Создание приложений для СУБД Firebird с использованием различных компонент и драйверов:

FireDAC

Автор: Денис Симонов
18.12.2015

Создание приложений с использованием FireDac

В данной статье будет описан процесс создания приложений для СУБД Firebird с использованием компонентов доступа FireDac и среды Delphi XE5. FireDac является стандартным набором компонентов доступа к различным базам данных начиная с Delphi XE3.

Наше приложение будет работать с базой данных модель, которой представлена на рисунке ниже.

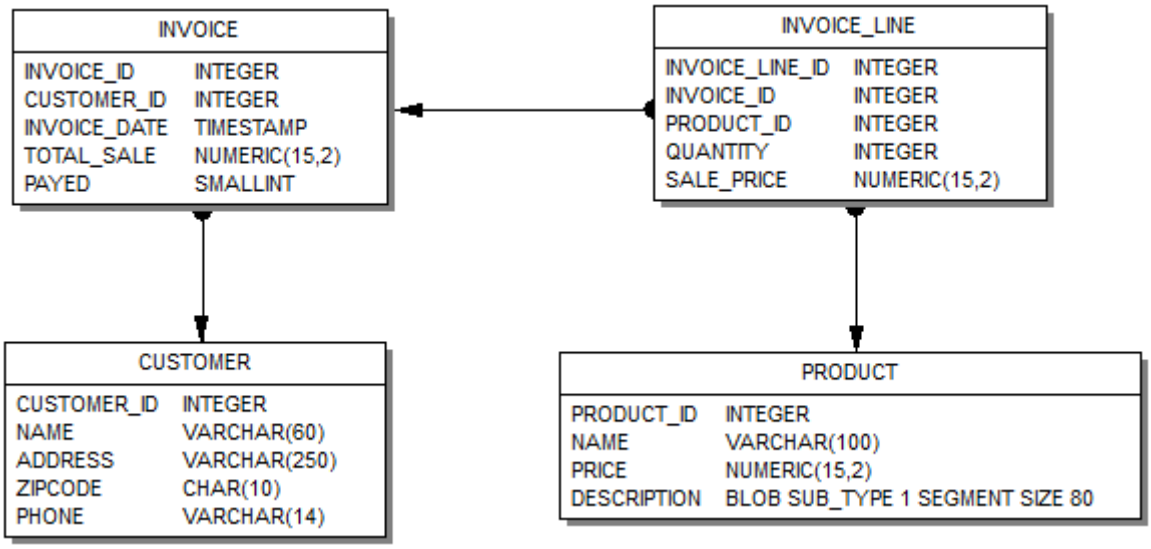

Рисунок 1. Модель базы данных.

В конце данной статьи приведены ссылки на другие статьи, в которых описывается процесс создания базы данных с этой моделью и ссылка на скрипт создания базы данных.

Внимание!

Эта модель является просто примером. Ваша предметная область может быть сложнее, или полностью другой. Модель, используемая в этой статье, максимально упрощена для того, чтобы не загромождать описание работы с компонентами описанием создания и модификации модели данных.

Создайте новый проект File->New->VCL Forms Application - Delphi. В новый проект добавьте новый дата модуль File->New->Other, в появившемся мастере выберите Delphi Projects->Delphi Files->Data Module. Этот дата модуль будет главным в нашем проекте. Он будет содержать некоторые экземпляры глобальных компонентов доступа, которые должны быть доступны всем формам, которые должны работать с данными. Например, таким компонентом является TFDConnection.

Компонент TFDConnection

Компонент TFDConnection обеспечивает подключение к различным типам баз данных. Будем указывать экземпляр этого компонента в свойствах Connection остальных компонентов FireDac. К какому именно типу баз данных будет происходить подключение, зависит от значения свойства DriverName. Для доступа к Firebird нам необходимо выставить это свойство в значение *FB*. Для того чтобы подключение знало, с какой именно библиотекой доступа необходимо работать, разместим в главном дата модуле компонент TFBPhysFBDriverLink. Его свойство VendorLib позволяет указывать путь до клиентской библиотеки. Если оно не указано, то подключение к Firebird будет осуществляться через библиотеки, зарегистрированные в системе, например в system32, что в ряде случаев может быть нежелательно.

Путь к клиентской библиотеке

Мы будем размещать необходимую библиотеку доступа в папке fbclient, которая расположена в папке приложения. Для этого в коде на событие OnCreate дата модуля пропишем следующий код.

```
// указываем путь до клиентской библиотеки
xAppPath := ExtractFileDir(Application.ExeName) + PathDelim;
FDPhysFBDriverLink.VendorLib := xAppPath + 'fbclient' + PathDelim +
  'fbclient.dll';
```

Важно!

Если вы компилируете 32 разрядное приложение, то вы должны использовать 32 разрядную библиотеку fbclient.dll. Для 64 разрядного – 64 разрядную. Помимо файла fbclient.dll в ту же папку желательно поместить библиотеки msvcp80.dll и msvcr80.dll (для Firebird 2.5), и msvcp100.dll и msvcr100.dll (для Firebird 3.0). Эти библиотеки можно найти либо в подпапке bin (Firebird 2.5), либо в корневой папке сервера (Firebird 3.0).

Для того чтобы приложение правильно отображало собственные ошибки firebird, необходимо также скопировать файл firebird.msg. Для Firebird 2.5 и в более ранних версиях он должен находиться на один уровень выше каталога клиентской библиотеки, т.е. в нашем случае в каталоге приложения. Для Firebird 3 он должен находиться в каталоге клиентской библиотеки, т.е. в каталоге fbclient.

Если вам необходимо чтобы ваше приложение работало без установленного сервера Firebird, т.е. в режиме Embedded, то для Firebird 2.5 необходимо заменить fbclient.dll на fbembed.dll. При желании имя библиотеки можно вынести в

конфигурационный файл вашего приложения. Для Firebird 3.0 ничего изменять не требуется (режим работы зависит от строки подключения и значения параметра Providers в файле firebird.conf/databases.conf).

Совет

Даже если ваше приложение будет работать с Firebird в режиме Embedded, разработку удобнее вести под полноценным сервером. Дело в том, что в режиме Embedded Firebird работает в одном адресном пространстве с вашим приложением, что может привести к нежелательным последствиям при возникновении ошибок в вашем приложении. Кроме того, в момент разработки среда Delphi и ваше приложение являются отдельными приложениями, использующими Embedded. До версии 2.5 они не могут работать с одной базой одновременно.

Параметры подключения

Компонент TFDConnection параметры подключения к базе данных содержатся в свойстве Params (имя пользователя, пароль, набор символов соединения и др.). Если воспользоваться редактором свойств TFDConnection (двойной клик на компоненте), то упомянутые свойства будут заполнены автоматически. Набор этих свойств зависит от типа базы данных.

Таблица 1. Основные свойство компонента TFDConnection

Параметр	Назначение
Pooled	Используется ли пул соединений.
Database	Путь к базе данных или её псевдоним, определённый в файле конфигурации aliases.conf (или databases.conf) сервера Firebird.
User_Name	Имя пользователя.
Password	Пароль.
OSAuthent	Используется ли аутентификация средствами операционной системы.
Protocol	Протокол соединения. Допускаются следующие значения: • Local – локальный протокол; • NetBEUI – именованные каналы; • SPX – локальный протокол (до Firebird 2.0, не поддерживается в современных версиях); • TCPIP – TCP/IP.
Server	Имя сервера или его IP адрес. Если сервер работает на нестандартном порту, то необходимо также указать порт через слэш, например localhost/3051.
SQLDialect	Диалект. Должен совпадать с диалектом базы данных.
RoleName	Имя роли.
CharacterSet	Имя набора символов соединения.

