL command to update records
        var sqlCommand = dbContext.Database.Connection.CreateCommand();
        sqlCommand.CommandText = sql;
        // specify which transaction to use
        sqlCommand.Transaction = dbTransaction.UnderlyingTransaction;
        sqlCommand.Parameters.Add(discountParam);
        sqlCommand.Parameters.Add(idParam);
        // prepare query
        sqlCommand.Prepare();
        // for all selected records in the grid
        foreach (DataGridViewRow gridRows in dataGridView.SelectedRows) {
            int id = (int)gridRows.Cells["Id"].Value;
            // initialize query parameters
            idParam.Value = id;
            discountParam.Value = 10.0m; // discount 10%
            // execute sql statement
            sqlCommand.ExecuteNonQuery();
        }
        dbTransaction.Commit();
    }
    catch (Exception ex) {
        dbTransaction.Rollback();
        MessageBox.Show(ex.Message, "error");
    }
}
```

Our code starts the transaction with the default parameters. To specify your own parameters for a transaction, you should use the *UseTransaction* method.

```
private void btnDiscount_Click(object sender, EventArgs e) {
    DiscountEditorForm editor = new DiscountEditorForm();
    editor.Text = "Enter discount";
    if (editor.ShowDialog() != DialogResult.OK)
        return;
}
```

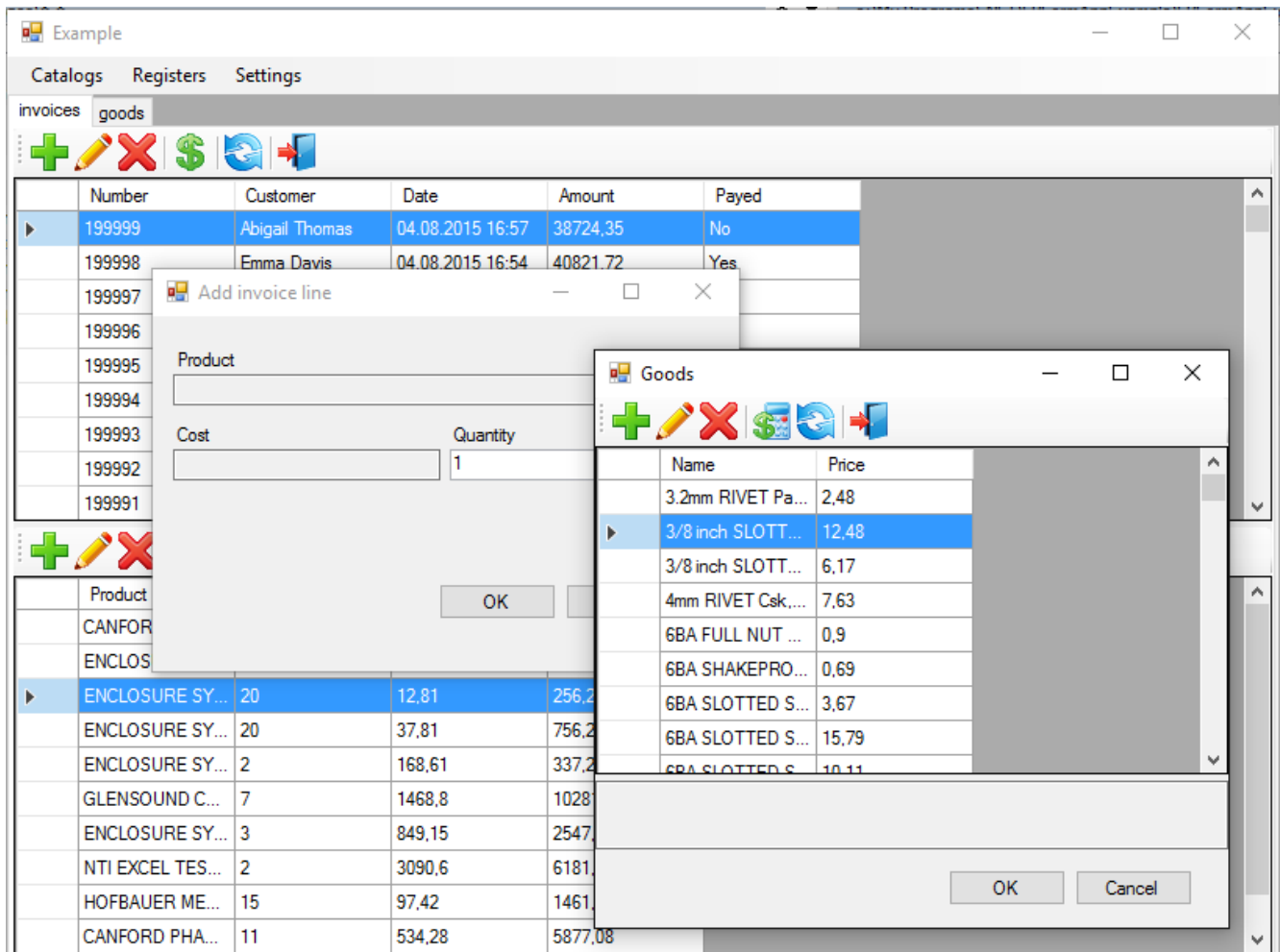
```
bool needUpdate = false;
var dbContext = AppVariables.getDbContext();
var connection = dbContext.Database.Connection;
// explicit start of transaction
using (var dbTransaction = connection.BeginTransaction(IsolationLevel.Snapshot)) {
    dbContext.Database.UseTransaction(dbTransaction);
    string sql =
        "UPDATE PRODUCT " +
        "SET PRICE = ROUND(PRICE * (100 - @DISCOUNT)/100, 2) " +
        "WHERE PRODUCT_ID = @PRODUCT_ID";
    try {
        // create query parameters
        var idParam = new FbParameter("PRODUCT_ID", FbDbType.Integer);
        var discountParam = new FbParameter("DISCOUNT", FbDbType.Decimal);
        // create a SQL command to update records
        var sqlCommand = connection.CreateCommand();
        sqlCommand.CommandText = sql;
        // specify which transaction to use
        sqlCommand.Transaction = dbTransaction;
