Урок №4

Содержание

1. Отсоединенный режим

До этого урока вы знали, что перед началом работы с базой данных вам необходимо открыть содержащий ее файл, выбрать нужную таблицу, переместиться на первую запись в ней и в цикле последовательно обходить все записи, считывая их содержимое или модифицируя их значение. В течение всего времени работы с базой данных вам необходимо поддерживать с ней непрерывный контакт.

Развитие всемирной сети внесло свои коррективы в этот алгоритм работы с базами данных. Идеология, при которой множество клиентов может быть одновременно подключено к одному серверу, и при этом отсутствует какая-либо возможность выяснить, активен ли сейчас каждый клиент или, напротив, давно отключился, очень плохо ложилась на существующие механизмы работы с базами данных. В результате был разработан новый алгоритм работы с базой данных: клиент подключается к базе, вычитывает себе порцию данных и отключается от сервера, продолжая обрабатывать данные на своей стороне. Если ему требуется удалить из

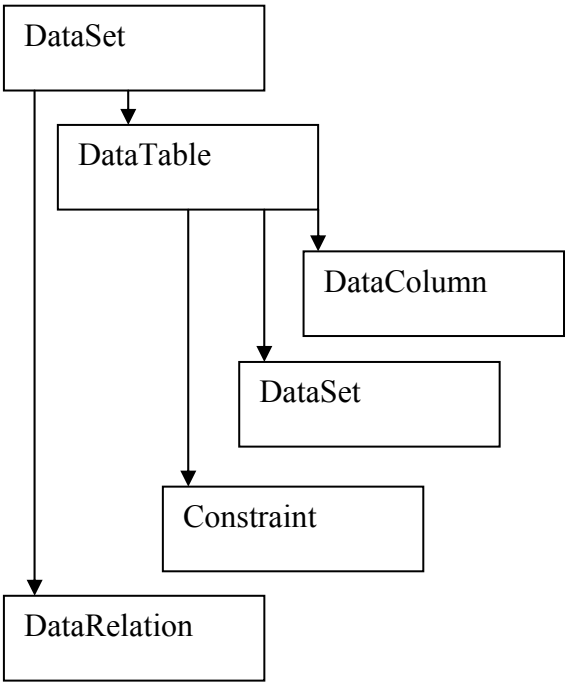
базы данных какие-то записи или модифицировать значение полей таблицы, он повторно подключается к серверу и посылает ему команду на совершение необходимых ему действий, после чего снова считывает очередную порцию данных и отключается, продолжая свою работу.

2. Концепция отсоединенного режима.

Концепция доступа к данным в **ADO .NET** основана на использовании двух компонентов:

- **Набора данных**, локального временного хранилища данных со стороны клиента.
- **Провайдера данных**, посредника, обеспечивающего взаимодействие приложения и базы данных со стороны базы данных (в распределенных приложениях – со стороны сервера).

Объектная модель **ADO .NET** для отсоединенного режима предполагает использование множества классов, выполняющих четко определенные задачи при работе с базой данных:



Классы отсоединенных объектов обеспечивают сохранение, использование и преобразование полученной от базы данных информации на стороне приложения.

3. Класс DataSet.

Класс **DataSet** является центральным классом в иерархии **ADO.NET**. Обрабатывая данные в программе, вы чаще всего будете общаться именно с ним. Класс **DataSet** содержит в себе набор из одной или нескольких таблиц, вместе с информацией об их полях и связях таблиц между собой. Также в **DataSet** содержится информация об **SQL** запросах, необходимых для заполнения этого объекта данными, модификации, вставки и удаления данных на сервере. Все четыре операции программируются и действуют независимо друг от друга.

При работе с **DataSet** различают типизированный и не типизированный **DataSet.** Типизированный объект **DataSet** представляет собой класс, производный от класса **DataSet**. Он наследует все методы, события и свойства класса **DataSet**. Кроме того, типизированный **DataSet** предоставляет строго типизированные методы, события и свойства. Это означает, что к таблицам и столбцам можно обращаться путем указания имен вместо использования методов на основе коллекций. Благодаря использованию строго типизированного объекта **DataSet** ошибки несоответствия типов выявляются во время компиляции кода, а не во время выполнения.