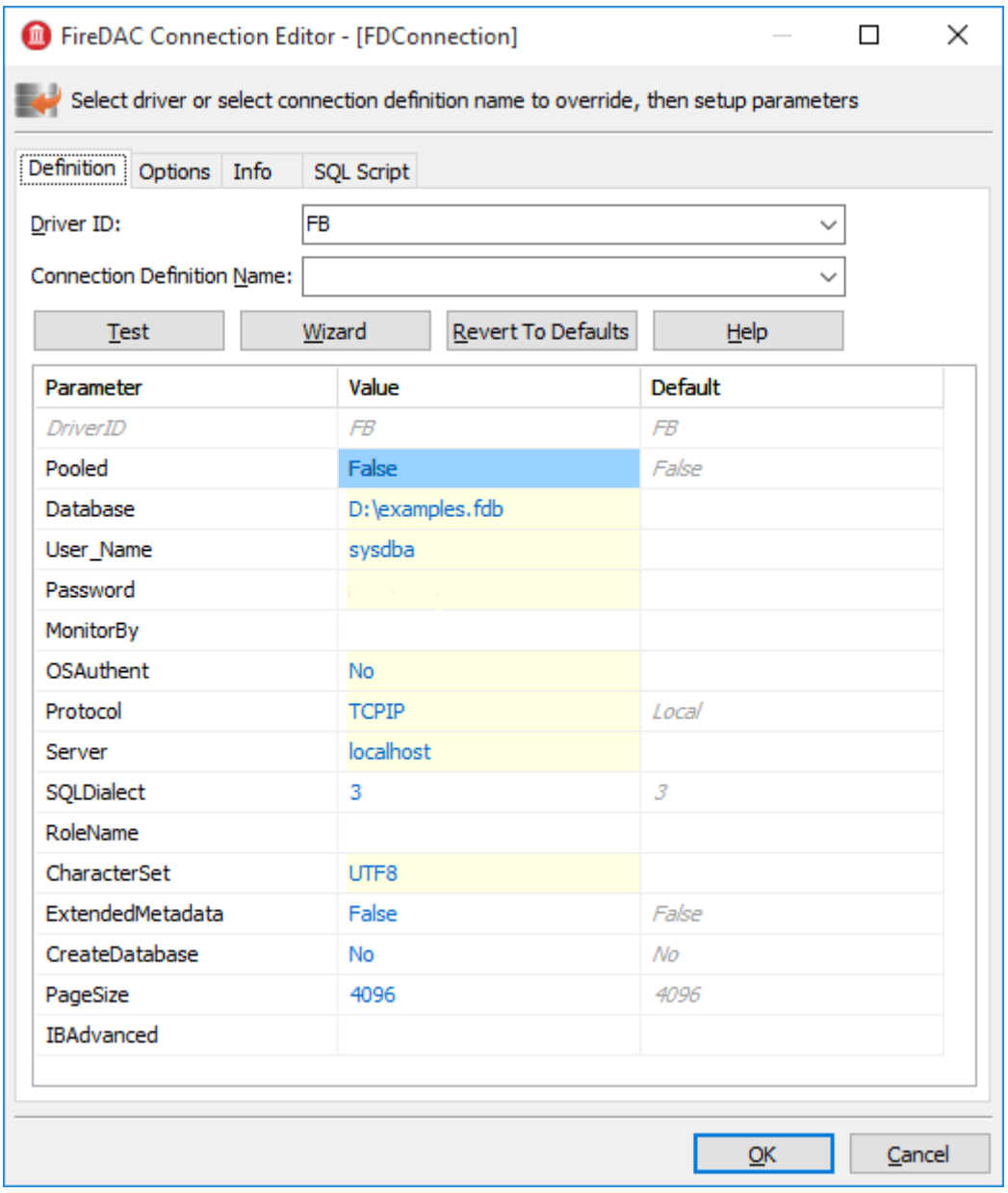

Рисунок 2. Параметры подключения

Дополнительные свойства:

Connected – управление подсоединением к БД, или проверка состояния соединения. Это свойство должно быть выставлено в True для работы мастеров других компонентов FireDac. Если ваше приложение должно запрашивать данные для авторизации, то важно не забыть сбросить это свойство в False перед компиляцией вашего приложения.

LoginPrompt – запрашивать ли имя пользователя и пароль при попытке соединения.

Transaction – компонент TFDTransaction, который будет использоваться в качестве умолчательного для выполнения различных операций TFDConnection. Если это свойство не назначено явно, TFDConnection создаст себе экземпляр

TFDTransaction самостоятельно, его параметры можно указать в свойстве TxOptions.

UpdateTransaction – компонент TFDTransaction, который будет использоваться в качестве умолчательного для одноимённых свойств компонентов TFDQuery. Если это свойство не назначено явно, будет использовано значение из свойства Transaction.

Поскольку параметры подключения, за исключением имени пользователя и пароля, обычно не изменяются в процессе эксплуатации приложения, мы будем считывать их из файла конфигурации.

```
// считываем параметры подключения
xIniFile := TIniFile.Create(xAppPath + 'config.ini');
try
  xIniFile.ReadSectionValues('connection', FDConnection.Params);
finally
  xIniFile.Free;
end;
```

Файл config.ini содержит примерно следующие строки:

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

Содержимое секции connection можно получить, скопировав содержимое свойства Params компонента TFDConnection после работы мастера.

Замечание

На самом деле общие настройки обычно находятся в %AppData%\Manufacture\AppName и сохраняются туда инсталлятором приложения. Однако при разработке удобно чтобы файл настроек лежал где-нибудь поближе, например, в папке с приложением.

Учтите, что если ваше приложение будет устанавливаться в папку Program Files и файл настройки будет лежать там же, то либо этот файл будет виртуализироваться в Program Data, либо будут проблемы с его модификацией и последующим чтением новых настроек.

Подключение к базе данных

Для подключения к базе данных необходимо изменить свойство Connected компонента TFDConnection в значение True или вызвать метод Open. В последний

метод можно передать имя пользователя и пароль в качестве параметров. В нашем приложении мы заменим стандартный диалог соединения с базой данных. Дадим возможность ошибиться при вводе регистрационной информации не более трёх раз, после чего приложение будет закрыто. Для этого напишем следующий код в обработчике события OnCreate главного датамодуля.

```
// делаем максимум 3 попытки входа в систему, потом закрываем приложение
xLoginCount := 0;
xLoginPromptDlg := TLoginPromptForm.Create(Self);
while (xLoginCount < MAX_LOGIN_COUNT) and
      (not FDConnection.Connected) do
begin
  try
    if xLoginPromptDlg.ShowModal = mrOK then
      FDConnection.Open(
        xLoginPromptDlg.UserName, xLoginPromptDlg.Password)
    else
      xLoginCount := MAX_LOGIN_COUNT;
  except
    on E: Exception do
    begin
      Inc(xLoginCount);
      Application.ShowException(E);
    end
  end;
end;
xLoginPromptDlg.Free;

if not FDConnection.Connected then
  Halt;
```

Компонент TFDTransaction

Компонент TFDTransaction предназначен для явной работы с транзакциями.

Клиентская часть Firebird допускает выполнение любых действий только в контексте транзакции. Поэтому если вы смогли получить доступ к данным без явного вызова TFDTransaction.StartTransaction, то значит где-то в недрах FireDac этот вызов произошёл автоматически. Такое поведение крайне не рекомендуется использовать. Для корректной работы приложений с базой данных желательно управлять транзакциями вручную, то есть явно вызывать методы StartTransaction, Commit и Rollback компонента TFDTransaction.

Таблица 2. Основные свойства компонента TFDTransacton

Параметр	Назначение
Connection	Связь с компонентом FDConnection.
Options.AutoCommit	Управляет автоматическим стартом и завершением транзакции. Значение по умолчанию True. Если значение этого свойства установлено в True, то FireDAC делает следующее: Запускается транзакция (если требуется) перед выполнением каждой SQL команды, и завершает транзакцию после выполнения SQL команды. Если

	команда выполнится успешно, то транзакция будет завершена как COMMIT, иначе — ROLLBACK. Если приложение вызывает метод StartTransaction, то автоматическое управление транзакциями будет отключено, до тех пор, пока транзакция не завершится как Commit или Rollback. В Firebird автоматическое управление транзакциями эмулируется самими компонентами FireDAC.
Options.AutoStart	Управляет автоматическим запуском транзакции. По умолчанию True.
Options.AutoStop	Управляет автоматическим завершением транзакции. По умолчанию True.
Options.DisconnectAction	Действие, которое будет выполнено при закрытии соединения, если транзакция активна. Значение по умолчанию xdCommit. Возможны следующие варианты: • xdNone – ничего не будет сделано. Действие будет отдано на откуп СУБД; • xdCommit – подтверждение транзакции; • xdRollback – откат транзакции. В других компонентах доступа значение по умолчанию для подобного свойства xdRollback. Поэтому необходимо выставлять это свойство вручную в то значние которое действительно требуется.
Options.EnableNested	Управляет вложенными транзакциями. Значение по умолчанию True. Когда транзакция активна, то следующий вызов StartTransaction создаст вложенную транзакцию. FireDAC эмулирует вложенные транзакции, используя точки сохранения, если СУБД не поддерживает вложенные транзакции в явном виде. Чтобы отключить вложенные транзакции, установите EnableNested в False и следующий вызов StartTransaction вызовет исключение. Firebird не поддерживает вложенные транзакции в явном виде.
Options.Isolation	Определяет уровень изолированности транзакции. Это самое важное свойство транзакции. Значение по умолчанию xiReadCommited. Возможны следующие варианты: • xiUnspecified – используется уровень изоляции по умолчанию для вашей СУБД (в Firebird это SNAPSHOT, т.е. с параметрами read write concurrency wait); • xiDirtyRead – этого уровня изолированности в Firebird не существует поэтому вместо него будет использован READ COMMITED; • xiReadCommited – уровень изолированности READ COMMITED. В Firebird такая транзакция стартует с параметрами read write read_committed rec_version nowait; • xiRepeatableRead – этого уровня

	изолированности в Firebird не существует поэтому вместо него будет использован SNAPSHOT; • xiSnapshot – уровень изолированности SNAPSHOT. В Firebird такая транзакция стартует с параметрами read write concurrency wait; • xiSerializable – уровень изолированности SERIALIZABLE. На самом деле в Firebird не существует транзакции с данным уровнем изолированности, но он эмулируется запуском транзакции с параметрами read write consistency wait.
Options.Params	Специфичные для СУБД параметры транзакции. В настоящее время используется только для Firebird и Interbase. Возможные значения: • read • write • read_committed • concurrency • consistency • wait • nowait • rec_version • no rec_version
Options.ReadOnly	Указывает является ли транзакция только для чтения. По умолчанию False. Если установлено в True, то любые изменения в рамках текущей транзакции невозможны, в Firebird в этом случае отсутствует значение read в параметрах транзакции. Установка этого свойства в True позволяет СУБД оптимизировать использование ресурсов.

