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

ADO.NET Entity Framework 6

Автор: Денис Симонов
31.01.2016

Оглавление

Создание приложений с использованием Entity Framework

В данной статье будет описан процесс создания приложений для СУБД Firebird с использованием компонентов доступа Entity Framework и среды Visual Studio 2015.

ADO.NET Entity Framework (EF) — объектно-ориентированная технология доступа к данным, является object-relational mapping (ORM) решением для .NET Framework от Microsoft. Предоставляет возможность взаимодействия с объектами как посредством LINQ в виде LINQ to Entities, так и с использованием Entity SQL.

Entity Framework предполагает три возможных способа взаимодействия с базой данных:

- **Database first**: Entity Framework создаёт набор классов, которые отражают модель конкретной базы данных
- **Model first**: сначала разработчик создаёт модель базы данных, по которой затем Entity Framework создаёт реальную базу данных на сервере.
- **Code first**: разработчик создаёт класс модели данных, которые будут храниться в БД, а затем Entity Framework по этой модели генерирует базу данных и её таблицы

В своём приложении мы будем использовать подход Code First, однако вы без труда сможете использовать и другие подходы.

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

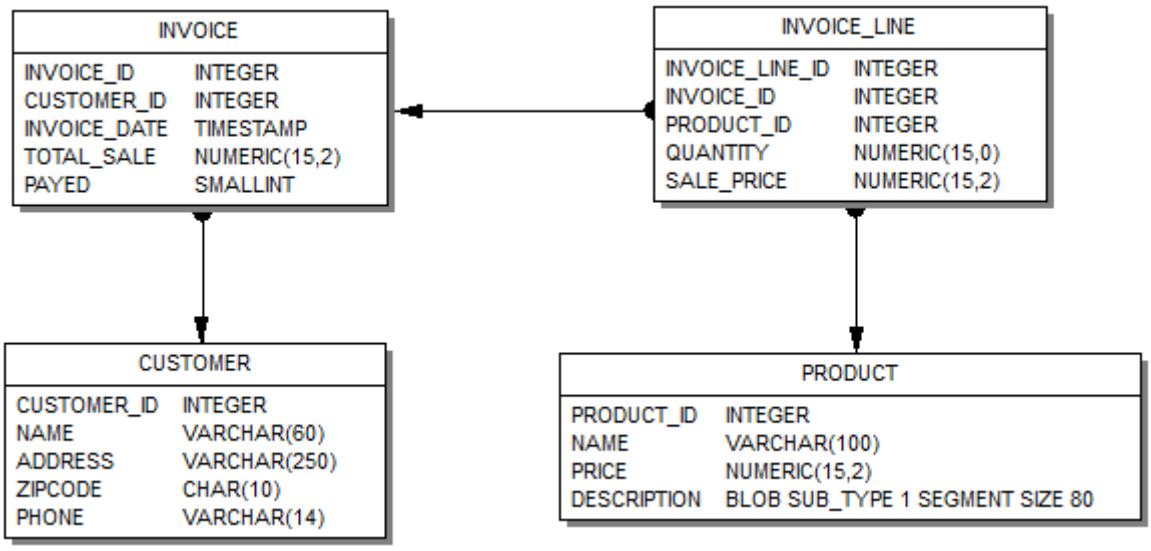


Рисунок 1. Модель базы данных.

В конце данной статьи приведена 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.

Подготовка Visual Studio 2015 для работы с Firebird

Для работы с Firebird вам необходимо установить:

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

Установка первых двух не вызывает никаких сложностей. В настоящий момент они распространяются и устанавливаются в проект с помощью NuGet. А вот последняя библиотека, предназначенная для работы мастеров Visual Studio, устанавливается не так легко и может потратить у вас много сил и времени.

Добрые люди попытались автоматизировать процесс установки и включить установку всех компонентов в один дистрибутив. Однако в ряде случаев вам может потребоваться ручная установка всех компонентов. В этом случае вам потребуется скачать:

- 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

Далее описан процесс установки:

1. Устанавливаем FirebirdSql.Data.FirebirdClient-4.10.0.0.msi
2. Распаковываем EntityFramework.Firebird-4.10.0.0-NET45.7z в папку с установленным клиентом Firebird. У меня это папка c:\Program Files (x86)\FirebirdClient\

   Важно!
   Это необходимо делать с правами администратора. Как и другие действия с защищёнными директориями.

3. Необходимо установить сборки Firebird в GAC. Для удобство пописываем в %PATH% путь до утилиты gacutil для .NET Framework 4.5. У меня этот путь c:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6.1 Tools\

4. Запускаем командную строку cmd от имени администратора и переходим в директорию с установленным клиентом.

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

5. Теперь проверяем что FirebirdSql.Data.FirebirdClient установлен в GAC. Для этого набираем команду

```
gacutil /l FirebirdSql.Data.FirebirdClient
Microsoft (R) .NET Global Assembly Cache Utility.  Version
4.0.30319.0
с Корпорация Майкрософт (Microsoft Corporation). Все права
защищены.

В глобальном кэше сборок содержатся следующие сборки:
  FirebirdSql.Data.FirebirdClient, Version=4.10.0.0,
Culture=neutral, PublicKeyToken=3750abcc3150b00c,
processorArchitecture=MSIL

Число элементов = 1
```

   Если FirebirdSql.Data.FirebirdClient не был установлен в GAC, то сделаем это с помощью команды

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

6. Теперь установим EntityFramework.Firebird в GAC

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

7. Распаковываем DDEXProvider-3.0.2.0.7z в удобную директорию. Я распаковал её в c:\Program Files (x86)\FirebirdDDEX\
8. Туда же распаковываем DDEXProvider-3.0.2.0-src.7z содержимое поддиректории архива /reg_files/VS2015

> **Примечание автора**
> Забавно, но по какой-то причине этих файлов нет в предыдущем архиве со скомпилированными dll библиотеками, но они присутствуют в архиве с исходными кодами.

9. Открываем файл FirebirdDDEXProvider64.reg с помощью блокнота. Находим строчку, которая содержит %path% и меняем его на полный путь к файлу FirebirdSql.VisualStudio.DataTools.dll

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

10. Сохраняем этот файл, запускаем его. На запрос добавить информацию в реестр нажимаем ДА.
11. Теперь нужно отредактировать файл machine.config, в моем случае он находится по пути: C:\Windows\Microsoft.NET\Framework\v4.0.30319\Config Открываем этот файл блокнотом. Находим секцию

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

Добавляем в эту секцию строчку:

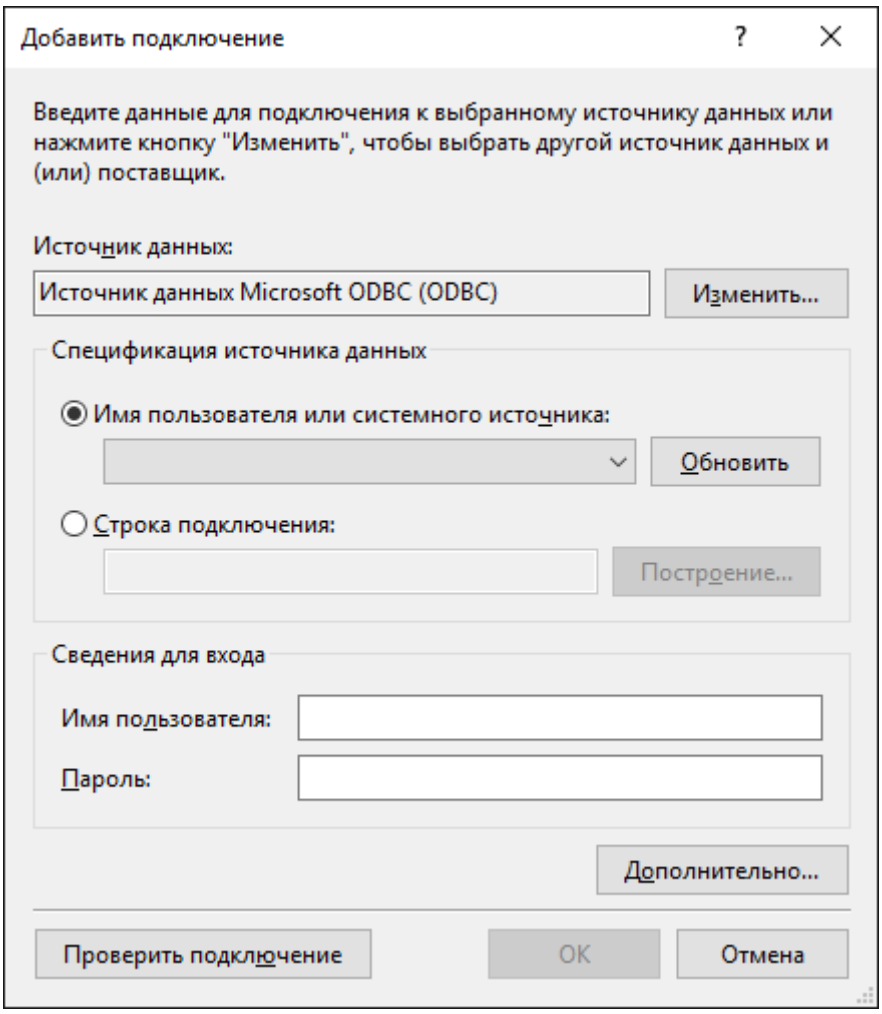
```
    <add name="FirebirdClient Data Provider" invariant="FirebirdSql.Data.FirebirdClient" description=".Net Framework Data Provider for Firebird" type="FirebirdSql.Data.FirebirdClient.FirebirdClientFactory, FirebirdSql.Data.FirebirdClient, Version=4.10.0.0, Culture=neutral, PublicKeyToken=3750abcc3150b00c"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

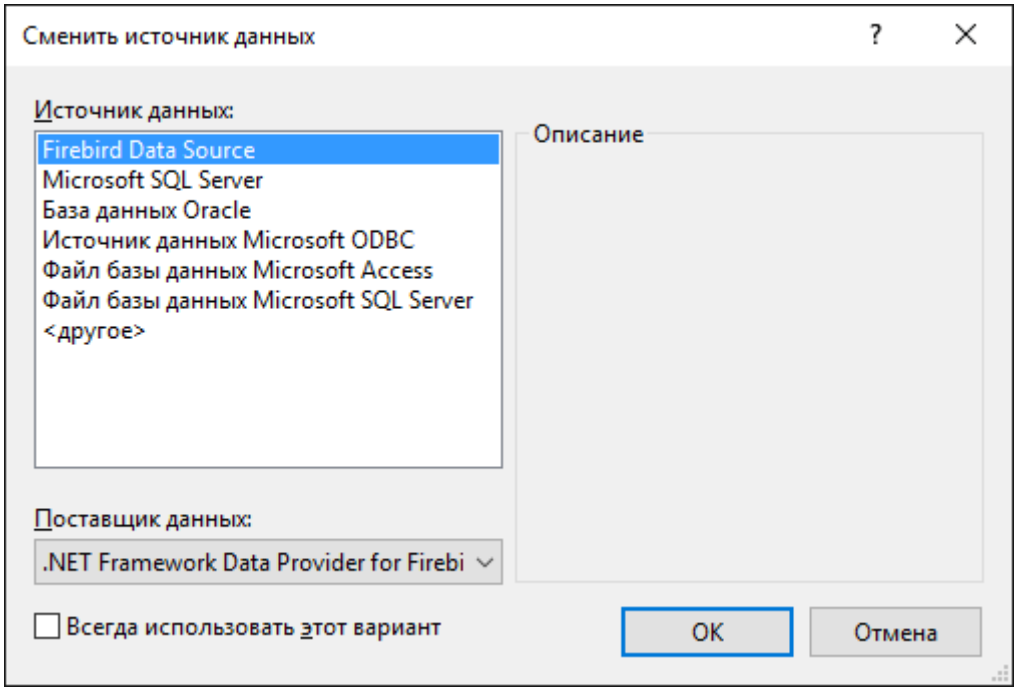
Замечание
Всё это действительно для версии 4.10.0.

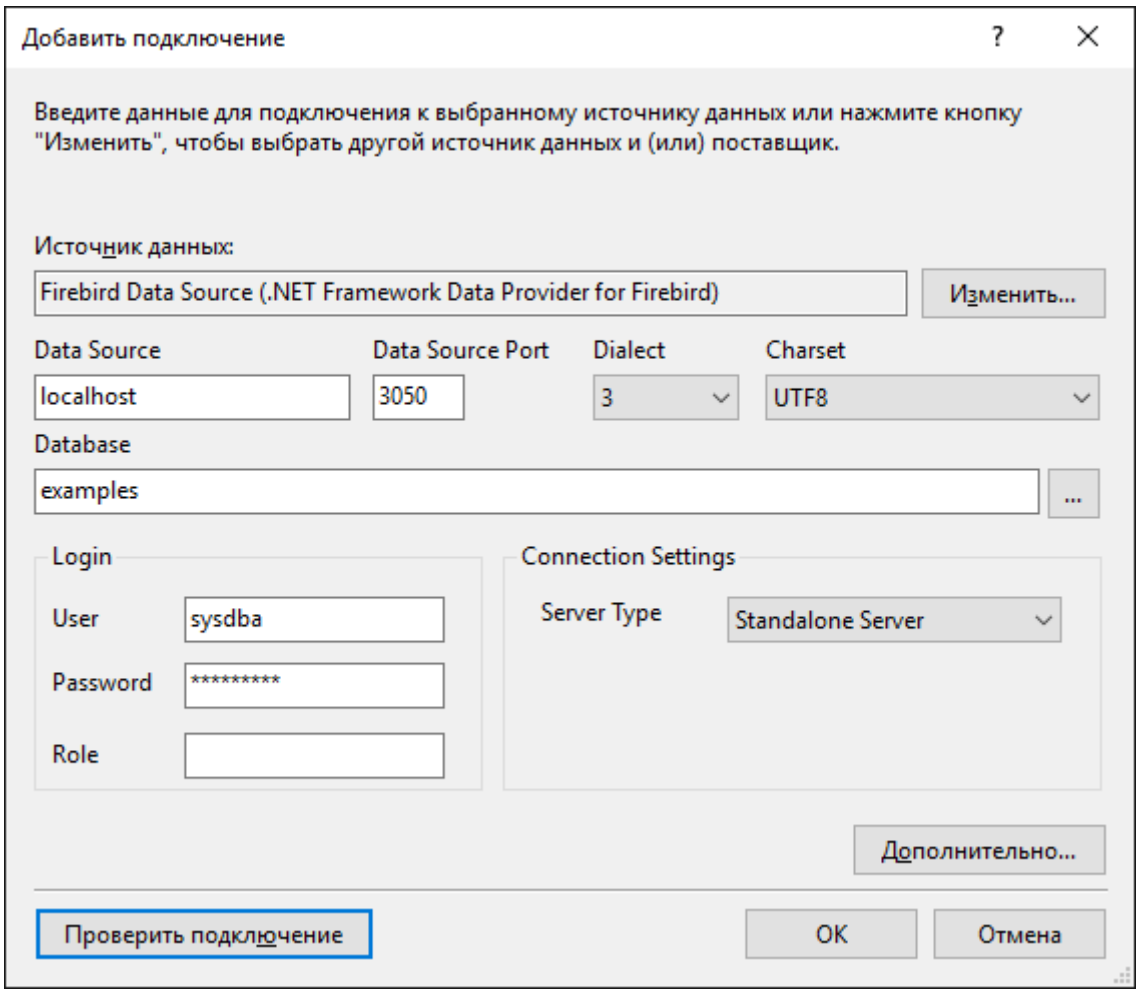
То же самое проделаем для machine.config. который находится в c:\Windows\Microsoft.NET\Framework64\v4.0.30319\Config\

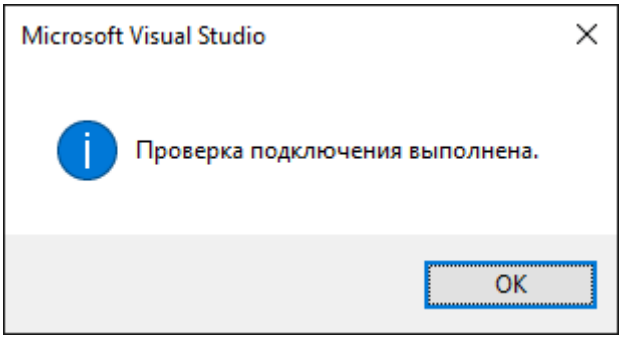
Установка закончена.

Для проверки, что всё успешно установилось, запускаем Visual Studio 2015. Находим обозреватель серверов и пытаемся подключиться к одной из существующих баз данных Firebird.


Добавить подключение
Введите данные для подключения к выбранному источнику данных или нажмите кнопку "Изменить", чтобы выбрать другой источник данных и (или) поставщик.
Источник данных:
Источник данных Microsoft ODBC (ODBC)
Изменить...
Спецификация источника данных
Имя пользователя или системного источника:
Обновить
Строка подключения:
Построение...
Сведения для входа
Имя пользователя:
Пароль:
Дополнительно...
Проверить подключение
OK
Отмена


Сменить источник данных
Источник данных:
Firebird Data Source
Microsoft SQL Server
База данных Oracle
Источник данных Microsoft ODBC
Файл базы данных Microsoft Access
Файл базы данных Microsoft SQL Server
<другое>
Описание
Поставщик данных:
.NET Framework Data Provider for Firebi
Всегда использовать этот вариант
OK
Отмена





Создание проекта

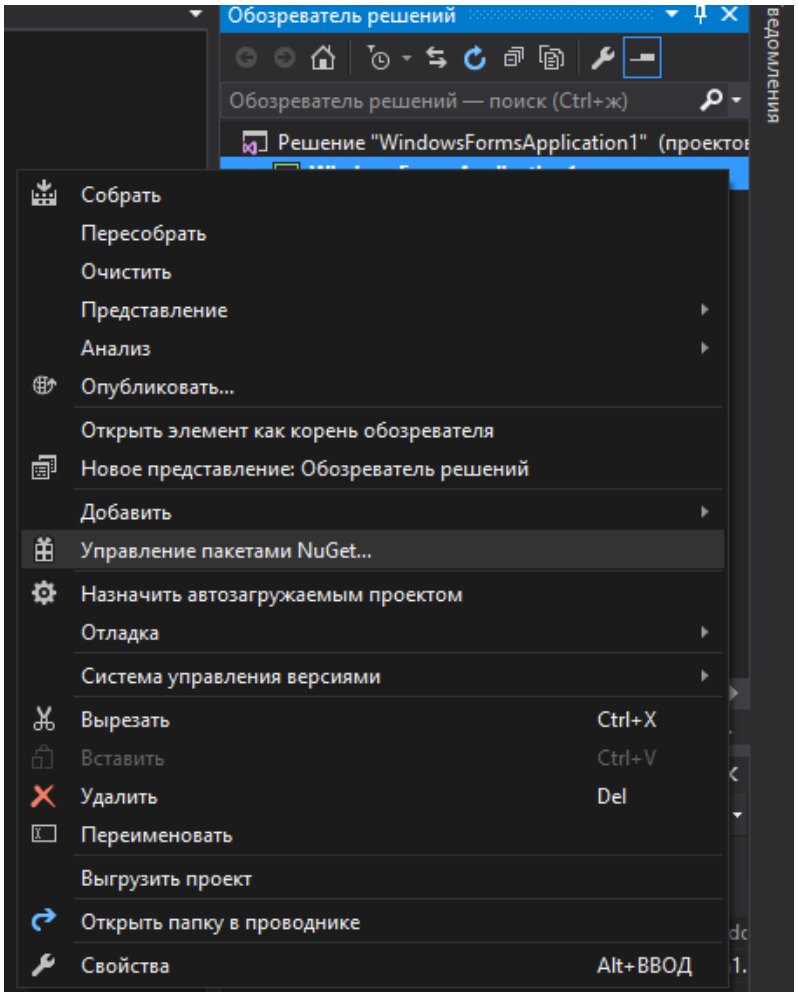
В данной статье мы рассмотрим пример создания Windows Forms приложения. Остальные типы приложений хоть и отличаются, но принципы работы с Firebird через Entity Framework остаются те же.

Прежде всего, после создания Windows Forms проекта нам необходимо добавить с помощью менеджера пакетов NuGet следующие пакеты:

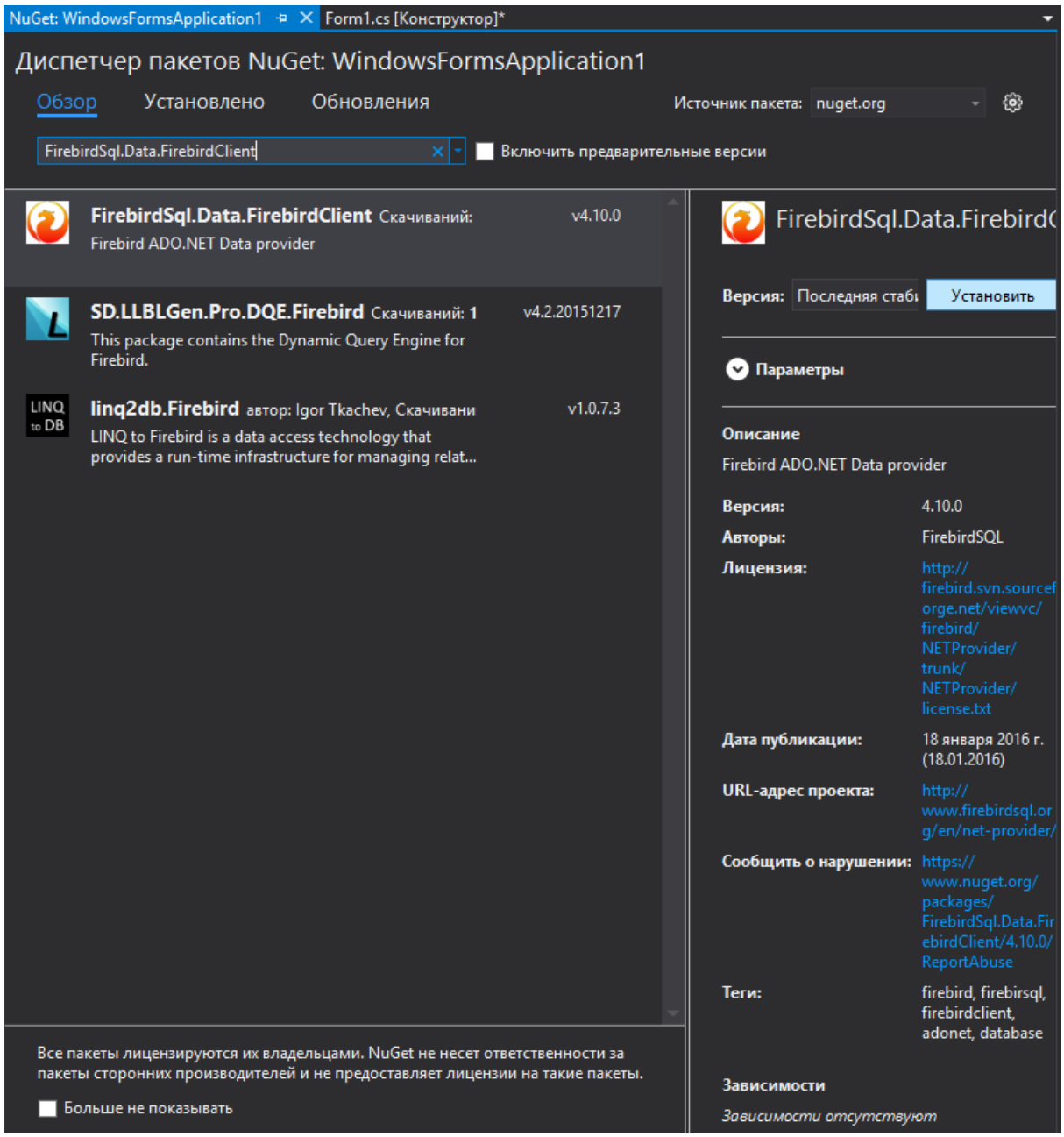
- FirebirdSql.Data.FirebirdClient
- EntityFramework

- EntityFramework.Firebird

Для этого необходимо щёлкнуть правой клавишей мыши по имени проекта в обозревателе решений и в выпадающем меню выбрать пункт «Управление пакетами NuGet».



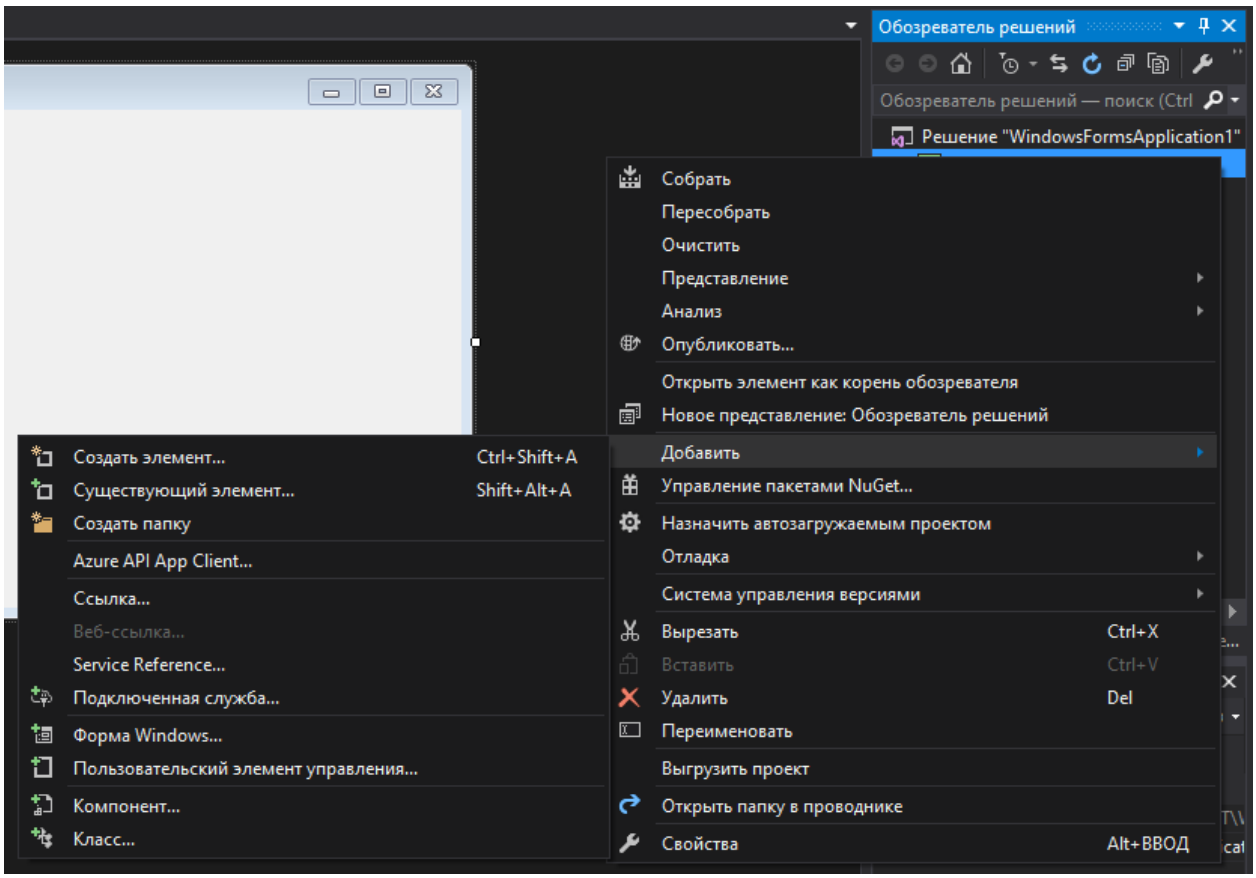
В появившемся менеджере пакетов произвести поиск и установку необходимых пакетов.



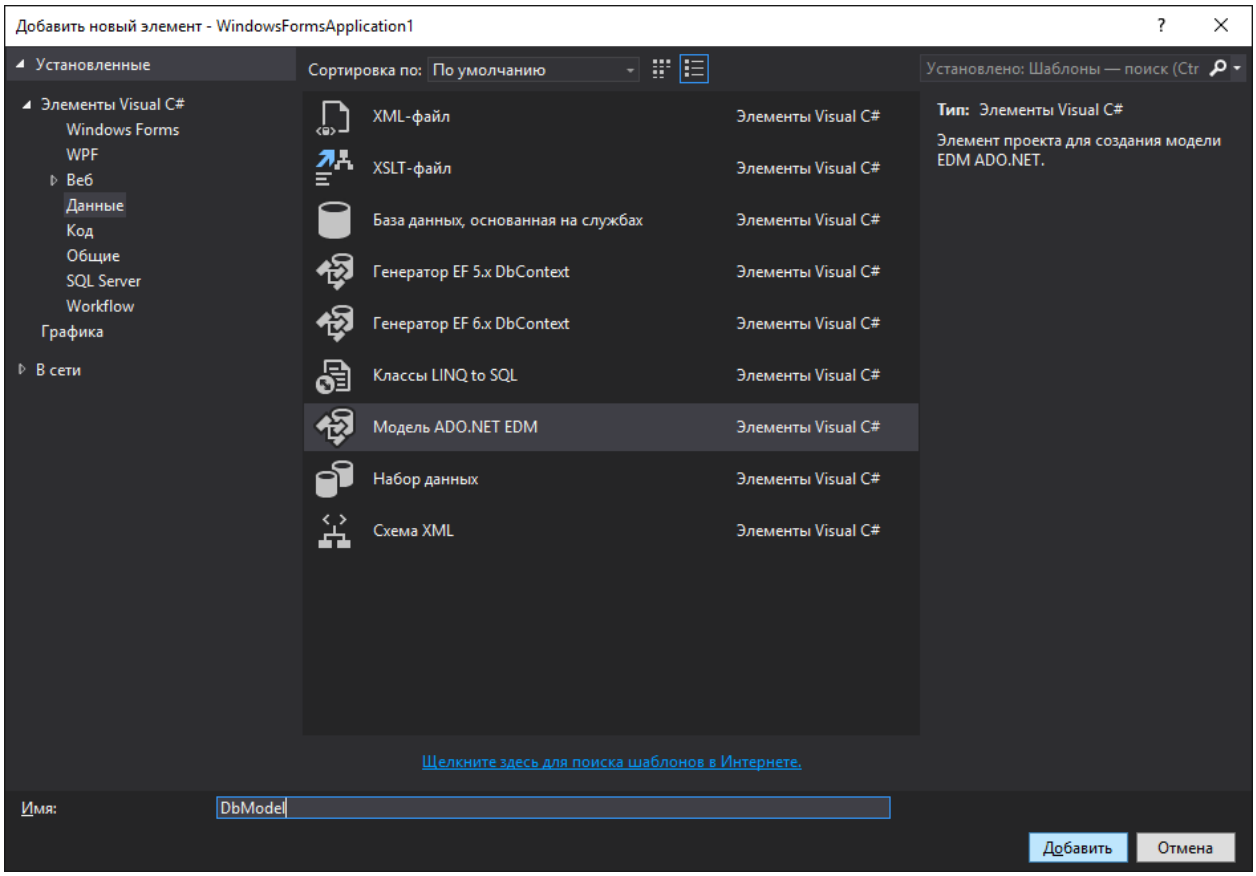
Создание EDM модели

В своём приложении мы будем использовать подход Code First.

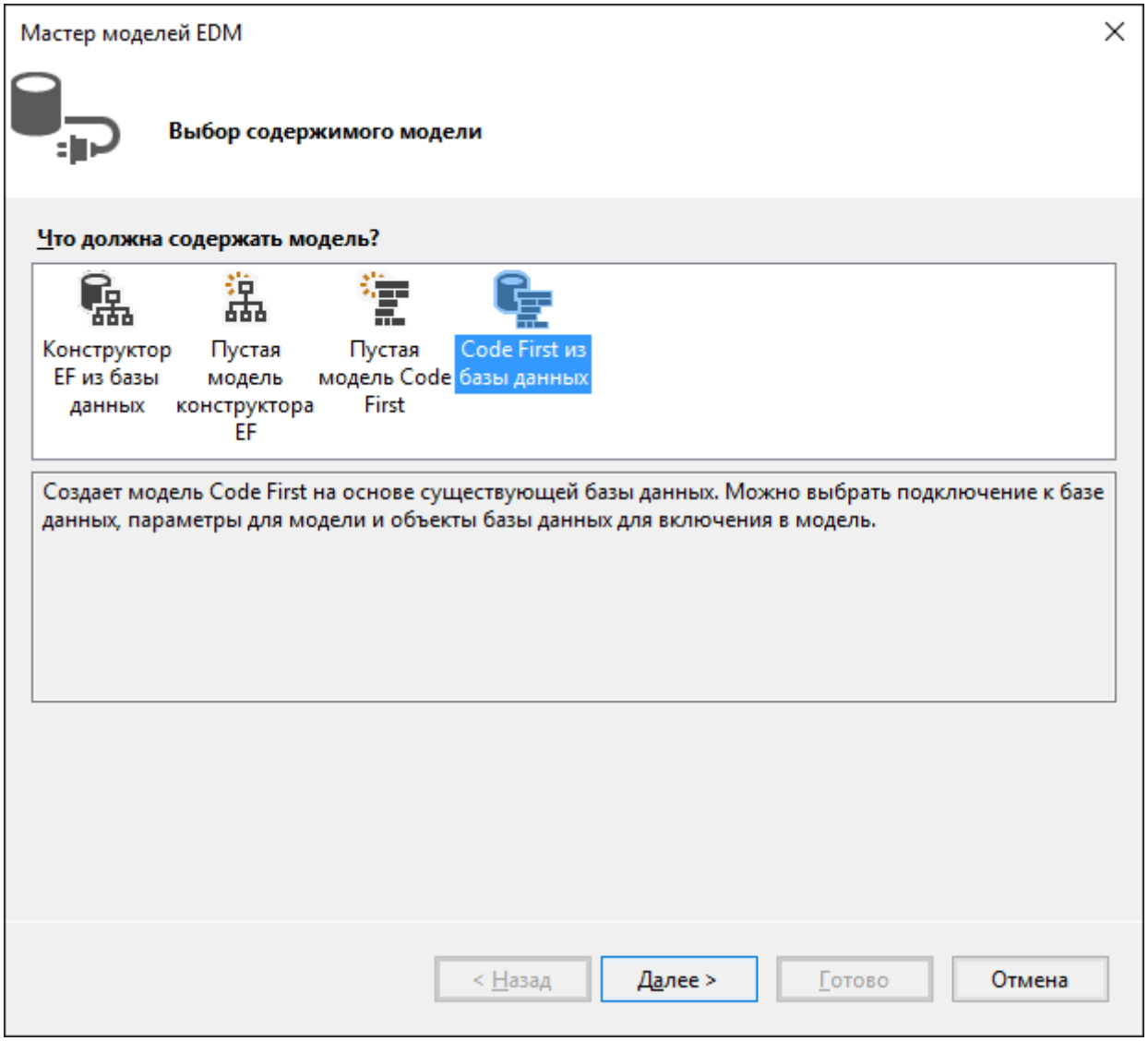
Для создания модели EDM необходимо щёлкнуть правой клавишей мыши по имени проекта в обозревателе решений и выбрать пункт меню Добавить -> Создать элемент.



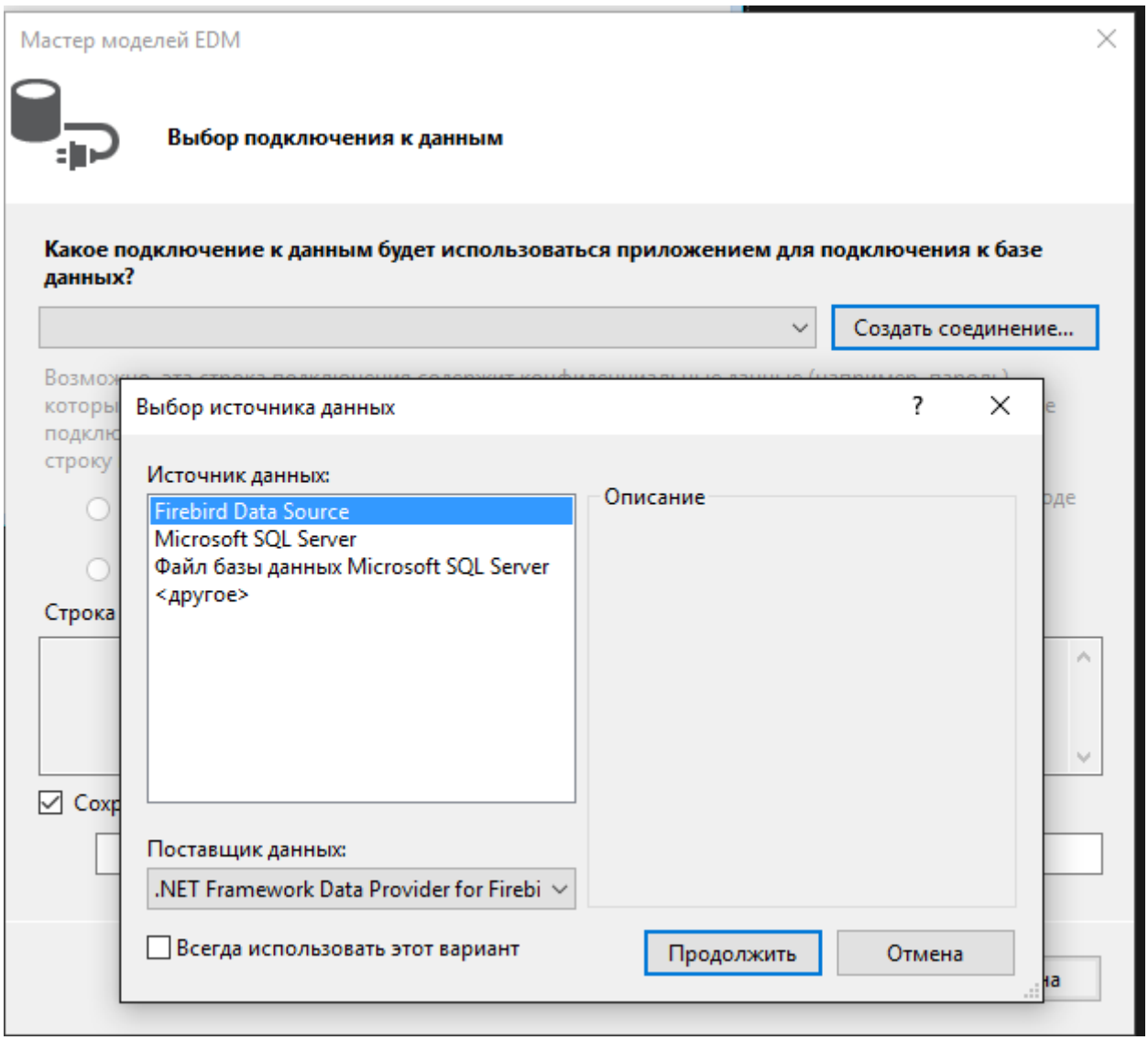
Далее в мастере добавления нового элемента выбираем пункт «Модель ADO.NET EDM».

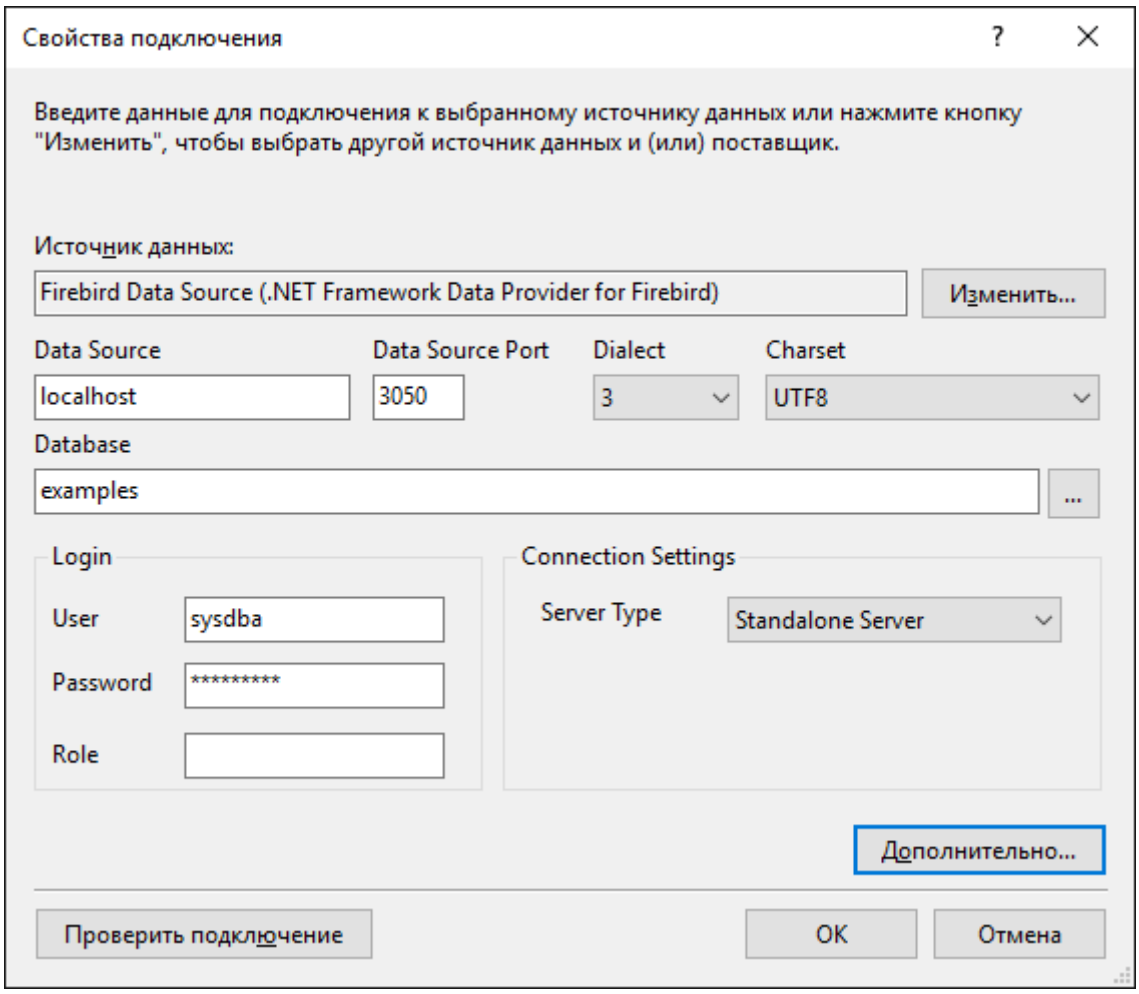


Поскольку у нас уже существует база данных (см. https://habrahabr.ru/post/273549/), то будем генерировать EDM модель из базы данных.

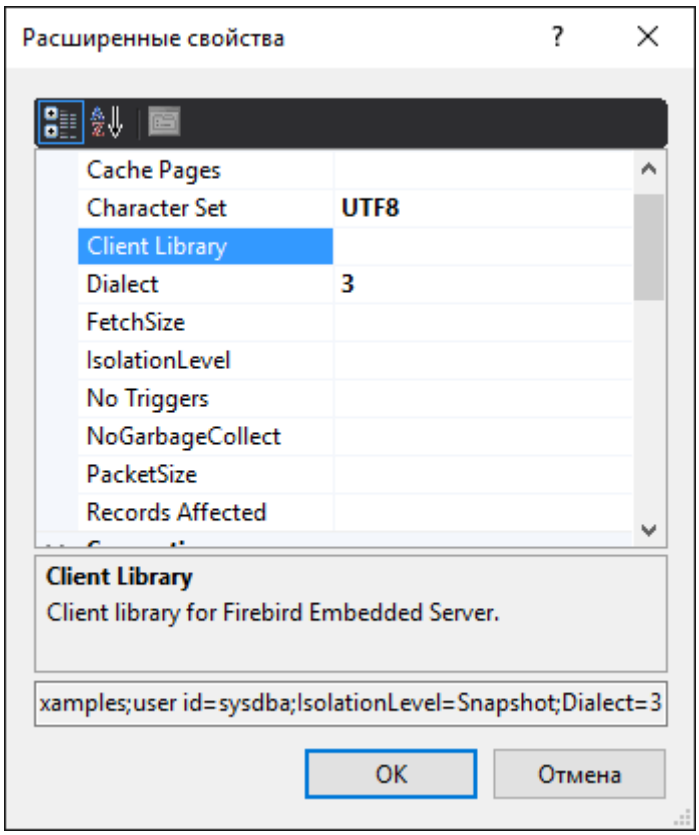


Теперь надо выбрать подключение, из которого будет создана модель. Если Такого подключения нет, то его надо создать.

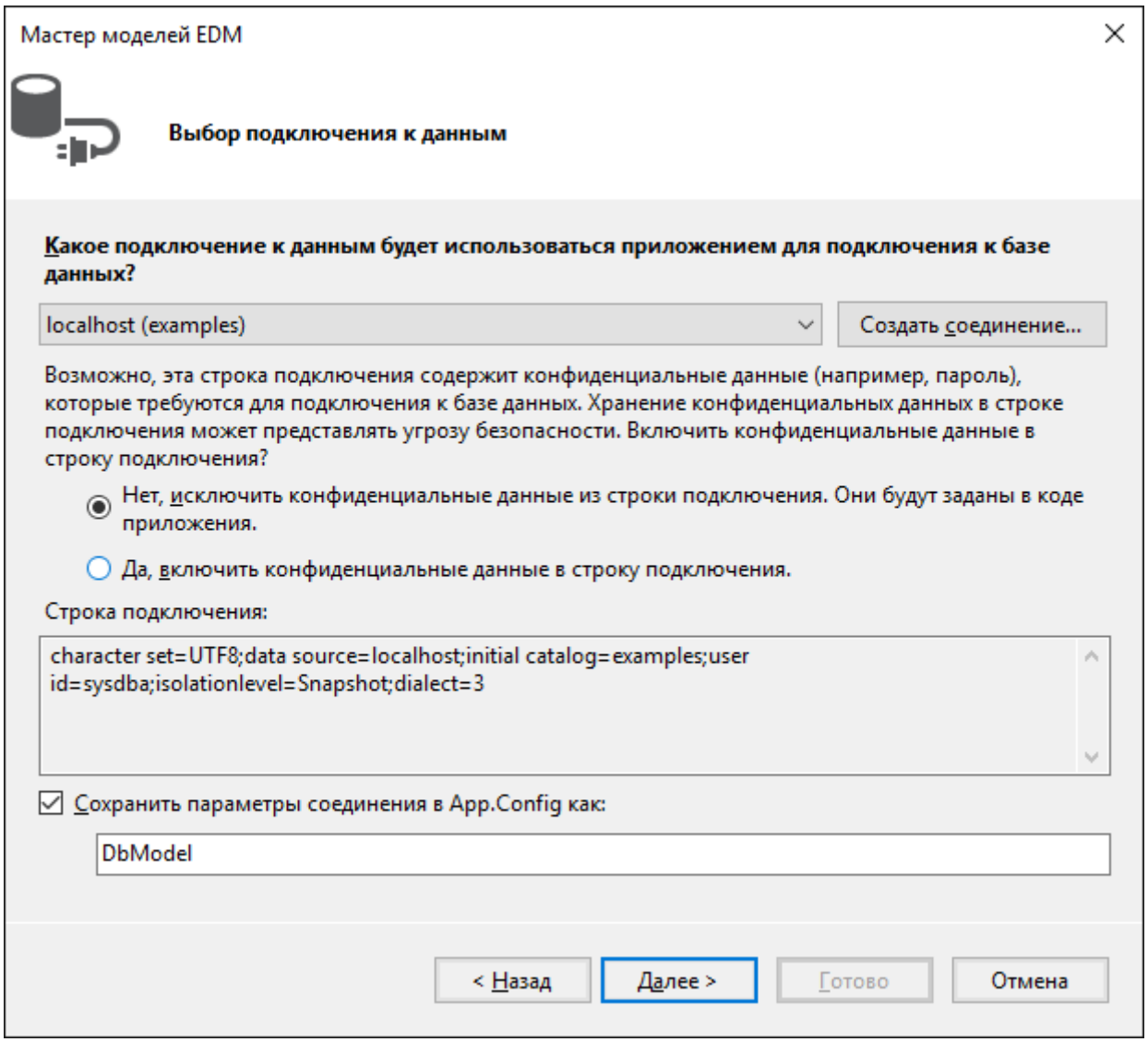

Мастер моделей EDM
Выбор подключения к данным
Какое подключение к данным будет использоваться приложением для подключения к базе данных?
Создать соединение...
Выбор источника данных
Источник данных:
Firebird Data Source
Microsoft SQL Server
Файл базы данных Microsoft SQL Server
<другое>
Описание
Поставщик данных:
.NET Framework Data Provider for Firebi
Всегда использовать этот вариант
Продолжить
Отмена



Кроме основных параметров подключения могут потребоваться также указать ряд дополнительных параметров, например, уровень изолированности транзакций (по умолчанию Read Commited), использование пула подключений и т.д. Поскольку Entity Framework (как впрочем, ADO.NET в целом) использует отсоединённую модель взаимодействия, при которой каждое подключение и транзакция активна очень короткий промежуток времени, то я бы рекомендовал задать режим изолированности Snapshot.



В процессе работы мастера создания модели у вас спросят, как хранить строку подключения.



Если вы строите веб приложение или трёхзвенку, где все пользователи будут работать с базой данных под одной и той же учётной записью, то смело выбирайте «Да». Если же ваше приложение должно запрашивать учётные данные для соединения с базой данных выбирайте «Нет». Впрочем, с мастерами гораздо более удобно работать, когда у вас выбран пункт «Да». Вы всегда можете это изменить в готовом приложении, просто отредактировав строку подключения в файле конфигурации приложения <AppName>.exe.conf. Строка подключения будет сохранена в секции **connectionStrings** примерно в таком виде