Основные методы класса **DataSet**:

Метод	**Описание**
AcceptChanges	Сохраняет все изменения, внесенные в класс **DataSet** после его загрузки или после последнего вызова метода

	AcceptChanges.
Clear	Удаляет из класса **DataSet** любые данные путем удаления всех строк во всех таблицах.
Clone	Копирует структуру класса **DataSet,** включая все схемы, соотношения и ограничения объекта **DataTable.** Данные не копируются.
Copy	Копирует структуру и данные для **DataSet.**
CreateDataReader	Возвращает объект **DataTableReader** с одним результирующим набором для каждой таблицы **DataTable** в той же последовательности, в которой таблицы отображаются в коллекции **Tables.**
GetChanges	Получает копию класса **DataSet,** содержащую все изменения, внесенные после его последней загрузки или после вызова метода **AcceptChanges.**
GetObjectData	Заносит в объект сведения о данных, необходимых для сериализации **DataSet.**
GetXml	Возвращает **XML**-представление данных, хранящихся в классе **DataSet.**
GetXmlSchema	Возвращает **XML**-схему для **XML**-представления данных, хранящихся в классе **DataSet.**
HasChanges	Получает значение, определяющее наличие изменений в классе **DataSet,** включая добавление, удаление или изменение строк.
Load	Заполняет таблицу **DataSet** значениями из источника данных с помощью предоставляемого объекта **IDataReader**.
Merge	Осуществляет слияние указанного класса **DataSet, DataTable** или массива объектов **DataRow** с текущим объектом **DataSet** или **DataTable.**
ReadXml	Считывает **XML**-схему и данные в **DataSet**.
ReadXmlSchema	Читает **XML**-схему в **DataSet.**
RejectChanges	Отменяет все изменения, внесенные в класс **DataSet** после его создания или после последнего вызова метода **AcceptChanges.**
Reset	Возвращает объект **DataSet** в исходное

	состояние. Для восстановления исходного состояния класса **DataSet** необходимо переопределить метод **Reset** в подклассах.
WriteXml	Записывает **XML**-данные и по возможности схемы из **DataSet**.
WriteXmlSchema	Записывает структуру класса **DataSet** в виде **XML**-схемы.

Основные свойства класса **DataSet**:

Метод	**Описание**
CaseSensitive	Возвращает или задает значение, определяющее, учитывается ли регистр при сравнении строк в объектах **DataTable**.
DataSetName	Возвращает или задает имя текущего **DataSet**.
HasErrors	Получает истину, если присутствуют ошибки в любом из объектов **DataTable** класса **DataSet**.
Tables	Возвращает коллекцию таблиц класса **DataSet**. К любой таблице из этой коллекции можно обратиться как по номеру, так и по имени.
Relations	Возвращает коллекцию отношений, связывающих таблицы и позволяющих переходить от родительских таблиц к дочерним. К любому отношению из этой коллекции можно обратиться как по номеру, так и по имени.
IsInitialized	Возвращает истину если набор данных **DataSet** проинициализирован.

Давайте рассмотрим пример, в котором будем вычитывать данные в **DataSet** из **XML** документа.

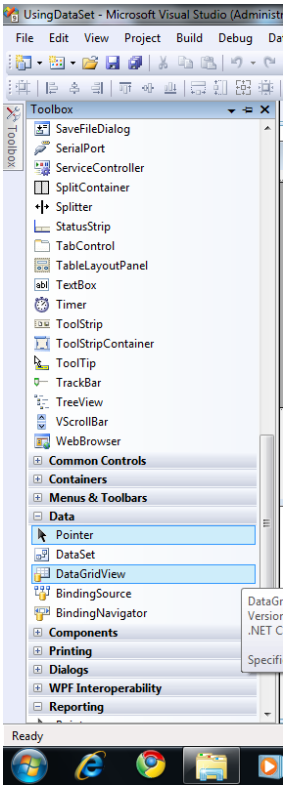
Для начала разработаем **XML** документ:

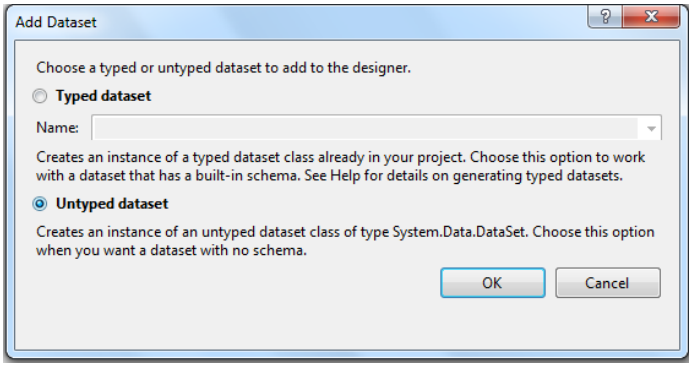
```
<?xml version="1.0" ?>
<library>
  <book>
    <name>BOOK1</name>
    <year>2006</year>
  </book>

 <book>
```

```
    <name>BOOK2</name>
    <year>2004</year>
  </book>

 <book>
    <name >BOOK3</name>
    <year>2007</year>
  </book>

</library>
```

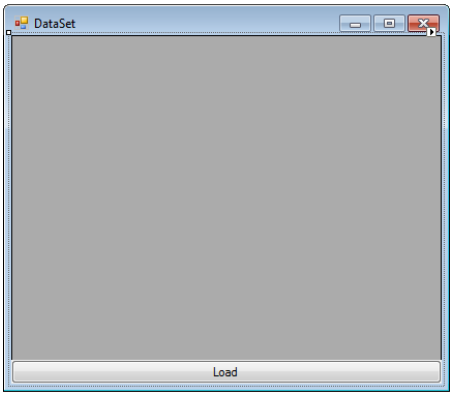
Назовем этот файл **1.xml**.

Создадим новый проект и перетащим с **ToolBox** два элемента управления **DataSet** **DataGridView**. При выборе **DataSet** укажите не типизированный (**Untyped dataset**).



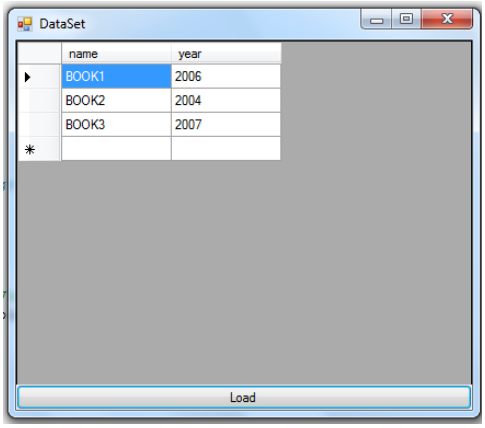


Кроме **DataGridView,** разместим на форме также кнопку «**Load**». Сделаем привязку для кнопки по нижнему краю (свойство **Dock** установим в **Bottom**), а для **DataGridView** по всей клиентской области (свойство **Dock** установим в **Fill**).



Добавим файл **1.xml** в папку с проектом и обработчик на кнопку:

```
private void Load_Click(object sender, EventArgs e)
        {
            try
            {
                //Загружаем данные в DataSet
                dataSet1.ReadXml("..\\..\\1.xml");
                //Привяка данных из DataSet к DataGridView
                dataGridView1.DataSource = dataSet1.Tables[0];
            }
            catch (System.Exception e1)
            {
                MessageBox.Show(e1.Message);
            }

        }
```



После клика на кнопку **Load** в **DataGridView** появиться содержимое **XML** файла.

4. Класс DataTable.

Таблица данных **DataTable** представляет одну таблицу с реляционными данными, размещенными в памяти приложения. Структура таблицы представляется столбцами и ограничениями. Столбцы таблицы могут сопоставляться со столбцами источника данных, содержать вычисляемые значения, автоматически увеличивать значения или содержать значения первичного ключа. **DataTable** также содержит строки, в которых хранятся сами данные.