В отличие от других СУБД в Firebird и Interbase разрешено использовать сколько угодно компонентов TFDTransaction привязанных к одному соединению. В нашем приложении мы будем использовать одну общую читающую транзакцию для всех справочников и оперативных журналов, и по одной пишущей транзакции на каждый справочник/журнал.

В нашем приложении мы не будем полагаться на автоматический старт и завершение транзакций, а потому во всех транзакциях Options.AutoCommit = False, Options.AutoStart = False и Options.AutoStop = False.

Поскольку читающая транзакция общая для всех справочников и журналов, то удобно разместить её в главном дата модуле. Для обычной работы (показ данных в гриде и т.п.) обычно используются режим изолированости READ COMMITED (Options.Isolation = xiReadCommited), т.к. он позволяет транзакции видеть чужие, committed изменения базы данных просто путём повторного выполнения запросов (перечитывания данных). Поскольку эта транзакция используется только для чтения, установим свойство Options.ReadOnly в значение True. Таким образом,

наша транзакция будет иметь параметры read read_commited rec_version. Транзакция с такими параметрами в Firebird может быть открытой сколь угодно долгое время (дни, недели, месяцы), без блокирования других транзакций или влияния на накопление мусора в базе данных (потому что на самом деле, на сервере такая транзакция стартует как committed).

Замечание

Такую транзакцию нельзя использовать для отчётов (особенно если они используют несколько последовательных запросов), потому что транзакция с режимом изолированности READ COMMITED во время перечитывания данных будет видеть все новые committed-изменения.

Для отчётов рекомендуется использовать короткую транзакцию только для чтения с режимом изолированности SNAPSHOT (Options.Isolation = xiSnapshot и Options.ReadOnly= True). В данном примере работа с отчётами не рассматривается.

Стартуем читающую транзакцию сразу после успешной установки соединения с базой данных, вызвав trRead.StartTransaction в событии OnCreate главного датамодуля, и завершаем перед закрытием соединения, вызвав tRead.Commit в событии OnDestroy главного датамодуля. Значение свойства Options.DisconnectAction равное xdCommit по умолчанию, подходит для транзакции только для чтения.

Пишущая транзакция будет отдельной для каждого справочника/журнала. Её мы разместим на форме, которая относится непосредственно к нужному журналу. Пишущая транзакция должна быть максимально короткой для того, чтобы не удерживать Oldest Active Transaction, которая не даёт собрать мусор, что в свою очередь приводит к деградации производительности. Поскольку пишущая транзакция очень короткая мы можем использовать уровень изолированности SNAPSHOT. Таким образом, наша пишущая транзакция будет иметь параметры Options.ReadOnly=False и Options.Isolation = xiSnapshot. Для пишущих транзакций значение свойства Options.DisconnectAction по умолчанию не подходит, его необходимо выставить в значение xdRollback.

Датасеты

Работать с данными в FireDac можно при помощи компонент FDQuery, FDTable, FDStoredProc, FDCommand, но FDCommand не является датасетом.

TFDQuery, TFDTable и TFDStoredProc унаследованы от TFDRdbmsDataSet. Помимо наборов данных для работы непосредственно с базой данных, в FireDac

существует также компонент TFDMemTable, который предназначен для работы с набором данных в памяти, является аналогом TClientDataSet.

Основным компонентом для работы с наборами данных является TFDQuery. Возможностей этого компонента хватает практически для любых целей. Компоненты TFDTable и TFDStoredProc всего лишь модификации, либо чуть расширенные, либо усеченные. Мы не будем их рассматривать и применять в нашем приложении. При желании вы можете ознакомиться с ними в документации по FireDac.

Назначение компонента — буферизация записей, выбираемых оператором SELECT, для представления этих данных в Grid, а также для обеспечения "редактируемости" записи (текущей в буфере (гриде)). В отличие от компонента IBX.IBDataSet компонент FDQuery не содержит свойств RefreshSQL, InsertSQL, UpdateSQL и DeleteSQL. Вместо этого «редактируемость» обеспечивается компонентом FDUpdateSQL, который устанавливается в свойство UpdateObject.

Замечание

В ряде случаев можно сделать компонент FDQuery редактируемым без установки свойства UpdateObject и прописывания запросов Insert/Update/Delete, просто установив свойство UpdateOptions.RequestLive = True, при этом модифицирующие запросы будут сгенерированы автоматически. Однако такой подход имеет множество ограничений на основной SELECT запрос, поэтому не стоит полагаться на него.

Таблица 3. Основные свойства компонента TFDQuery.

Параметр	Назначение
Connection	Связь с компонентом FDConnection.
MasterSource	Ссылка на Master-источник данных (TDataSource) для FDQuery, используемого в качестве Detail.
Transaction	Транзакция, в рамках которой будет выполняться запрос, прописанный в свойстве SQL. Если свойство не указано будет использоваться транзакция по умолчанию для подключения.
UpdateObject	Связь с компонентом FDUpdateSQL, который обеспечивает «редактируемость» набора данных, когда SELECT запрос не отвечает требованиям для автоматического формирования модифицирующих запросов при установке UpdateOptions.RequestLive = True.
UpdateTransaction	Транзакция, в рамках которой будут выполняться модифицирующие запросы. Если свойство не указано, будет использована транзакция из свойства

	Transaction.
UpdateOptions.CheckRequired	Если свойство CheckRequired установлено в True, то FireDac контролирует свойство Required соответвующих полей, т.е. полей с ограничением NOT NULL. По умолчанию установлено в True. Если CheckRequired=True и в поле имеющее свойство Required=True не присвоено значение, то при вызове метода Post будет возбуждено исключение. Это может быть нежательно в том случае, если значение этого поля может быть присвоено позже в BEFORE триггерах.
UpdateOptions.EnableDelete	Определяет, позволяется ли удаление записи из набора данных. Если EnableDelete=False, то при вызове метода Delete будет возбуждено исключение.
UpdateOptions.EnableInsert	Определяет, позволяется ли вставка записи в набор данных. Если EnableInsert=False, то при вызове метода Insert/Append будет возбуждено исключение.
UpdateOptions.EnableUpdate	Определяет, позволяется ли изменение записи в наборе данных. Если EnableUpdate=False, то при вызове метода Edit будет возбуждено исключение.
UpdateOptions.FetchGeneratorsPoint	Управляет моментом получения следующего значения генератора указанного в свойстве UpdateOptions.GeneratorName или свойстве GeneratorName автоинкрементного поля AutoGenerateValue = arAutoInc. Имеет следующие варианты значений: • gpNone – значение генератора не извлекается; • gpImmediate – следующее значение генератора извлекается сразу после вызова метода Insert/Append; • gpDeffered – следующее значение генератора извлекается до публикации новой записи в базе данных, т.е. во время выполнения методов Post или ApplyUpdates. Значение по умолчанию gpDeffered.
UpdateOptions.GeneratorName	Имя генератора для извлечения следующего значения автоинкрементного поля.
UpdateOptions.ReadOnly	Указывает, является ли набор данных только для чтения. По умолчанию False. Если значение этого свойства установлено в True, то значения свойств EnableDelete, EnableInsert и EnableUpdate будут автоматически выставлены в False.
UpdateOptions.RequestLive	Установка RequestLive в True делает запрос «живым», т.е. редактируемым, если это

	возможно. При этом запросы Insert/Update/Delete будут сгенерированы автоматически. Эта опция накладывает множество ограничений на SELECT запрос, введена для обратной совместимости с BDE и не рекомендуется.

UpdateOptions.UpdateMode	Отвечает за проверку модификации записи. Это свойство позволяло контролировать возможное "перекрытие" обновлений для случаев, когда пользователь выполняет редактирование записи "долго", а другой пользователь может успеть отредактировать эту же запись и сохранить её раньше. То есть, первый пользователь на этапе редактирования даже не будет знать, что запись уже изменилась, возможно не один раз, и сумеет "затереть" эти обновления своим:
	• upWhereAll – проверка на существование записи по первичному ключу + проверка всех столбцов на старые значения. Например
	`update table set ...`
	`where pkfield = :old_ pkfield and client_name = :old_client_name and info = :old_info ...`
	То есть, в данном случае запрос поменяет информацию в записи только в том случае, если запись до нас никто не успел изменить. Особенно это важно, если существуют взаимозависимости между значениями столбцов — например, минимальная и максимальная зарплата, и т.п.
	• upWhereCahnged – проверка записи на существование по первичному ключу + плюс проверка на старые значения только изменяемых столбцов.
	`update table set ...`
	`where pkfield = :old_pkfield and client_name = :old_client`
	• upWhereKeyOnly (по умолчанию) – проверка записи на существование по первичному ключу.
	Последняя проверка соответствует генерируемому автоматически для

	UpdateSQL запросу. Поэтому, при возможных конфликтах обновлений в многопользовательской среде необходимо дописывать условия к where самостоятельно. И, разумеется, также необходимо при реализации аналога upWhereChanged удалять лишние изменения столбцов в update table set ... - то есть, оставлять в перечне set только действительно изменённые столбцы, иначе запрос перепишет чужие обновления этой записи. Как вы понимаете, это означает необходимость динамического конструирования запроса UpdateSQL. Если вы хотите задать настройки обнаружения конфликтов обновления индивидуально для каждого поля, то вы можете воспользоваться свойством ProviderFlags для каждого поля.
CachedUpdates	Определяет, будет ли набор данных кэшировать изменения без немедленного внесения их в базу данных. Если это свойство установлено в значение True, то любые изменения (Insert/Post, Update/Post, Delete) вносятся в базу данных не сразу, а сохраняется в специальном журнале. Приложение должно явно применить изменения, вызвав метод ApplyUpdates. В этом случае все изменения будут выполнены в течение малого промежутка времени и в одной короткой транзакции. По умолчанию значение этого свойства False.
SQL	Содержит SQL запрос. Если это свойство содержит SELECT запрос, то его необходимо выполнять методом Open. В противном случае необходимо использовать методы Execute или ExecSQL.