```
<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" />
```

Для того чтобы файл конфигурации перестал хранить конфиденциальную информацию просто удалите из строки подключения `password=masterkey;`

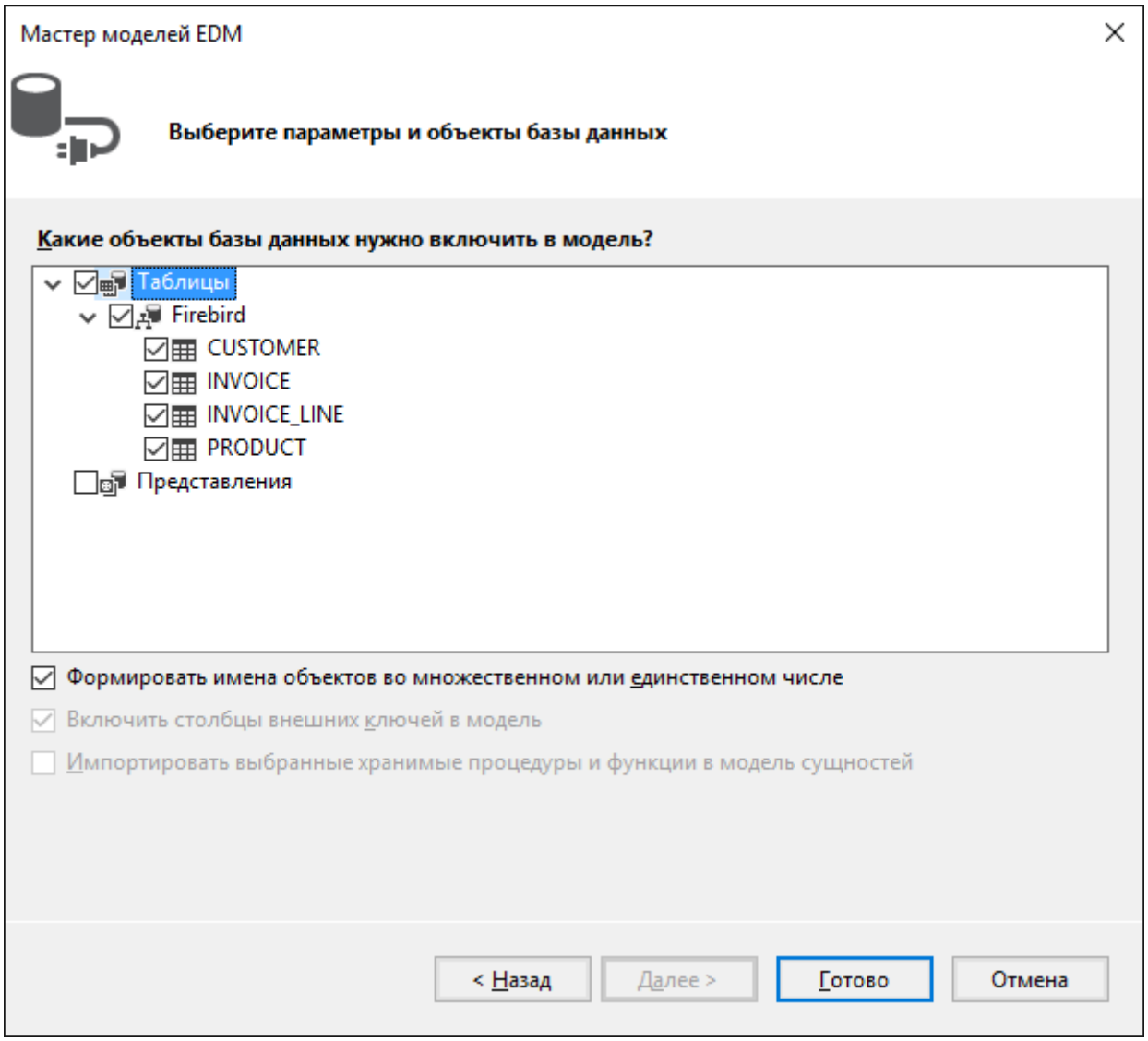
Замечание о работе с Firebird 3.0

К сожалению текущий ADO .Net провайдер для Firebird (версия 4.10.0) не поддерживает аутентификацию по протоколу SRP (по умолчанию в Firebird 3.0). Поэтому если вы желаете работать с Firebird 3.0, то вам необходимо изменить некоторые настройки в firebird.conf (или в databases.conf для конкретной БД), чтобы Firebird работал через Legacy_Auth. Для этого необходимо поменять следующие настройки:

UserManager = Legacy_UserManager
WireCrypt = Disabled
AuthServer = Legacy_Auth, Srp, WinSspi

Сохранить настройки. После чего необходимо создать пользователя SYSDBA и других пользователей с использованием Legacy_UserManager.

Далее у вас спросят, какие таблицы и представления должны быть включены модель.



В принципе EDM модель готова. После работы этого мастера у вас должно появиться 5 новых файлов. Один файл модели и четыре файла описывающих каждую из сущностей модели.

Давайте посмотрим один из сгенерированных файлов описывающих сущность INVOICE.