        sqlCommand.Parameters.Add(discountParam);
        sqlCommand.Parameters.Add(idParam);
        // prepare statement
        sqlCommand.Prepare();
        // for all selected records in the grid
        foreach (DataGridViewRow gridRows in dataGridView.SelectedRows) {
            int id = (int)gridRows.Cells["PRODUCT_ID"].Value;
            // initialize query parameters
            idParam.Value = id;
            discountParam.Value = editor.Discount;
            // execute SQL statement
            needUpdate = (sqlCommand.ExecuteNonQuery() > 0) || needUpdate;
        }
        dbTransaction.Commit();
    }
    catch (Exception ex) {
        dbTransaction.Rollback();
        MessageBox.Show(ex.Message, "error");
        needUpdate = false;
    }
}
// refresh grid
if (needUpdate) {
    // for all selected records in the grid
    foreach (DataGridViewRow gridRows in dataGridView.SelectedRows) {
        var product = (PRODUCT)bindingSource.List[gridRows.Index];
        dbContext.Refresh(RefreshMode.StoreWins, product);
    }
    bindingSource.ResetBindings(false);
}
}
```

That's it. Now only one transaction is used for the entire set of updates and there are no unnecessary commands for finding data.

All that is left to do is to add a dialog box for entering the value of the discount and code to update data in the grid. Try to do it on your own.

The Result

Figure 3.18. The result of the Entity Framework project



Source Code

You can get the source code for the sample application using this link: <https://github.com/sim1984/FB-FormAppExample>.

Appendix A:

License notice

The contents of this Documentation are subject to the Public Documentation License Version 1.0 (the “License”); you may only use this Documentation if you comply with the terms of this License. Copies of the License are available at <http://www.firebirdsql.org/pdfmanual/pdl.pdf> (PDF) and <http://www.firebirdsql.org/manual/pdl.html> (HTML).

The Original Documentation is titled *Firebird 3.0 Developer's Guide*.

The Initial Writer of the Original Documentation is Denis Simonov.

Copyright (C) 2017. All Rights Reserved. Initial Writers contact: paul at vinkenoog dot nl.

Included portions are Copyright (C) 2001-2017 by the author. All Rights Reserved.