Компонент TFDUpdateSQL

Компонент TFDUpdateSQL позволяет переопределять SQL команды, сгенерированные для автоматического обновления набора данных. Он может быть использован для внесения обновлений в компоненты TFDQuery, TFDTable и TFDStoredProc. Использование TFDUpdateSQL является необязательным для компонентов TFDQuery и TFDTable, потому что эти компоненты способны автоматически генерировать команды для публикации обновлений из набора данных в СУБД. Использование TFDUpdateSQL является обязательным для возможности обновления набора данных TFDStoredProc. Рекомендуем применять его всегда, даже для самых простых случаев, чтобы получать полный контроль над тем какие запросы выполняются в вашем приложении.

Для того чтобы указать SQL команды на этапе проектирования, используйте редактор TFDUpdateSQL времени проектирования, который вызывается двойным щелчком по компоненту.

Замечание

Для работы многих редакторов времени проектирования FireDac требуется, чтобы было активно подключение к базе данных (TFDConnection.Connected = True) и транзакция находилась в режиме автостарта (TFDTransaction.Options.AutoStart = True). Но такие настройки могут мешать при работе приложения. Например, пользователь должен входить в программу под своим логином, а TFDConnection подключается к базе данных под SYSDBA. Поэтому после каждого использования редакторов времени проектирования рекомендуем проверять свойство TFDConnection.Connected и сбрасывать его. Кроме того, вам придётся включать и выключать автостарт транзакции предназначенной только для чтения.

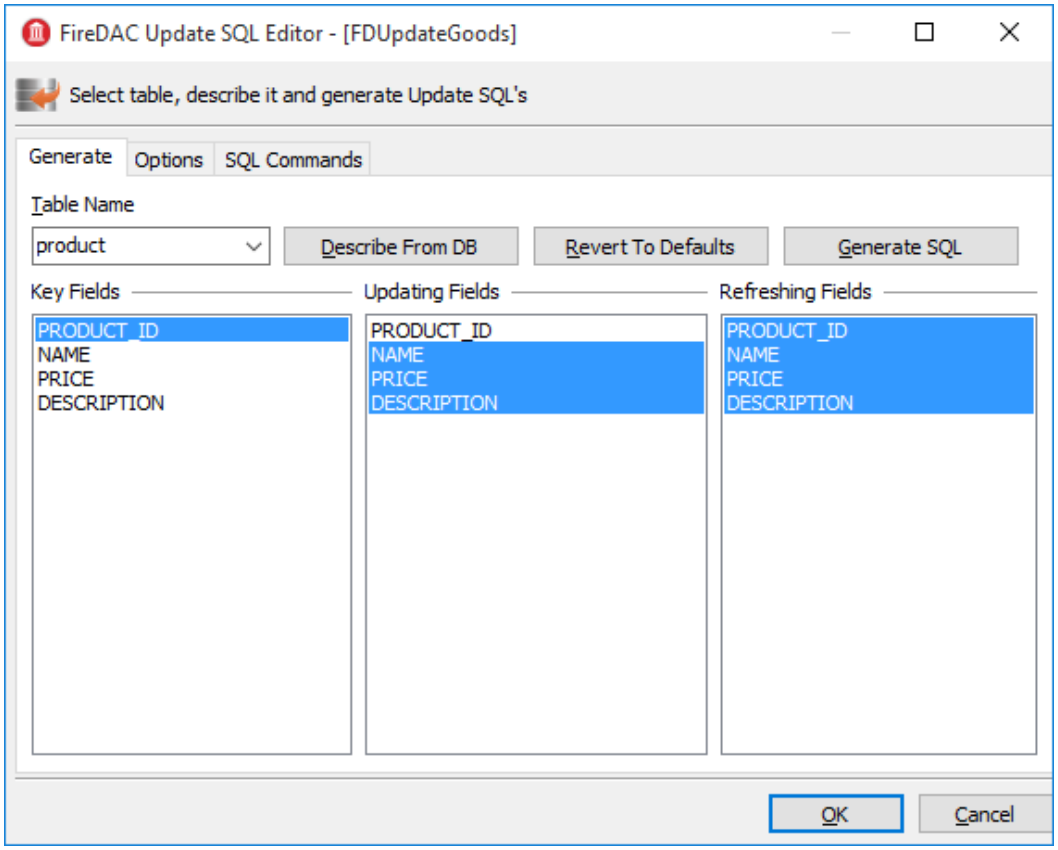

Рисунок 3. Настройка TFDUpdateSQL. Генерация запросов.

На закладке Generate вы можете упростить себе задачу по написанию Insert/Update/Delete/Refresh запросов. Для этого выберете таблицу для обновления, её ключевые поля, поля для обновления, и поля которые будут перечитаны после обновления, и нажмите на кнопку «Generate SQL».

После чего запросы будут сгенерированы автоматически, и вы перейдёте на закладку «SQL Commands», где можете поправить каждый из запросов.

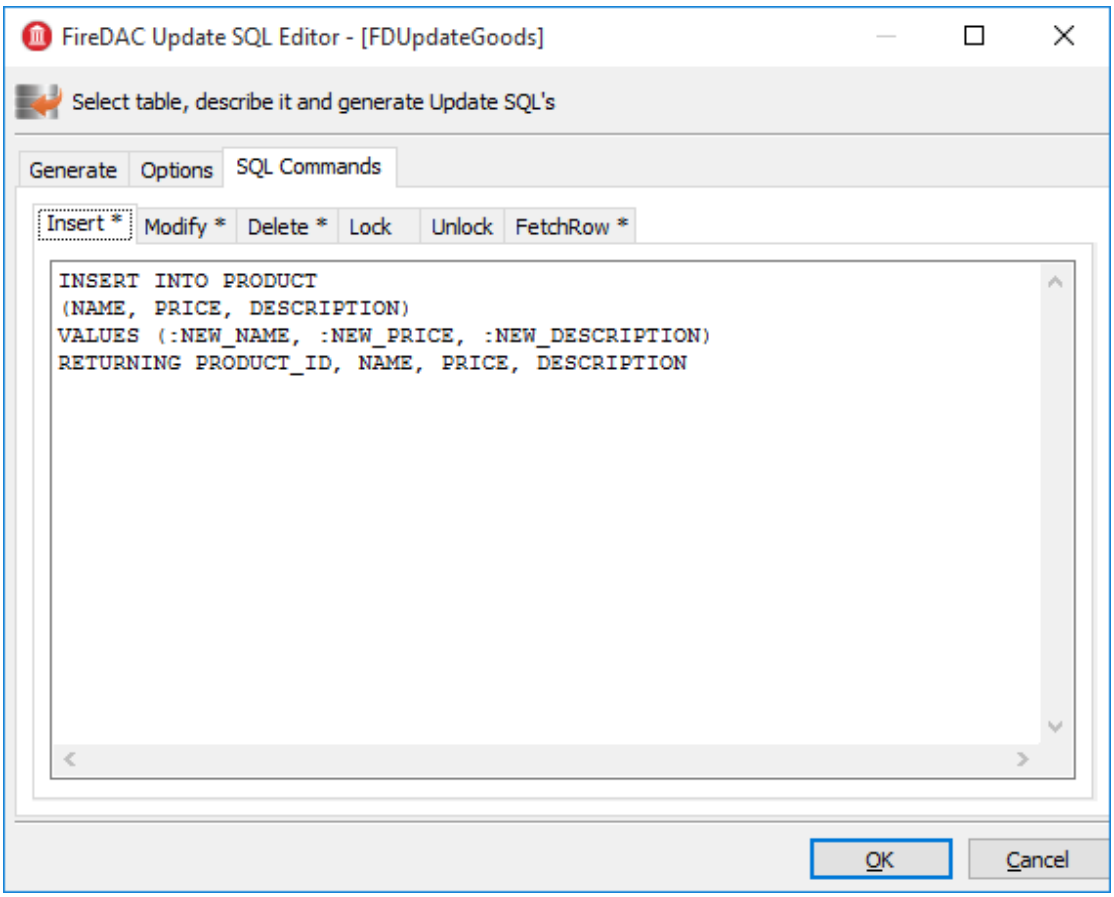

Рисунок 4. Настройка TFDUpdateSQL. SQL команды.