```
    [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 double? 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; }
    }
```

Класс содержат свойства, которые отображают поля таблицы INVOICE. Каждое из таких свойств снабжено атрибутами, описывающими ограничения. Подробнее об различных атрибутах вы можете почитать в документации Майкрософт Code First Data Annotations.

Кроме того, было сгенерировано ещё два навигационных свойства CUSTOMER и INVOICE_LINES. Первое содержит ссылку на сущность поставщика, второе – коллекцию строк накладных. Оно было сгенерировано потому, что таблица INVOICE_LINE имеет внешний ключ на таблицу INVOICE. Конечно, вы можете удалить это свойство из сущности INVOICE, но делать это вовсе не обязательно. Дело в том, что в данном случае свойства CUSTOMER и INVOICE_LINES использует так называемую «ленивую загрузку». При таком загрузка осуществляется при первом обращении к объекту, т.е. если связанные данные не нужны, то они не подгружаются. Однако при первом же обращении к навигационному свойству эти данные автоматически подгружаются из бд.

При использовании ленивой загрузки надо иметь в виду некоторые моменты при объявлении классов. Так, классы, использующие ленивую загрузку должны быть публичными, а их свойства должны иметь модификаторы **public** и **virtual**.

В этом же классе нас ожидает первый неприятный сюрприз. Поле TOTAL_SALE было отображено в сущности как double, хотя в базе данных оно имеет тип NUMERIC(15, 2), таким образом, мы имеем потерю точности. Я склонен расценивать это как баг в Firebird ADO.NET Provider. Давайте попробуем

исправить эту досадную оплошность. В C# существует тип decimal для операций над числами с фиксированной точностью.