Замечание

Поскольку product_id не включено в Updating Fields, оно отсутствует в генерируемом запросе insert. Предполагается, что этот столбец заполняется автоматически триггером (с генератором), или же этот это IDENTITY столбец (начиная с Firebird 3.0). При получении значения генератора для этого столбца с сервера, рекомендуется вручную добавить столбец PRODUCT_ID в предложение RETURNING оператора INSERT.

На закладке Options находятся некоторые свойства, которые могут повлиять на генерацию запросов. Эти свойства не относятся к самому компоненту TFDUpdateSQL, а являются ссылками на свойства UpdateOptions набора данных, у которого указан текущий TFDUpdateSQL в свойстве UpdateObject. Так сделано исключительно ради удобства.

Таблица 4. Основные свойства компонента TFDUpdateSQL.

Параметр	Назначение
Connection	Связь с компонентом FDConnection.
DeleteSQL	SQL запрос для удаления записи.
FetchRowSQL	SQL запрос для возврата одной текущей (обновлённой, вставленной) записи. (RefreshSQL)

InsertSQL	SQL запрос для вставки записи.
LockSQL	SQL запрос для блокировки одной текущей записи. (FOR UPDATE WITH LOCK).
ModifySQL	SQL запрос для модификации записи.
UnlockSQL	SQL запрос для разблокировки текущей записи. В Firebird не применяется.

Как вы уже заметили, у компонента TFDUpdateSQL нет свойства Transaction. Это потому, что компонент не выполняет модифицирующие запросы непосредственно, а лишь заменяет автоматически сгенерированные запросы в наборе данных, который является предком TFDRdbmsDataSet.

Компонент TFDCommand

Компонент TFDCommand предназначен для выполнения SQL запросов. Он не является предком TDataSet, а потому удобен лишь для выполнения SQL запросов, не возвращающих набор данных.

Таблица 5. Основные свойства компонента TFDCommand.

Параметр	Назначение
Connection	Связь с компонентом FDConnection.
Transaction	Транзакция, в рамках которой будет выполняться SQL команда.
CommandKind	Тип команды. • skUnknown – неизвестен. В этом случае тип команды будет определятся автоматически по тексту команды внутренним парсером; • skStartTransaction – команда для старта транзакции; • skCommit – команда завершения и подтверждения транзакции; • skRollback – команда завершения и отката транзакции; • skCreate – команда CREATE … для создания нового объекта метаданных; • skAlter – команда ALTER … для модификации объекта метаданных; • skDrop – команда DROP … для удаления объекта метаданных; • skSelect – команда SELECT для выборки данных; • skSelectForLock – команда SELECT … WITH LOCK для блокировки выбранных строк; • skInsert – команда INSERT … для вставки новой записи; • skUpdate – команда UPDATE … для модификации записей;

	• skDelete – команда DELETE … для удаления записей; • skMerge – команда MERGE INTO … • skExecute – команда EXECUTE PROCEDURE или EXECUTE BLOCK; • skStoredProc – вызов хранимой процедуры; • skStoredProcNoCrs – Вызов хранимой процедуры не возвращающей курсор; • skStoredProcWithCrs – вызов хранимой процедуры возвращающей курсор. Обычно тип команды определяется автоматически по тексту SQL запроса.
CommandText	Текст SQL запроса.

Создание справочников

В нашем приложении мы создадим два справочника: справочник товаров и справочник заказчиков. Каждый из справочников представляет собой форму с сеткой TDBGrid, источником данных TDataSource, набором данных TFDQuery, пишущей транзакции TFDTransaction.

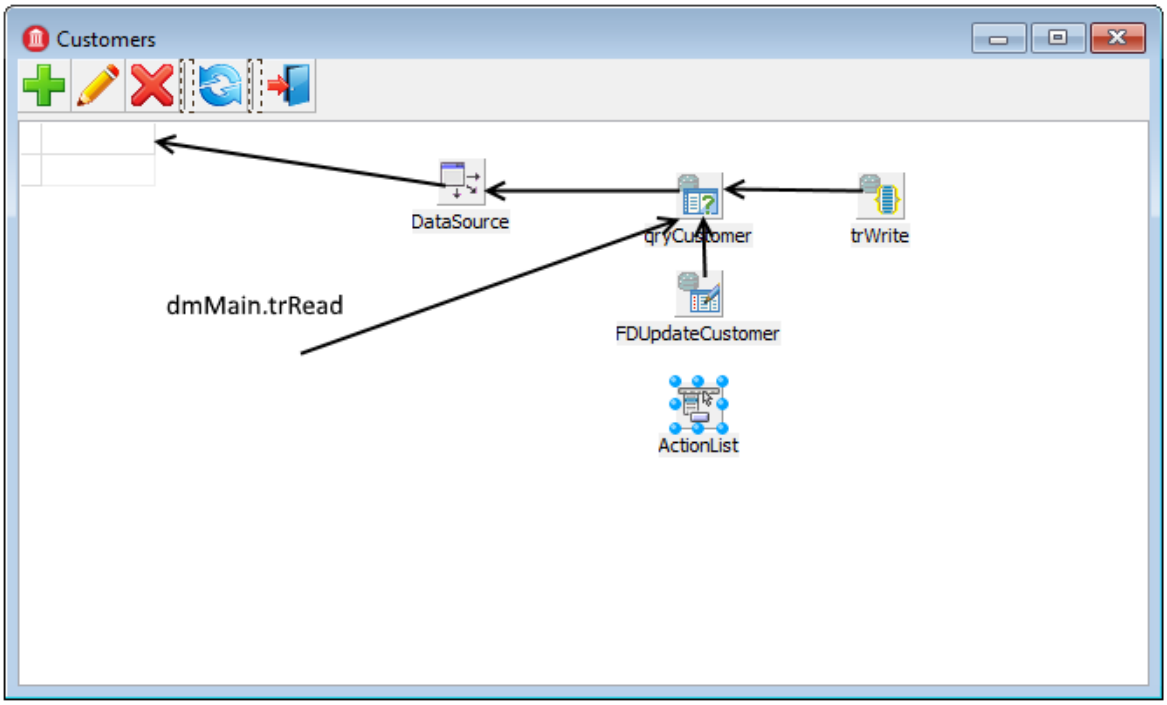

Рисунок 5. Форма справочника Customers

Замечание

Компонент trRead не виден, потому что находится не на форме, а в модуле dmMain.

Рассмотрим создание справочников на примере справочника заказчиков.

Разместим компонент TFDQuery на форме с именем qryCustomers. Этот набор данных будет указан в свойстве DataSet источника данных DataSource. В свойстве Transaction укажем ReadOnly транзакцию trRead, которая была создана в главном датамодуле проекта. В свойстве UpdateTransaction указываем транзакцию trWrite, в свойстве Connection — соединение расположенное в главном датамодуле. В свойстве SQL напишем следующий запрос:

```
SELECT
    customer_id,
    name,
    address,
    zipcode,
    phone
FROM
    customer
ORDER BY name
```

Пишущая транзакция trWrite должна быть максимально короткой, и иметь режим изолированности SNAPSHOT. Мы не будем полагаться на автоматический старт и завершение транзакции, а будем стартовать и завершать транзакцию явно. Таким образом, наша транзакция должна иметь следующие свойства:

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

На самом деле необязательно устанавливать режим изолированности SNAPSHOT для простых INSERT/UPDATE/DELETE. Однако если у таблицы есть сложные триггеры, или вместо простых запросов INSERT/UPDATE/DELETE вызывается хранимая процедура, то желательно использовать уровень изолированности SNAPSHOT.

Дело в том, что уровень изолированности READ COMMITED не обеспечивает атомарности оператора в пределах одной транзакции (statement read consistency). Таким образом, оператор SELECT может возвращать данные, которые попали в базу данных после начала выполнения запроса. В принципе режим изолированности SNAPSHOT можно рекомендовать почти всегда, если транзакция будет короткой.

Для возможности редактирования набора данных необходимо заполнить свойства InsertSQL, ModifySQL, DeleteSQL и FetchRowSQL. Эти свойства могут быть сгенерированы мастером, но после этого может потребоваться некоторая правка. Например вы можете дописать предложение RETURNING, удалить модификацию

некоторых столбцов, или же вовсе заменить автоматически сгенерированный запрос на вызов хранимой процедуры.