```
public decimal TOTAL_SALE { get; set; }
```

Кроме того, изменим описание всех полей во всех сущностях, где используется тип Firebird NUMERIC(x, y). А именно PRODUCT.PRICE, INVOICE_LINE.QUANTITY, INVOICE_LINE.SALE_PRICE.

Теперь откроем файл DbModel.cs описывающий модель в целом.

```
    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);

        }
    }
```

Здесь мы видим свойства описывающие набор данных для каждой сущности. А так же задание дополнительных свойств создания модели с помощью Fluent API. Полное описание Fluent API вы может прочитать в документации Microsoft Configuring/Mapping Properties and Types with the Fluent API.

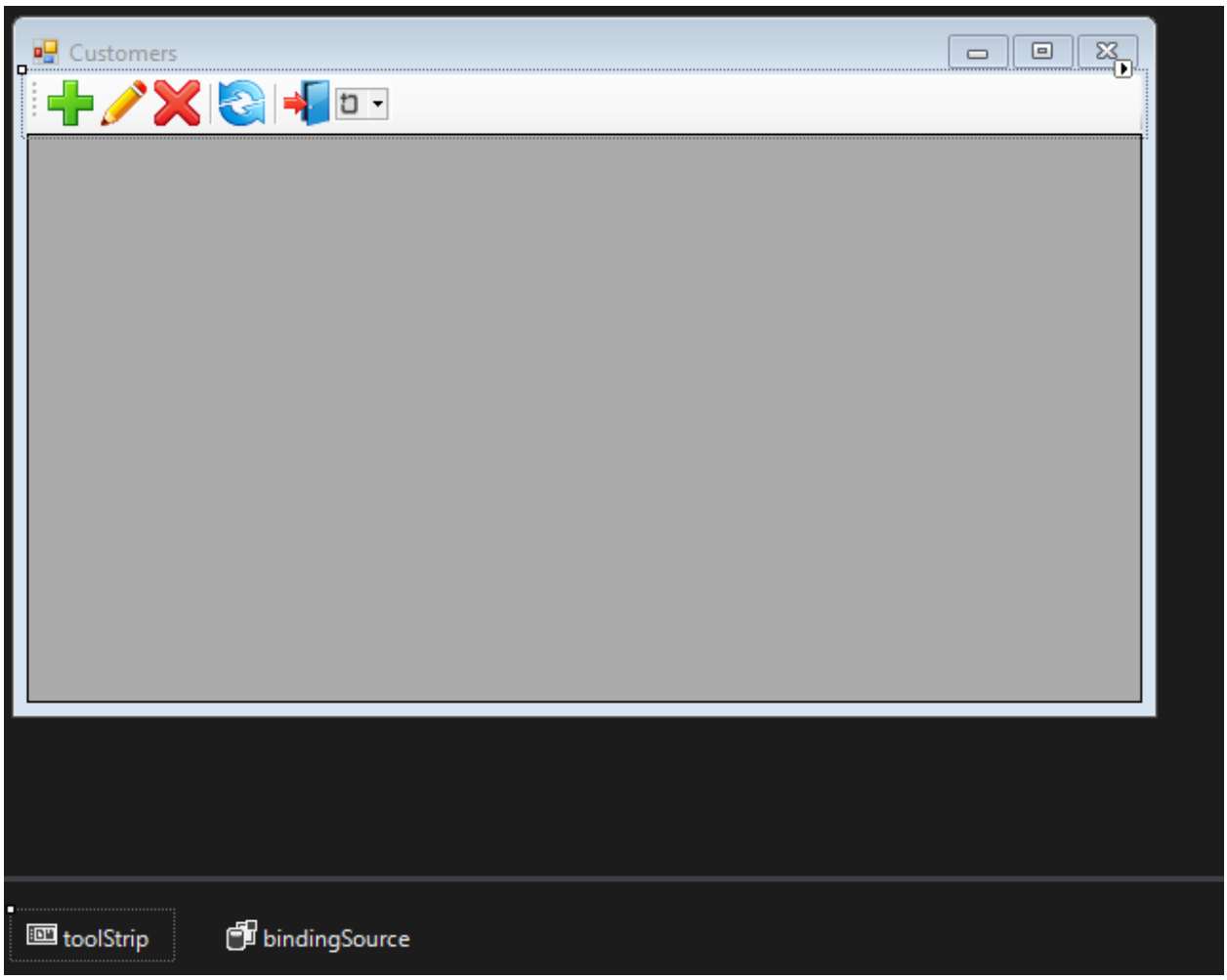
Зададим в методе OnModelCreating точность для свойств типа decimal с помощью Fluent API. Для этого допишем следующие строчки

```
            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);
```

Создание пользовательского интерфейса

В нашем приложении мы создадим два справочника: справочник товаров и справочник заказчиков. Каждый справочник содержит сетку DataGridView, панель с кнопками ToolStrip, а также компонент BindingSource, который служит для упрощения привязки данных к элементам управления в форме.



Поскольку по функционалу оба справочника похожи и реализованы схожим образом описывать мы будем только один.

Получение контекста

Для работы с нашей моделью нам потребуется метод для получения контекста (или модели). В принципе для этого достаточно выполнить:

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

Однако, если в строке подключения не хранятся конфиденциальные данные (например, пароль), а мы инициализируем во время авторизации их при старте приложения, то нам потребуется специальный метод для хранения и восстановления строки подключения или сохранение ранее созданного контекста. Для этого создадим специальный класс, который помимо метода для получения контекста будет также содержать некоторые глобальные переменные уровня приложения, например рабочий период.

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

         /// <summary>
         /// Дата начала рабочего периода
         /// </summary>
         public static DateTime StartDate { get; set; }

         /// <summary>
         /// Дата окончания рабочего периода
         /// </summary>
         public static DateTime FinishDate { get; set; }

        /// <summary>
        /// Возвращает экземпляр модели (контекста)
        /// </summary>
        /// <returns>Модель</returns>
        public static DbModel CreateDbContext() {
            dbContext = dbContext ?? new DbModel();
            return dbContext;
        }
    }
```

Сама строка подключения инициализируется при старте приложения, после того как успешно прошла авторизация. Для этого в обработчике события Load главной формы напишем следующий код.

```
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;

            // пробуем подключится
            dbContext.Database.Connection.Open();
        }
        catch (Exception ex)
```

```
        {
            // отображаем ошибку
            MessageBox.Show(ex.Message, "Error");
            Application.Exit();
        }
    }
    else
        Application.Exit();

}
```

Теперь для получения контекста мы будем использовать статический метод CreateDbContext.

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

Работа с данными

Сами по себе сущности модели не содержат никаких данных. Самым простым способом загрузить данные является вызовам метода Load, например вот так:

```
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;
}
```

Однако такой способ имеет ряд недостатков:

1. Метод Load загружает сразу все данные из таблицы CUSTOMER в память.
2. Ленивые свойства (`INVOICES`) хоть и не загружаются сразу, а лишь по мере обращения к ним, всё равно будут загружены при отображении записей в гриде. Причём ровно столько раз, сколько записей будет выведено.
3. Порядок записей неопределён.

Для обхода этих недостатком мы будем использовать технологию LINQ (Language Integrated Query), или точнее **LINQ to Entities**. LINQ to Entities предлагает простой и интуитивно понятный подход для получения данных с помощью выражений, которые по форме близки выражениям языка SQL. С синтаксисом LINQ вы можете ознакомиться по LINQ to Entities.

Методы расширений LINQ могут возвращать два объекта: **IEnumerable** и **IQueryable**. Интерфейс IQueryable наследуется от IEnumerable, поэтому по идее

объект IQueryable это и есть также объект IEnumerable. Но между ними есть существенная разница.

Интерфейс IEnumerable находится в пространстве имён **System.Collections**. Объект IEnumerable представляет набор данных в памяти и может перемещаться по этим данным только вперёд. При выполнении запроса IEnumerable загружает все данные, и если нам надо выполнить их фильтрацию, то сама фильтрация происходит на стороне клиента.

Интерфейс IQueryable располагается в пространстве имён System.Linq. Объект IQueryable предоставляет удалённый доступ к базе данных и позволяет перемещаться по данным как в прямом порядке от начала до конца, так и в обратном порядке. В процессе создания запроса, возвращаемым объектом которого является IQueryable, происходит оптимизация запроса. В итоге в процессе его выполнения тратится меньше памяти, меньше пропускной способности сети.

Свойство Local возвращает интерфейс IEnumerable. Поэтому мы можем составлять LINQ запросы к нему.

```
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();
}
```

Однако как уже сказано этот запрос будет выполняться над данными в памяти. В принципе для маленьких таблиц, которым не требуется предварительная фильтрация это приемлемо.

Для того чтобы LINQ запрос был преобразован в SQL и выполнялся на стороне сервера нам необходимо использовать в LINQ запросе вместо обращения к свойству `dbContext.CUSTOMERS.Local` обращаться сразу к `dbContext.CUSTOMERS`. В этом случае нам не потребуется предварительный вызов `dbContext.CUSTOMERS.Load();` для загрузки коллекции в память.

Однако тут нас подстерегает одна маленькая засада. Объекты IQueryable не умеют возвращать BindingList. BindingList является базовым классом для создания двустороннего механизма привязки данных. Из интерфейса IQueryable мы можем получить обычный список посредством вызова ToList, но в этом случае мы лишаемся приятных бонусов, таких как сортировка в гриде и многих других. Кстати в .NET Framework 5 это уже исправили и создали специальное расширение. Сделаем своё расширение, которое будет делать тоже самое.

```
public static class DbExtensions
{
```

```
    // Внутренний класс для маппинга на него значения генератора
    private class IdResult
    {
        public int Id { get; set; }
    }

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

    // Получение следующего значения последовательности
    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;
    }

    // Отсоединение всех объектов коллекции DbSet от контекста
    // Полезно для обновлении кеша
    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;
        }
    }

    // Обновление всех изменённых объектов в коллекции
    public static void Refresh(this DbModel dbContext, RefreshMode mode,
       IEnumerable collection)
    {
        var objectContext = ((IObjectContextAdapter)dbContext).ObjectContext;
        objectContext.Refresh(mode, collection);
    }

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

В этом же классе присутствует ещё несколько расширений.

Метод `NextValueFor` предназначен для получения следующего значения генератора. Метод `dbContext.Database.SqlQuery` позволяет выполнять SQL запросы напрямую и отображать их результаты на некоторую сущность (проекцию). Вы можете воспользоваться им, если вам потребуется выполнить SQL запрос напрямую.

Метод `DetachAll` предназначен для отсоединения всех объектов коллекции DBSet от контекста. Это необходимо для обновления внутреннего кеша. Дело в том, что в рамках контекста все извлекаемые кешируются и не извлекаются из базы данных снова. Однако это не всегда полезно, поскольку затрудняет получение изменённых записей сделанных в другом контексте.

Замечание
В Web приложениях контекст обычно живёт очень короткое время, а новый контекст имеет не заполненный кеш.

Метод `Refresh` предназначен для обновления свойств объекта-сущности. Он полезен для обновления свойств объекта после его редактирования или добавления.

Таким образом, наш код загрузки данных будет выглядеть так

```
        private void LoadCustomersData()
        {
            var dbContext = AppVariables.getDbContext();
            // отсоединяем все загруженные объекты
            // это необходимо чтобы обновился внутренний кеш
            // при второй и последующих вызовах этого метода
            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";

        }
```

Код обработчика события на нажатие кнопки добавления выглядит следующим образом.

```
private void btnAdd_Click(object sender, EventArgs e) {
    var dbContext = AppVariables.getDbContext();
    // создание нового экземпляра сущности
    var customer = (CUSTOMER)bindingSource.AddNew();
    // создаём форму для редактирования
    using (CustomerEditorForm editor = new CustomerEditorForm()) {
        editor.Text = "Добавление заказчика";
        editor.Customer = customer;
        // Обработчик закрытия формы
        editor.FormClosing += delegate (object fSender, FormClosingEventArgs fe) {
           if (editor.DialogResult == DialogResult.OK) {
              try {
                 // получаем новое значение генератора
                 // и присваиваем его идентификатору
```

```
                customer.CUSTOMER_ID = dbContext.NextValueFor("GEN_CUSTOMER_ID");
                // добавляем нового заказчика
                dbContext.CUSTOMERS.Add(customer);
                // пытаемся сохранить изменения
                dbContext.SaveChanges();
                // и обновить текущую запись
                dbContext.Refresh(RefreshMode.StoreWins, customer);
            }
            catch (Exception ex) {
                // отображаем ошибку
                MessageBox.Show(ex.Message, "Error");
                // не закрываем форму для возможности исправления ошибки
                fe.Cancel = true;
            }
        }
        else
            bindingSource.CancelEdit();

    };
    // показываем модальную форму
    editor.ShowDialog(this);
  }
}
```

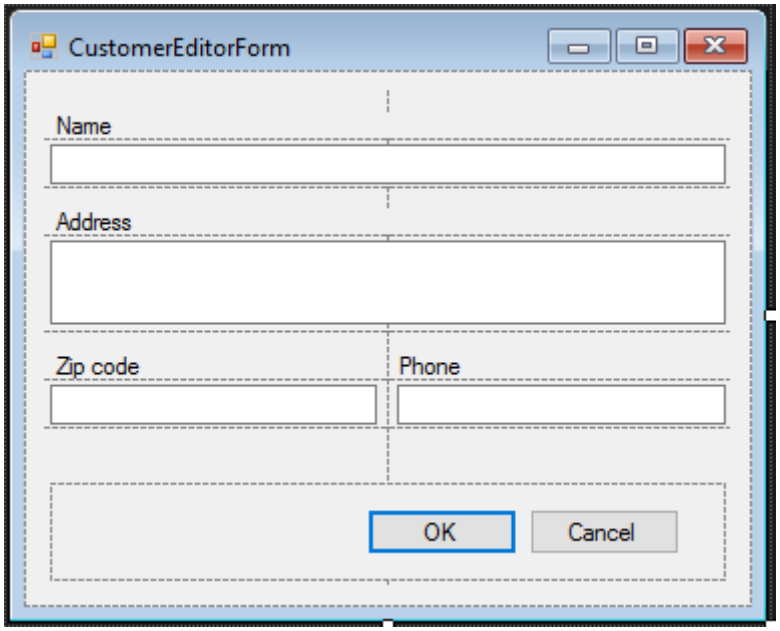
При добавлении новой записи мы получаем значение следующего идентификатора с помощью генератора. Мы могли бы не инициализировать значение идентификатора, и в этом случае отработал бы BEFORE INSERT триггер, который всё равно дёрнул бы следующее значение генератора. Однако в этом случае мы не смогли бы обновить вновь добавленную запись.

Код обработчика события на нажатие кнопки редактирования выглядит следующим образом.

```
private void btnEdit_Click(object sender, EventArgs e) {
    var dbContext = AppVariables.getDbContext();
    // получаем сущность
    var customer = (CUSTOMER)bindingSource.Current;
    // создаём форму для редактирования
    using (CustomerEditorForm editor = new CustomerEditorForm()) {
        editor.Text = "Редактирование заказчика";
        editor.Customer = customer;
        // Обработчик закрытия формы
        editor.FormClosing += delegate (object fSender, FormClosingEventArgs fe) {
            if (editor.DialogResult == DialogResult.OK) {
                try {
                    // пытаемся сохранить изменения
                    dbContext.SaveChanges();
                    dbContext.Refresh(RefreshMode.StoreWins, customer);
                    // обновляем все связанные контролы
                    bindingSource.ResetCurrentItem();
                }
                catch (Exception ex) {
                    // отображаем ошибку
                    MessageBox.Show(ex.Message, "Error");
                    // не закрываем форму для возможности исправления ошибки
                    fe.Cancel = true;
                }
            }
            else
                bindingSource.CancelEdit();

        };
```

```
        // показываем модальную форму
        editor.ShowDialog(this);
    }
}
```

Форма для редактирования заказчика выглядит следующим образом.



Код привязки к данным очень прост.

```
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");
}
```

Код обработчика события на нажатие кнопки удаления выглядит следующим образом.

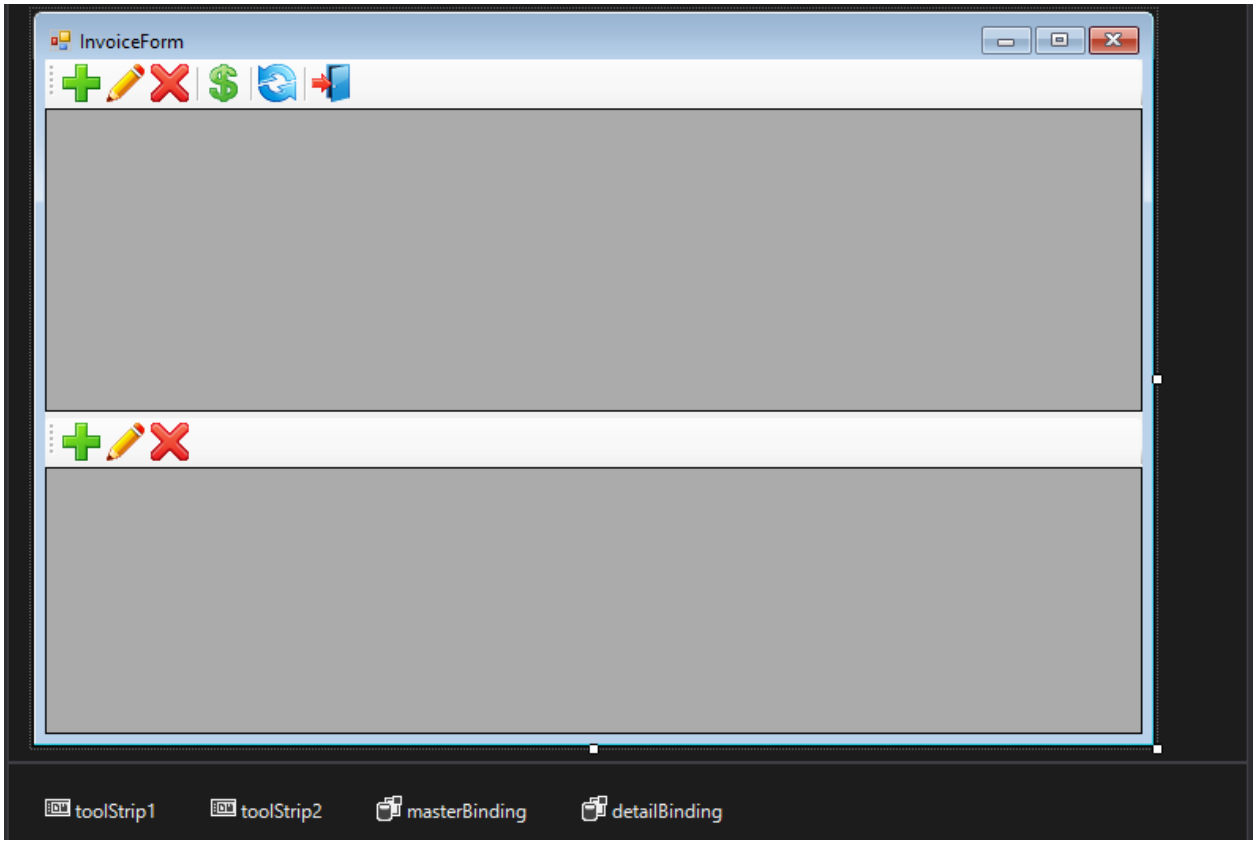
```
private void btnDelete_Click(object sender, EventArgs e) {
    var dbContext = AppVariables.getDbContext();
    var result = MessageBox.Show("Вы действительно хотите удалить заказчика?",
                "Подтверждение",
                MessageBoxButtons.YesNo,
                MessageBoxIcon.Question);
    if (result == DialogResult.Yes) {
       // получаем сущность
       var customer = (CUSTOMER)bindingSource.Current;
       try {
           dbContext.CUSTOMERS.Remove(customer);
           // пытаемся сохранить изменения
           dbContext.SaveChanges();
           // удаляем из связанного списка
           bindingSource.RemoveCurrent();
       }
       catch (Exception ex) {
           // отображаем ошибку
           MessageBox.Show(ex.Message, "Error");
```