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
```

В этом справочнике будем получать значение генератора перед вставкой записи в таблицу. Для этого необходимо установить значение свойств компонента TFDQuery в следующие значения UpdateOptions.GeneratorName = GEN_CUSTOMER_ID и UpdateOptions.AutoIncFields = CUSTOMER_ID. Есть другой способ, когда значение генератора (автоинкрементного поля) возвращается после выполнения INSERT запроса с помощью предложения RETURNING. Этот способ будет показан позже.

Для добавления новой записи и редактирования существующей принято использовать модальные формы, по закрытию которых с результатом mrOK изменения вносятся в базу данных. Обычно для создания таких форм используются DBAware компоненты, которые позволяют отображать значения некоторого поля в текущей записи и немедленно вносить изменения в текущую

запись набора данных в режимах Insert/Edit, т.е. до Post. Но перевести набор данных в режим Insert/Edit можно только стартовав пишущую транзакцию. Таким образом, если кто-то откроет форму для внесения новой записи и уйдёт на обед, не закрыв эту форму, у нас будет висеть активная транзакция до тех пор, пока сотрудник не вернётся с обеда и не закроет форму. Это в свою очередь приведёт к тому, что активная транзакция будет удерживать сборку мусора, что позже приведёт к снижению производительности. Эту проблему можно решить одним из двух способов:

1. Использовать режим CachedUpdates, что позволяет держать транзакцию активной только на очень короткий промежуток времени, а именно на время внесения изменений.
2. Отказаться от применения DBAware компонентов. Однако этот путь потребует от вас дополнительных усилий.

Мы покажем применение обоих способов. Для справочников гораздо удобнее использовать первый способ. Рассмотрим код редактирования записи поставщика

```
procedure TCustomerForm.actEditRecordExecute(Sender: TObject);
var
  xEditor: TEditCustomerForm;
begin
  xEditor := TEditCustomerForm.Create(Self);
  try
    xEditor.OnClose := CustomerEditorClose;
    xEditor.CustomerForm := Self;
    xEditor.Caption := 'Edit customer';
    qryCustomer.CachedUpdates := True;
    qryCustomer.Edit;
    xEditor.ShowModal;
  finally
    xEditor.Free;
  end;
end;
```

По коду видно, что перед переводом набора данных в режим редактирования мы устанавливаем ему режим CachedUpdates, а вся логика обработки редактирования происходит в модальной форме.

```
procedure TCustomerForm.CustomerEditorClose (Sender: TObject;
  var Action: TCloseAction);
begin
  if TForm(Sender).ModalResult <> mrOK then
  begin
    // отменяем все изменения
    qryCustomer.Cancel;
    qryCustomer.CancelUpdates;
    // возвращаем набор данных в обычный режим обновления
    qryCustomer.CachedUpdates := False;
    // и позволяем закрыть форму
    Action := caFree;
    Exit;
  end;
```

```
  try
    // подтверждаем изменения на уровне набора данных
    qryCustomer.Post;
    // стартуем транзакцию
    trWrite.StartTransaction;
    // если в наборе данных есть изменения
    if (qryCustomer.ApplyUpdates = 0) then
    begin
      // записываем их в БД
      qryCustomer.CommitUpdates;
      // и подтверждаем транзакцию
      trWrite.Commit;
    end
    else begin
      raise Exception.Create(qryCustomer.RowError.Message);
    end;
    qryCustomer.CachedUpdates := False;
    Action := caFree;
  except
    on E: Exception do
    begin
      // откатываем транзакцию
      if trWrite.Active then
        trWrite.Rollback;
      Application.ShowException(E);
      // Не закрываем окно, даём возможность исправить ошибку
      Action := caNone;
    end;
  end;
end;
```

Из кода видно, что до тех пор, пока кнопка OK не нажата, пишущая транзакция не стартует вовсе. Таким образом, пишущая транзакция активна только на время переноса данных из буфера набора данных в базу данных. Поскольку мы копим в буфере не более одной записи, транзакция будет активна очень короткое время, что и требовалось.

Справочник товаров делается аналогично справочнику заказчиков. Однако в нём мы продемонстрируем другой способ получения автоинкрементных значений.

Основной запрос будет выглядеть следующим образом:

```
SELECT
    product_id,
    name,
    price,
    description
FROM product
ORDER BY name
```

Свойство компонента TFDUpdateSQL.InsertSQL будет содержать следующий запрос:

```
INSERT INTO PRODUCT
(NAME, PRICE, DESCRIPTION)
VALUES (:NEW_NAME, :NEW_PRICE, :NEW_DESCRIPTION)
RETURNING PRODUCT_ID
```

В этом запросе появилось предложение RETURNING, которое вернёт значение поля `PRODUCT_ID` после изменения его в BEFORE INSERT триггере. В этом случае не имеет смысла выставлять значение свойства UpdateOptions.GeneratorName. Кроме того, полю `PRODUCT_ID` необходимо выставить свойства Required = False и ReadOnly = True, поскольку значение этого свойства не вносится напрямую. В остальном всё примерно также как это организовано для справочника производителей.

Создание журналов

В нашем приложении будет один журнал «Счёт-фактуры». В отличие от справочников журналы содержат довольно большое количество записей и являются часто пополняемыми.

Счёт-фактура – состоит из заголовка, где описываются общие атрибуты (номер, дата, заказчик …), и строк счёт-фактуры со списком товаром, их количеством, стоимостью и т.д. Для таких документов удобно иметь два грида: в главном отображаются данные о шапке документа, а в детализирующем - список товаров. Таким образом, на форму документа нам потребуется поместить два компонента TDBGrid, к каждому из которых привязать свой TDataSource, которые в свою очередь будут привязаны к своим TFDQuery. В нашем случае набор данных с шапками документы будет называться qryInvoice, а со строками документа qryInvoiceLine.

В свойстве Transaction обоих наборов данных укажем ReadOnly транзакцию trRead, которая была создана в главном датамодуле проекта. В свойстве UpdateTransaction указываем транзакцию trWrite, в свойстве Connection — соединение, расположенное в главном датамодуле.

Большинство журналов содержат поле с датой создания документа. Чтобы уменьшить количество выбираемых данных обычно принято вводить такое понятие как рабочий период для того, чтобы уменьшить объём данных передаваемый на клиента. Рабочий период – это диапазон дат, внутри которого требуются рабочие документы. Поскольку приложение может содержать более одного журнала, то имеет смысл разместить переменные, содержащие дату начала и окончания рабочего периода, в глобальном датамодуле dmMain, который, так или иначе, используется всеми модулями, работающими с БД. При старте приложения рабочий период обычно инициализируется датой начала и окончания текущего квартала (могут быть другие варианты). В ходе работы приложения можно изменить рабочий период по желанию пользователя.

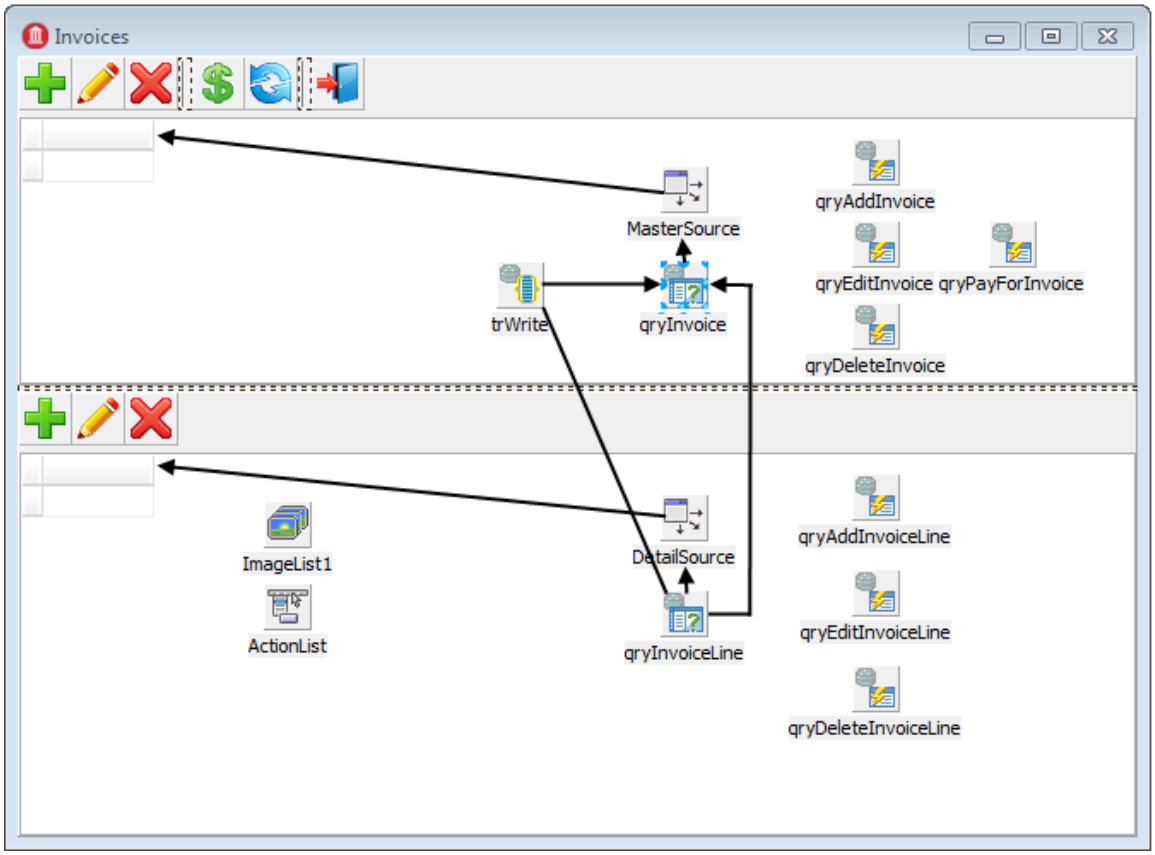

Рисунок 6. Форма журнала Invoices

Поскольку чаще всего требуются именно последние введённые документы, то имеет смысл сортировать их по дате в обратном порядке. С учётом вышесказанного, в свойстве SQL набора данных qryInvoice запрос будет выглядеть следующим образом:

```
SELECT
  invoice.invoice_id AS invoice_id,
  invoice.customer_id AS customer_id,
  customer.NAME AS customer_name,
  invoice.invoice_date AS invoice_date,
  invoice.total_sale AS total_sale,
  IIF(invoice.payed=1, 'Yes', 'No') AS payed
FROM
  invoice
  JOIN customer ON customer.customer_id = invoice.customer_id
WHERE invoice.invoice_date BETWEEN :date_begin AND :date_end
ORDER BY invoice.invoice_date DESC
```

При открытии этого набора данных необходимо будет инициализировать параметры запроса:

```
  qryInvoice.ParamByName('date_begin').AsSqlTimeStamp := dmMain.BeginDateSt;
  qryInvoice.ParamByName('date_end').AsSqlTimeStamp := dmMain.EndDateSt;
  qryInvoice.Open;
```

Все операции над счёт-фактурой будем производить с помощью хранимых процедур, хотя в более простых случаях это можно делать и с помощью обычных запросов INSERT/UPDATE/DELETE.

Каждую хранимую процедуру будем выполнять как отдельный запрос в компонентах TFDCommand. Этот компонент не является предком TFDRdbmsDataSet, не буферизирует данные и возвращает максимум одну строку результата, поэтому его использование несёт меньше накладных расходов для запросов, не возвращающих данные. Поскольку наши хранимые процедуры выполняют модификацию данных, то свойство Transaction компонентов TFDCommand необходимо установить транзакцию trWrite.

Замечание

Хранимые процедуры вставки, редактирования и добавления записи можно также разместить в соответствующих свойствах компонента TFDUpdateSQL.

Для работы с шапкой счёт-фактуры предусмотрено четыре операции: добавление, редактирование, удаление и установка признака «оплачено». Как только счёт-фактура оплачена, мы запрещаем любые её модификации, как в шапке, так и в строках. Это сделано на уровне хранимых процедур. Приведём тексты запросов для вызова хранимых процедур.

qryAddInvoice.CommandText:

```
EXECUTE PROCEDURE sp_add_invoice(
  NEXT VALUE FOR gen_invoice_id,
  :CUSTOMER_ID,
  :INVOICE_DATE
)
```

qryEditInvoice.CommandText:

```
EXECUTE PROCEDURE sp_edit_invoice(
  :INVOICE_ID,
  :CUSTOMER_ID,
  :INVOICE_DATE
)
```

qryDeleteInvoice.CommandText:

```
EXECUTE PROCEDURE sp_delete_invoice(:INVOICE_ID)
```

qryPayForInvoice.CommandText:

```
EXECUTE PROCEDURE sp_pay_for_inovice(:invoice_id)
```

Поскольку наши хранимые процедуры вызываются не из компонента TFDUpdateSQL, то после их выполнения необходимо вызвать `qryInvoice.Refresh` для обновления данных в гриде.

Вызов хранимых процедур, для которых не требуется ввод данных, производится следующим образом:

```
if MessageDlg('Вы действительно хотите удалить счёт фактуру?', mtConfirmation,
  [mbYes, mbNo], 0) = mrYes then
begin
  // Стартуем транзакцию
  trWrite.StartTransaction;
  try
    qryDeleteInvoice.ParamByName('INVOICE_ID').AsInteger :=
      qryInvoice.FieldByName('INVOICE_ID').AsInteger;
    // выполнение хранимой процедуры
    qryDeleteInvoice.Execute;
    // подтверждение транзакции
    trWrite.Commit;
    // обновление данных в гриде
    qryInvoice.Refresh;
  except
    on E: EFDDBEngineException do
    begin
      if trWrite.Active then
        trWrite.Rollback;
      Application.ShowException(E);
    end;
  end;
end;
```

Для добавления новой записи и редактирования существующей, как и в случае со справочниками мы будем использовать модальные формы. В данном случае мы не будем использовать DBAware компоненты. Ещё одна особенность - для выбора заказчика мы будем использовать компонент TButtonedEdit. Он будет отображать наименование текущего заказчика, а по нажатию кнопки вызывать модальную форму с гридом для выбора заказчика. Конечно, можно было бы воспользоваться чем-то вроде TDBLookupCombobox, но, во-первых заказчиков может быть очень много и пролистывать такой выпадающий список будет неудобно, во-вторых для поиска нужного заказчика одного названия может быть недостаточно.

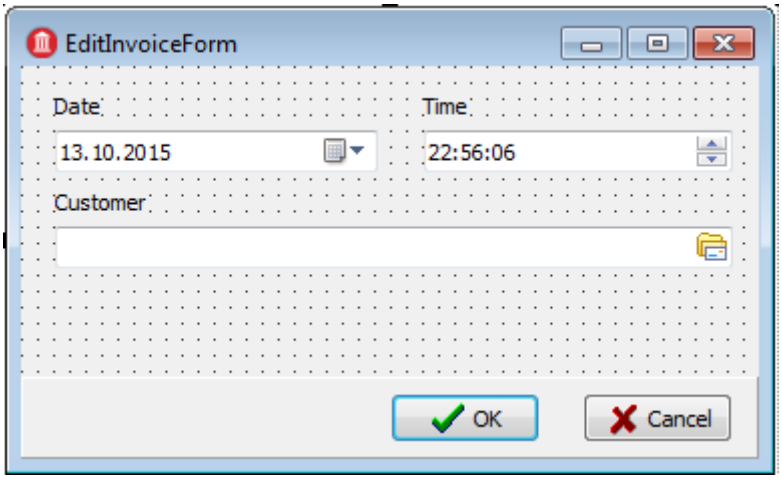

Рисунок 7. Форма редактирования счёт-фактуры

В качестве модальной окна для выбора заказчика используем ту же форму, что была создана для ввода заказчиков. Код обработчика нажатия кнопки в компоненте TButtonedEdit будет выглядеть следующим образом:

```
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.qryCustomer.FieldByName('CUSTOMER_ID')
        .AsInteger;
      edtCustomer.Text :=
xSelectForm.qryCustomer.FieldByName('NAME').AsString;
    end;
  finally
    xSelectForm.Free;
  end;
end;
```

Поскольку мы используем не DBAware компоненты, то при вызове формы редактирования нам будет необходимо инициализировать код заказчика и его наименование для отображения.

```
procedure TInvoiceForm.actEditInvoiceExecute(Sender: TObject);
var
  xEditor: TEditInvoiceForm;
begin
  xEditor := TEditInvoiceForm.Create(Self);
  try
    xEditor.InvoiceForm := Self;
    xEditor.EditMode := emInvoiceEdit;
    xEditor.Caption := 'Редактирование счёт-фактуры';

    xEditor.InvoiceId := qryInvoice.FieldByName('INVOICE_ID').AsInteger;
    xEditor.SetCustomer(qryInvoice.FieldByName('CUSTOMER_ID').AsInteger,
                        qryInvoice.FieldByName('CUSTOMER_NAME').AsString);
    xEditor.InvoiceDate := qryInvoice.FieldByName('INVOICE_DATE').AsDateTime;

    xEditor.ShowModal;
  finally
    xEditor.Free;
  end;
end;

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

Обработку добавления новой счёт-фактуры и редактирование существующей будем осуществлять в событии закрытия модальной формы, также, как это сделано для справочников. Однако здесь мы уже не будем переводить набор данных в режим CachedUpdates, поскольку модификация производится с помощью хранимых процедур, и мы не используем DBAware компоненты.

```
procedure TEditInvoiceForm.FormClose(Sender: TObject; var Action: TCloseAction);
var
  xCustomerId: Integer;
begin
  // если форма закрыта не по нажатию кнопки OK,
  // то вообще ничего не делаем. Транзакция не стартует.
  if ModalResult <> mrOK then
  begin
    Action := caFree;
    Exit;
  end;

  // Выполняем всё в короткой транзакции
  FInvoiceForm.trWrite.StartTransaction;
  try
    if FEditMode = emInvoiceAdd then
    begin
      // если FcustomerId = 0, то прамаетру CUSTOMER_ID выставляем значение NULL
      if FCustomerId <> 0 then
        FInvoiceForm.qryAddInvoice.ParamByName('CUSTOMER_ID').AsInteger :=
          FCustomerId
      else
        FInvoiceForm.qryAddInvoice.ParamByName('CUSTOMER_ID').Clear;

      FInvoiceForm.qryAddInvoice.ParamByName('INVOICE_DATE').AsSQLTimeStamp :=
        DateTimeToSQLTimeStamp(FInvoiceDate);
      // выполяем хранимую процедуру для вставки записи
      FInvoiceForm.qryAddInvoice.Execute();
    end;
    if FEditMode = emInvoiceEdit then
    begin
      FInvoiceForm.qryEditInvoice.ParamByName('INVOICE_ID').AsInteger :=
        FInvoiceId;
      FInvoiceForm.qryEditInvoice.ParamByName('CUSTOMER_ID').AsInteger :=
        FCustomerId;
      FInvoiceForm.qryEditInvoice.ParamByName('INVOICE_DATE').AsSQLTimeStamp :=
        DateTimeToSQLTimeStamp(FInvoiceDate);
      // выполняем хранимую процедуру для редактирования записи
      FInvoiceForm.qryEditInvoice.Execute();
    end;
    // подтверждение транзакции
    FInvoiceForm.trWrite.Commit;
    // обновление данных в гриде
    FInvoiceForm.qryInvoice.Refresh;

    Action := caFree;
  except
    on E: Exception do
    begin
      if FInvoiceForm.trWrite.Active then
        FInvoiceForm.trWrite.Rollback;
      Application.ShowException(E);
      // Не закрываем модальное окно. Даём пользователю возможность
      // исправить ошибку
      Action := caNone;
    end;
  end;
end;
```