```
        }
    }
}
```

Журналы

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 DataGridView, к каждому из которых привязать свой BindingSource



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 модуле AppVariables (см.

Получение контекста), 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.

Поскольку чаще всего требуются именно последние введённые документы, то имеет смысл сортировать их по дате в обратном порядке. Извлекать данные, как и в случае со справочниками будем при помощи LINQ. С учётом вышесказанного, метод для загрузки данных шапок счёт-фактур будет выглядеть следующим образом:

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

    // запрос на LINQ преобразуется в SQL
    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,
            Cusomer_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();
}
```

В качестве проекции мы использовали не анонимный тип, а класс `InvoiceView`. Это упрощает приведение типа. Определение класса InvoiceView выглядит следующим образом:

```
public class InvoiceView {
    public int Id { get; set; }
    public int Cusomer_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,
                Cusomer_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.Cusomer_Id = invoiceView.Cusomer_Id;
        this.Customer = invoiceView.Customer;
        this.Date = invoiceView.Date;
        this.Amount = invoiceView.Amount;
        this.Payed = invoiceView.Payed;
    }
}
```

Метод Load позволяет нам быстро обновить 1 добавленную или обновлённую запись в гриде, вместо того чтобы полностью перезагружать все записи.

Код обработчика события на нажатие кнопки добавления выглядит следующим образом.

```
private void btnAddInvoice_Click(object sender, EventArgs e) {
    var dbContext = AppVariables.getDbContext();
    var invoice = dbContext.INVOICES.Create();

    using (InvoiceEditorForm editor = new InvoiceEditorForm()) {
        editor.Text = "Добавление счёт фактуры";
        editor.Invoice = invoice;
        // Обработчик закрытия формы
        editor.FormClosing += delegate (object fSender, FormClosingEventArgs fe) {
            if (editor.DialogResult == DialogResult.OK) {
                try {
                    // получаем значение генератора
                    invoice.INVOICE_ID = dbContext.NextValueFor("GEN_INVOICE_ID");
                    // добавляем запись
                    dbContext.INVOICES.Add(invoice);
                    // пытаемся сохранить изменения
                    dbContext.SaveChanges();
                    // добавляем проекцию в список для грида
                    ((InvoiceView)masterBinding.AddNew()).Load(invoice.INVOICE_ID);
                }
                catch (Exception ex) {
                    // отображаем ошибку
                    MessageBox.Show(ex.Message, "Error");
                    // не закрываем форму для возможности исправления ошибки
                    fe.Cancel = true;
                }
            }
        };
        // показываем модальную форму
        editor.ShowDialog(this);
    }
}
```

В отличие от аналогичного метода справочника здесь обновление записи происходит не с помощью вызова `dbContext.Refresh`, а с помощью метода Load проекции `InvoiceView`. Дело в том, что `dbContext.Refresh` предназначен для обновления объектов сущностей, а не произвольных проекций, которые могут получаться сложными LINQ запросами.

Код обработчика события на нажатие кнопки редактирования выглядит следующим образом.

```
private void btnEditInvoice_Click(object sender, EventArgs e) {
    // получение контекста
    var dbContext = AppVariables.getDbContext();
    // поиск сущности по идентификатору
    var invoice = dbContext.INVOICES.Find(this.CurrentInvoice.Id);

    if (invoice.PAYED == 1) {
        MessageBox.Show("Изменение не возможно, счёт фактура уже оплачена.", "Ошибка");
        return;
    }

    using (InvoiceEditorForm editor = new InvoiceEditorForm()) {
        editor.Text = "Edit invoice";
        editor.Invoice = invoice;
        // Обработчик закрытия формы
        editor.FormClosing += delegate (object fSender, FormClosingEventArgs fe) {
            if (editor.DialogResult == DialogResult.OK) {
                try {
                    // пытаемся сохранить изменения
                    dbContext.SaveChanges();
                    // перезагружаем проекцию
                    CurrentInvoice.Load(invoice.INVOICE_ID);
                    masterBinding.ResetCurrentItem();
                }
                catch (Exception ex) {
                    // отображаем ошибку
                    MessageBox.Show(ex.Message, "Error");
                    // не закрываем форму для возможности исправления ошибки
                    fe.Cancel = true;
                }
            }
        };
        // показываем модальную форму
        editor.ShowDialog(this);
    }
}
```

Здесь нам потребовалось найти сущность по её идентификатору доступному в текущей записи. Свойство CurrentInvoice предназначено для получения выделенной в гриде счёт-фактуры. Оно реализовано так:

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

Удаление шапки счёт фактуры вы можете сделать самостоятельно.

Помимо добавления, редактирования и удаления для счёт-фактур мы ввели ещё одну операцию оплаты, код метода реализующего эту операцию выглядит следующим образом:

```
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("Изменение не возможно, счёт фактура уже оплачена.");

        invoice.PAYED = 1;
        // сохраняем изменения
        dbContext.SaveChanges();
        // перезагружаем изменённую запись
        CurrentInvoice.Load(invoice.INVOICE_ID);
        masterBinding.ResetCurrentItem();
    }
    catch (Exception ex) {
        // отображаем ошибку
        MessageBox.Show(ex.Message, "Ошибка");
    }
}
```

Для отображения позиций счёт-фактуры существует два метода:

1. Получать данные по каждой счёт-фактуре из навигационного свойства INVOICE_LINE и отображать содержимое этого сложного свойства (возможно с преобразованиями LINQ) в детейл гриде.
2. Получать данные по каждой счёт-фактуре отдельным LINQ запросом, который будет перевыполняться при перемещении в указателя в мастер гриде.

Каждый из методов имеет свои преимущества и недостатки.

Первый метод предполагает, что при открытии формы счёт-фактуры мы должны сразу извлечь все счёт-фактуры за указанный период и связанные данные по их позициям. Это хоть и выполняется одним SQL запросом, но может занять довольно много времени, и требует значительного объёма оперативной памяти. Этот метод лучше подходит для WEB приложений где вывод записей обычно происходит постранично.

Второй метод несколько более сложен в реализации, но позволяет быстро открывать форму счёт-фактуры и менее требователен к ресурсам, однако при каждом перемещении указателя в мастер гриде будет перевыполняться SQL запрос и загружать сетевой трафик (хотя объём будет невелик).

В нашем приложении я буду использовать второй подход. Для этого необходимо написать обработчик события изменения текущей записи для компонента BindingSource.

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

}
```

Метод для загрузки данных о позициях счёт-фактуры выглядит следующим образом:

```
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();

}
```

В качестве проекции мы использовали класс `InvoiceLineView`.

```
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; }
}
```

Замечу, что в отличие от класса InvoiceView здесь отсутствует метод для загрузки одной текущей записи. Здесь скорость перезагрузки детейл грида не настолько критична, поскольку один документ не содержит тысячи позиций, однако при желании вы можете реализовать такой метод.

Добавим специальное свойство для получения текущей строки документа выделенной в детейл гриде.

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

В методах для добавления, редактирования и удаления мы покажем, как работать с хранимыми процедурами в Entity Framework. Например, метод для добавления новой записи выглядит так:

```
private void btnAddInvoiceLine_Click(object sender, EventArgs e) {
    var dbContext = AppVariables.getDbContext();
    // получаем текущую счёт-фактуру
    var invoice = dbContext.INVOICES.Find(this.CurrentInvoice.Id);
    // проверяем не оплачена ли счёт-фактура
    if (invoice.PAYED == 1) {
        MessageBox.Show("Невозможно изменение, счёт-фактура оплачена.", "Error");
        return;
    }
    // создаём позицию счёт-фактуры
    var invoiceLine = dbContext.INVOICE_LINES.Create();
```

```
    invoiceLine.INVOICE_ID = invoice.INVOICE_ID;
    // создаём редактор позиции счёт фактуры
    using (InvoiceLineEditorForm editor = new InvoiceLineEditorForm()) {
        editor.Text = "Add invoice line";
        editor.InvoiceLine = invoiceLine;
        // Обработчик закрытия формы
        editor.FormClosing += delegate (object fSender, FormClosingEventArgs fe) {
            if (editor.DialogResult == DialogResult.OK) {
                try {
                   // создаём параметры ХП
                   var invoiceIdParam = new FbParameter("INVOICE_ID", FbDbType.Integer);
                   var productIdParam = new FbParameter("PRODUCT_ID", FbDbType.Integer);
                   var quantityParam = new FbParameter("QUANTITY", FbDbType.Integer);
                   // инициализируем параметры значениями
                   invoiceIdParam.Value = invoiceLine.INVOICE_ID;
                   productIdParam.Value = invoiceLine.PRODUCT_ID;
                   quantityParam.Value = invoiceLine.QUANTITY;
                   // выполняем хранимую процедуру
                   dbContext.Database.ExecuteSqlCommand(
                     "EXECUTE PROCEDURE SP_ADD_INVOICE_LINE(@INVOICE_ID, @PRODUCT_ID,
@QUANTITY)",
                     invoiceIdParam,
                     productIdParam,
                     quantityParam);
                   // обновляем гриды
                   // перезагрузка текущей записи счёт-фактуры
                   CurrentInvoice.Load(invoice.INVOICE_ID);
                   // перезагрузка всех записей детейл грида
                   LoadInvoiceLineData(invoice.INVOICE_ID);
                   // обновляем связанные данные
                   masterBinding.ResetCurrentItem();
                }
                catch (Exception ex) {
                   // отображаем ошибку
                   MessageBox.Show(ex.Message, "Error");
                   // не закрываем форму для возможности исправления ошибки
                   fe.Cancel = true;
                }
            }
        };
        // показываем модальную форму
        editor.ShowDialog(this);
    }
}
```

Здесь обновление записи мастер грида требуется потому, что одно из его полей (TotalSale) содержит агрегированную информацию по строкам документа.

Метод для обновления записи реализован так.

```
private void btnEditInvoiceLine_Click(object sender, EventArgs e) {
    var dbContext = AppVariables.getDbContext();
    // получаем текущую счёт-фактуру
    var invoice = dbContext.INVOICES.Find(this.CurrentInvoice.Id);
    // проверяем не оплачена ли счёт-фактура
    if (invoice.PAYED == 1) {
        MessageBox.Show("Изменение не возможно, счёт фактура оплачена.", "Error");
        return;
    }
    // получаем текущую позицию счёт-фактуры
    var invoiceLine = invoice.INVOICE_LINES
        .Where(p => p.INVOICE_LINE_ID == this.CurrentInvoiceLine.Id)
        .First();
    // создаём редактор позиции счёт фактуры
```

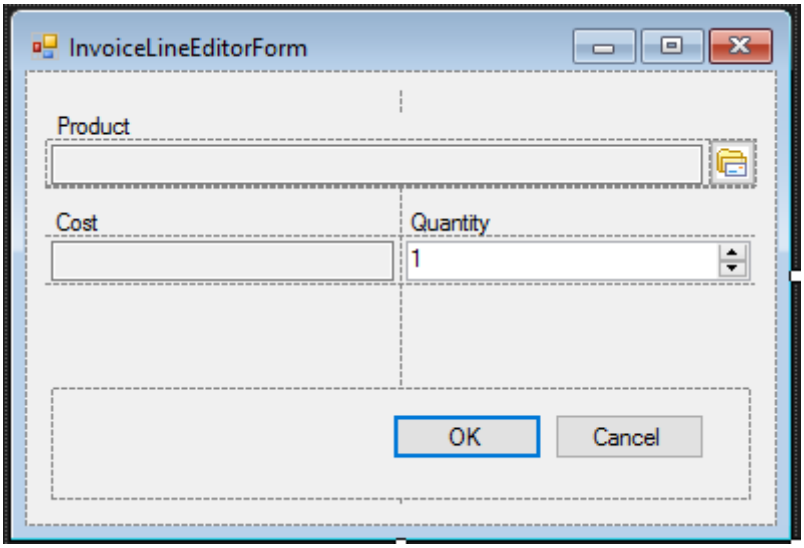
```
    using (InvoiceLineEditorForm editor = new InvoiceLineEditorForm()) {
        editor.Text = "Edit invoice line";
        editor.InvoiceLine = invoiceLine;

        // Обработчик закрытия формы
        editor.FormClosing += delegate (object fSender, FormClosingEventArgs fe) {
           if (editor.DialogResult == DialogResult.OK) {
               try {
                   // создаём параметры ХП
                   var idParam = new FbParameter("INVOICE_LINE_ID", FbDbType.Integer);
                   var quantityParam = new FbParameter("QUANTITY", FbDbType.Integer);
                   // инициализируем параметры значениями
                   idParam.Value = invoiceLine.INVOICE_LINE_ID;
                   quantityParam.Value = invoiceLine.QUANTITY;
                   // выполняем хранимую процедуру
                   dbContext.Database.ExecuteSqlCommand(
                      "EXECUTE PROCEDURE SP_EDIT_INVOICE_LINE(@INVOICE_LINE_ID,
@QUANTITY)",
                      idParam,
                      quantityParam);
                   // обновляем гриды
                   // перезагрузка текущей записи счёт-фактуры
                   CurrentInvoice.Load(invoice.INVOICE_ID);
                   // перезагрузка всех записей детейл грида
                   LoadInvoiceLineData(invoice.INVOICE_ID);
                   // обновляем связанные контролы
                   masterBinding.ResetCurrentItem();
               }
               catch (Exception ex) {
                   // отображаем ошибку
                   MessageBox.Show(ex.Message, "Error");
                   // не закрываем форму для возможности исправления ошибки
                   fe.Cancel = true;
               }
           }
       };

       // показываем модальную форму
       editor.ShowDialog(this);
    }
}
```

Метод для удаления записи реализован так.