Теперь перейдём к позициям накладной. Набору данных qryInvoiceLine установим свойство MasterSource = MasterSource, который привязан к qryInvoice, а свойство MasterFields = INVOICE_ID. В свойстве SQL напишем следующий запрос:

```
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
```

Все модификации, как и в случае с шапкой счёт-фактуры, будем осуществлять с помощью хранимых процедур. Приведём тексты запросов для вызова хранимых процедур.

qryAddInvoiceLine:

```
EXECUTE PROCEDURE sp_add_invoice_line(
  :invoice_id,
  :product_id,
  :quantity
)
```

qryEditInvoiceLine:

```
EXECUTE PROCEDURE sp_edit_invoice_line(
  :invoice_line_id,
  :quantity
)
```

qryDeleteInvoiceLine:

```
EXECUTE PROCEDURE sp_delete_invoice_line(
  :invoice_line_id
)
```

Форма для добавления новой записи и редактирования существующей, как и в случае с шапкой не будет использовать DBAware. Для выбора товара мы будем использовать компонент TButtonedEdit. Код обработчика нажатия кнопки в компоненте TButtonedEdit будет выглядеть следующим образом:

```
procedure TEditInvoiceLineForm.edtProductRightButtonClick(Sender: TObject);
var
  xSelectForm: TGoodsForm;
begin
  // не позволяем изменять товар в режиме редактирования
  // это можно сделать только при добавлении новой позиции
  if FEditMode = emInvoiceLineEdit then
    Exit;

  xSelectForm := TGoodsForm.Create(Self);
  try
    xSelectForm.Visible := False;
    if xSelectForm.ShowModal = mrOK then
    begin
      FProductId := xSelectForm.qryGoods.FieldByName('PRODUCT_ID')
        .AsInteger;
      edtProduct.Text := xSelectForm.qryGoods.FieldByName('NAME').AsString;
      // в данном случае мы копируем также цену по прайсу
      edtPrice.Text := xSelectForm.qryGoods.FieldByName('PRICE').AsString;
    end;
  finally
    xSelectForm.Free;
```

```
  end;
end;
```

Поскольку мы используем не DBAware компоненты, то при вызове формы редактирования нам будет необходимо инициализировать код товара, его наименование и стоимость для отображения.

```
procedure TInvoiceForm.actEditInvoiceLineExecute(Sender: TObject);
var
  xEditor: TEditInvoiceLineForm;
begin
  xEditor := TEditInvoiceLineForm.Create(Self);
  try
    xEditor.InvoiceForm := Self;
    xEditor.EditMode := emInvoiceLineEdit;
    xEditor.Caption := 'Редактирование позиции';

    xEditor.InvoiceLineId :=
qryInvoiceLine.FieldByName('INVOICE_LINE_ID').AsInteger;
    xEditor.SetProduct(qryInvoiceLine.FieldByName('PRODUCT_ID').AsInteger,
                       qryInvoiceLine.FieldByName('PRODUCTNAME').AsString,
                       qryInvoiceLine.FieldByName('SALE_PRICE').AsCurrency);
    xEditor.Quantity := qryInvoiceLine.FieldByName('QUANTITY').AsInteger;

    xEditor.ShowModal;
  finally
    xEditor.Free;
  end;
end;

procedure TEditInvoiceLineForm.SetProduct(AProductId: Integer;
  AProductName: string; APrice: Currency);
begin
  FProductId := AProductId;
  edtProduct.Text := AProductName;
  edtPrice.Text := CurrToStr(APrice);
end;
```

Обработку добавления новой позиции и редактирование существующей будем производить в событии закрытия модальной формы.

```
procedure TEditInvoiceLineForm.FormClose(Sender: TObject;
  var Action: TCloseAction);
var
  xCustomerId: Integer;
begin
  // если форма закрыта не по нажатию кнопки OK,
  // то вообще ничего не делаем. Транзакция не стартует.
  if ModalResult <> mrOK then
  begin
    Action := caFree;
    Exit;
  end;

  // Всё делаем в короткой транзакции
  FInvoiceForm.trWrite.StartTransaction;
  try
    if FEditMode = emInvoiceLineAdd then
    begin
      FInvoiceForm.qryAddInvoiceLine.ParamByName('INVOICE_ID').AsInteger :=
        FInvoiceId;

      if FProductId = 0 then
        raise Exception.Create('Не выбран товар');
```

```
      FInvoiceForm.qryAddInvoiceLine.ParamByName('PRODUCT_ID').AsInteger :=
        FProductId;
      FInvoiceForm.qryAddInvoiceLine.ParamByName('QUANTITY').AsInteger :=
        FQuantity;
      // Выполняем хранимую процедуру для добавления позиции
      FInvoiceForm.qryAddInvoiceLine.Execute();
    end;

    if FEditMode = emInvoiceLineEdit then
    begin
      FInvoiceForm.qryEditInvoiceLine.ParamByName('INVOICE_LINE_ID').AsInteger :=
        FInvoiceLineId;
      FInvoiceForm.qryEditInvoiceLine.ParamByName('QUANTITY').AsInteger :=
        FQuantity;
      // Выполняем хранимую процедуру для редактирования
      FInvoiceForm.qryEditInvoiceLine.Execute();
    end;
    // Подтверждаем транзакцию
    FInvoiceForm.trWrite.Commit;
    // Обновляем оба грида
    FInvoiceForm.qryInvoice.Refresh;
    FInvoiceForm.qryInvoiceLine.Refresh;

    Action := caFree;
  except
    on E: Exception do
    begin
      if FInvoiceForm.trWrite.Active then
        FInvoiceForm.trWrite.Rollback;
      Application.ShowException(E);
      // Не загрываем окно редактирования. Позволяем пользователю исправить ошибку
      Action := caNone;
    end;
  end;
end;
```

Ну, вот и всё. Надеюсь, эта статья помогла вам разобраться в особенностях написания приложения на Delphi с использованием компонентов FireDac при работе с СУБД Firebird.

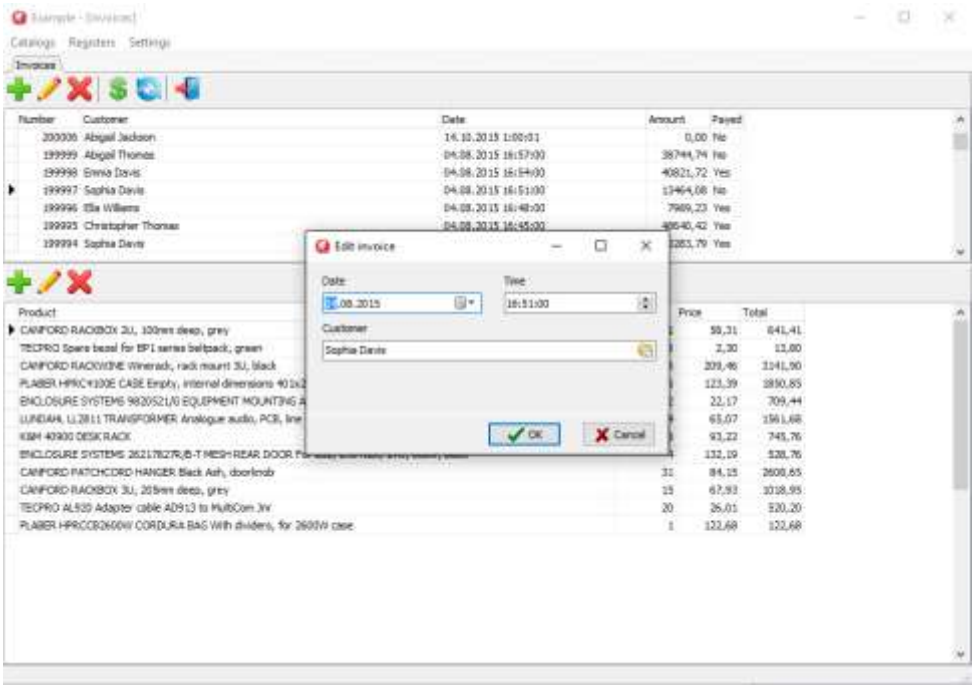

Рисунок 8. Скриншот работающего приложения

Ссылки

Исходные коды примера приложения
Готовая БД

www.ibase.ru, www.ibsurgeon.com

support@ibase.ru, support@ib-aid.com