```
private void btnDeleteInvoiceLine_Click(object sender, EventArgs e) {
    var result = MessageBox.Show("Вы действительно хотите удалить строку счёт-фактуры?",
                 "Подтверждение",
                 MessageBoxButtons.YesNo,
                 MessageBoxIcon.Question);
    if (result == DialogResult.Yes) {
       var dbContext = AppVariables.getDbContext();
       // получаем текущую счёт-фактуру
       var invoice = dbContext.INVOICES.Find(this.CurrentInvoice.Id);
       try {
           // проверяем не оплачена ли счёт-фактура
           if (invoice.PAYED == 1)
              throw new Exception("Не возможно удалить запись, счёт-фактура оплачена.");
           // создаём параметры ХП
           var idParam = new FbParameter("INVOICE_LINE_ID", FbDbType.Integer);
           // инициализируем параметры значениями
           idParam.Value = this.CurrentInvoiceLine.Id;
           // выполняем хранимую процедуру
           dbContext.Database.ExecuteSqlCommand(
              "EXECUTE PROCEDURE SP_DELETE_INVOICE_LINE(@INVOICE_LINE_ID)",
              idParam);
```

```
            // обновляем гриды
            // перезагрузка текущей записи счёт-фактуры
            CurrentInvoice.Load(invoice.INVOICE_ID);
            // перезагрузка всех записей детейл грида
            LoadInvoiceLineData(invoice.INVOICE_ID);
            // обновляем связанные контролы
            masterBinding.ResetCurrentItem();
        }
        catch (Exception ex) {
            // отображаем ошибку
            MessageBox.Show(ex.Message, "Error");
        }
    }
}
```

В методах для добавления и редактирования позиций счёт-фактуры мы использовали форму для редактирования.



Для отображения товара мы будем использовать TextBox. По нажатию кнопки, расположенной рядом, будет вызываться модальная форма с гридом для выбора товара. В качестве модальной окна для выбора продукта используем ту же форму, что была создана для их отображения. Код обработчика нажатия кнопки и инициализации формы будет выглядеть следующим образом:

```
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");
       }
    }

}
```

Работа с транзакциями

Когда мы вызываем при добавлении, обновлении, удалении метод SaveChanges(), то фактически Entity Framework неявно стартует и завершает транзакцию. Поскольку используется отсоединённая модель, то все операции происходят в рамках одной транзакции. Кроме того EF автоматически стартует и завершает транзакцию при каждом извлечении данных. Рассмотрим работу автоматических транзакций на следующем примере. Допустим нам необходимо сделать скидку на товары, выделенные в гриде. Код без явного использования транзакций будет выглядеть следующим образом:

```
var dbContext = AppVariables.getDbContext();
foreach (DataGridViewRow gridRows in dataGridView.SelectedRows) {
    int id = (int)gridRows.Cells["Id"].Value;
    // здесь происходит неявный старт и завершение транзакции
    var product = dbContext.PRODUCTS.Find(id);
    // скидка 10%
    decimal discount = 10.0m;
    product.PRICE = product.PRICE * (100 - discount) /100;
}
// здесь происходит неявный старт и завершение транзакции
// все изменения происходят за одну транзакцию
dbContext.SaveChanges();
```

Допустим, мы выбрали 10 товаров. В этом случае будет неявно использовано 10 транзакций для поиска товара по идентификатору и одиннадцатая для сохранения изменений. В данном случае можно использовать всего одну транзакцию, если использовать явное управление транзакциями. Например, вот так:

```
var dbContext = AppVariables.getDbContext();
// явный старт транзакции по умолчанию
using (var dbTransaction = dbContext.Database.BeginTransaction()) {
    string sql =
         "UPDATE PRODUCT " +
         "SET PRICE =  PRICE * ROUND((100 - @DISCOUNT)/100, 2) " +
         "WHERE PRODUCT_ID = @PRODUCT_ID";
    try {
        // создаём параметры запроса
        var idParam = new FbParameter("PRODUCT_ID", FbDbType.Integer);
        var discountParam = new FbParameter("DISCOUNT", FbDbType.Decimal);
        // создаём SQL команду для обновления записей
        var sqlCommand = dbContext.Database.Connection.CreateCommand();
        sqlCommand.CommandText = sql;
```

```
            // указываем команде, какую транзакцию использовать
            sqlCommand.Transaction = dbTransaction.UnderlyingTransaction;
            sqlCommand.Parameters.Add(discountParam);
            sqlCommand.Parameters.Add(idParam);
            // подготавливаем команду
            sqlCommand.Prepare();
            // для всех выделенных записей в гриде
            foreach (DataGridViewRow gridRows in dataGridView.SelectedRows) {
                int id = (int)gridRows.Cells["Id"].Value;
                // инициализируем параметры запроса
                idParam.Value = id;
                discountParam.Value = 10.0m; // скидка 10%
                // выполняем sql запрос
                sqlCommand.ExecuteNonQuery();
            }
            dbTransaction.Commit();
        }
        catch (Exception ex) {
            dbTransaction.Rollback();
            MessageBox.Show(ex.Message, "error");
        }
    }
```

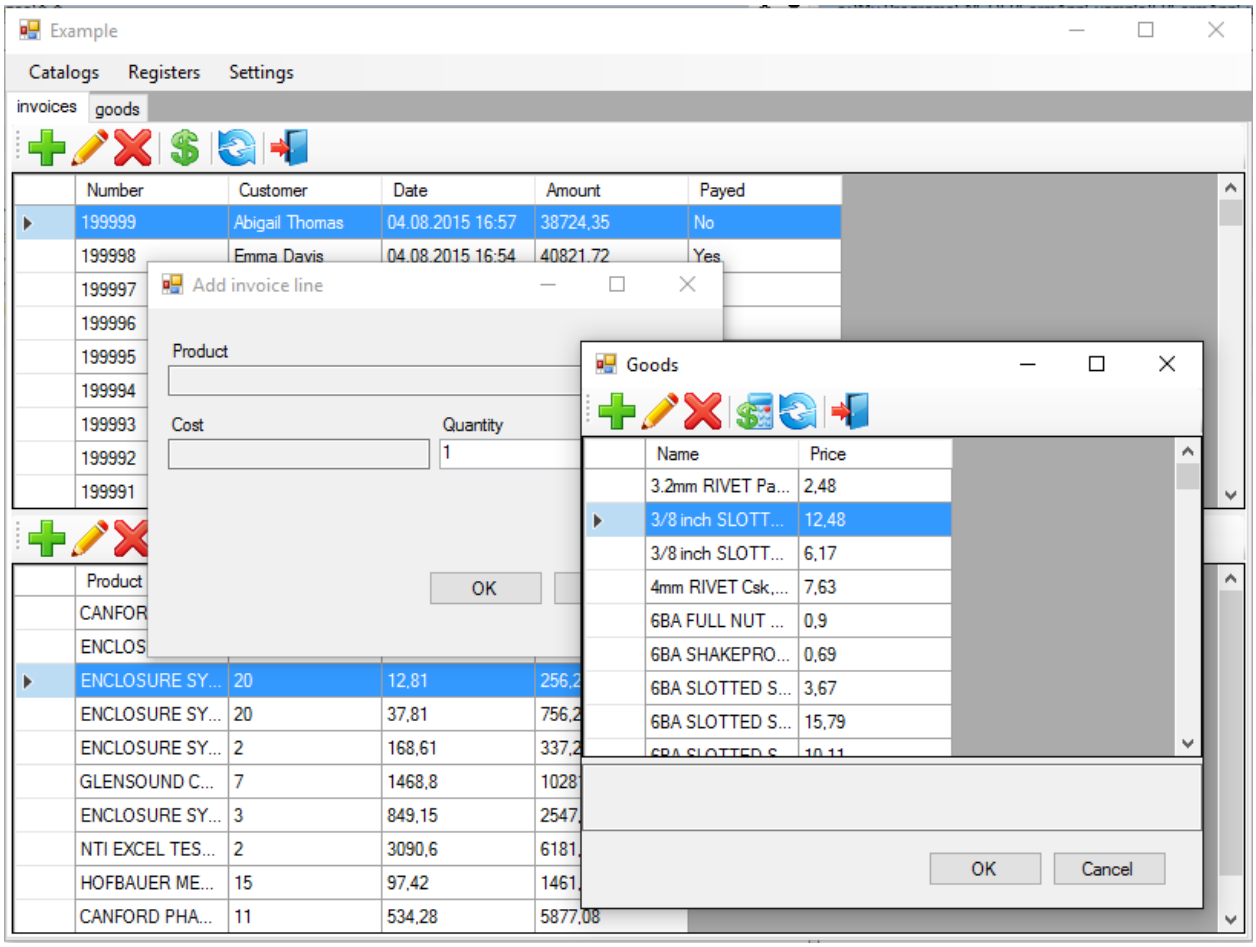
В данном случае мы стартовали транзакцию с параметрами по умолчанию. Для того чтобы задавать свои параметры транзакции необходимо использовать метод UseTransaction.

```
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;
    // явный старт транзакции по умолчанию
    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 {
            // создаём параметры запроса
            var idParam = new FbParameter("PRODUCT_ID", FbDbType.Integer);
            var discountParam = new FbParameter("DISCOUNT", FbDbType.Decimal);
            // создаём SQL комманду для обновления записей
            var sqlCommand = connection.CreateCommand();
            sqlCommand.CommandText = sql;
            // указываем команде какую транзакцию использовать
            sqlCommand.Transaction = dbTransaction;
            sqlCommand.Parameters.Add(discountParam);
            sqlCommand.Parameters.Add(idParam);
            // подготавливаем команду
            sqlCommand.Prepare();
            // для всех выделенных записей в гриде
            foreach (DataGridViewRow gridRows in dataGridView.SelectedRows) {
                int id = (int)gridRows.Cells["PRODUCT_ID"].Value;
```

```
                // инициализируем параметры запроса
                idParam.Value = id;
                discountParam.Value = editor.Discount;
                // выполняем sql запрос
                needUpdate = (sqlCommand.ExecuteNonQuery() > 0) || needUpdate;
            }
            dbTransaction.Commit();
        }
        catch (Exception ex) {
            dbTransaction.Rollback();
            MessageBox.Show(ex.Message, "error");
            needUpdate = false;
        }
    }
    // перезагружаем содержимое грида
    if (needUpdate) {
       // для всех выделенных записей в гриде
       foreach (DataGridViewRow gridRows in dataGridView.SelectedRows) {
            var product = (PRODUCT)bindingSource.List[gridRows.Index];
            dbContext.Refresh(RefreshMode.StoreWins, product);
       }
       bindingSource.ResetBindings(false);
    }

}
```

Ну вот. Теперь у нас для всего набора обновлений используется всего одна транзакция, и нет лишних команд для поиска данных. Осталось только добавить диалог для ввода значения скидки и обновление данных в гриде. Попробуйте сделать это самостоятельно.

Надеюсь, эта статья помогла вам разобраться в особенностях написания приложения на C# с использованием Entity Framework при работе с СУБД Firebird.



Ссылки

Исходные коды примера приложения
Готовая БД 2.5 и 3.0

www.ibase.ru, www.ibsurgeon.com

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