Основные методы класса **DataTable**:

Метод	**Описание**
AcceptChanges	Фиксирует все изменения, внесенные в таблицу после создания или после последнего вызова метода **AcceptChanges.**
Clear	Очищает **DataTable** от всех данных.
Clone	Копирует структуру объекта **DataTable,** включая все схемы и ограничения **DataTable.** Данные не копируются.
Compute	Вычисляет заданное выражение для текущих строк, отвечающих условию фильтра.
Copy	Копирует структуру и данные для **DataTable.**
CreateDataReader	Возвращает объект **DataTableReader,** соответствующий данным в этой таблице **DataTable.**
GetChanges	Получает копию таблицы **DataTable,** содержащую все изменения, внесенные после ее последней загрузки или после вызова метода **AcceptChanges.**
GetErrors	Получает массив объектов **DataRow,** содержащих ошибки.
GetObjectData	Заносит в объект данные, необходимые для сериализации объекта **DataTable.**
ImportRow	Копирует объект **DataRow** в **DataTable,** сохраняя все параметры свойств, а также текущие и исходные значения.
Load	Заполняет таблицу **DataTable** значениями из источника данных с помощью предоставляемого объекта **IDataReader.** Если объект **DataTable** уже содержит строки, поступающие данные из источника данных

	объединяются с существующими строками.
LoadDataRow	Находит и обновляет конкретную строку. Если нужная строка не найдена, то с помощью заданных значений создается новая строка.
Merge	Объединяет заданный объект **DataTable** с текущим объектом **DataTable**.
NewRow	Создает новый класс **DataRow**, имеющий ту же схему, что и таблица.
NewRowFromBuilder	Создает новую строку из существующей строки.
ReadXml	Считывает **XML**-схему и данные в **DataTable**.
ReadXmlSchema	Считывает **XML**-схему в **DataTable**.
Reset	Возвращает объект **DataTable** в исходное состояние.
RejectChanges	Выполняется откат всех изменений, внесенных в таблицу с момента ее загрузки или после последнего вызова метода **AcceptChanges**.
Select	Получает массив объектов **DataRow**.
WriteXml	Записывает текущее содержимое таблицы **DataTable** в формате **XML**.
WriteXmlSchema	Записывает текущую структуру данных таблицы **DataTable** в виде **XML**-схемы.

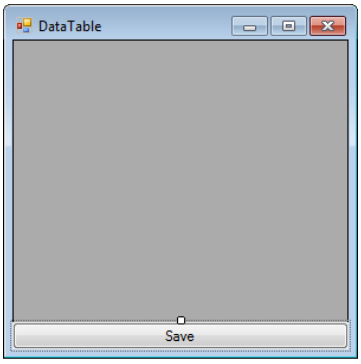
Основные свойства класса **DataTable**:

Метод	**Описание**
CaseSensitive	Показывает, учитывается ли регистр при сравнении строк в таблице.
ChildRelations	Получает коллекцию дочерних отношений для объекта **DataTable**.
Columns	Получает коллекцию столбцов, принадлежащих данной таблице. К столбцам можно обращаться как по имени так и по номеру.
Constraints	Получает коллекцию ограничений, содержащихся в данной таблице.
Container	Возвращает контейнер для компонента.
DataSet	Получает класс **DataSet**, к которому принадлежит данная таблица.
HasErrors	Возвращает истину, если есть ошибки в

	строках таблиц класса **DataSet**, к которому принадлежат таблицы.
IsInitialized	Возвращает истину, если таблица инициализирована.
Locale	Возвращает или задает сведения о языковом стандарте, используемые для сравнения строк таблицы.
ParentRelations	Получает коллекцию родительских отношений для объекта **DataTable**.
PrimaryKey	Возвращает или задает массив столбцов, которые являются столбцами первичного ключа для таблицы данных.
Rows	Получает коллекцию строк, принадлежащих данной таблице.
TableName	Возвращает или задает имя **DataTable**.

Давайте рассмотрим пример, в котором будем сохранять данные из **DataSet** в **XML.**

Создадим новый проект и перетащим с **ToolBox** два элемента управления **DataSet** **DataGridView**. Кроме **DataGridView,** разместим на форме также кнопку «**Save**». Сделаем привязку для кнопки по нижнему краю (свойство **Dock** установим в **Bottom**), а для **DataGridView** по всей клиентской области (свойство **Dock** установим в **Fill**).

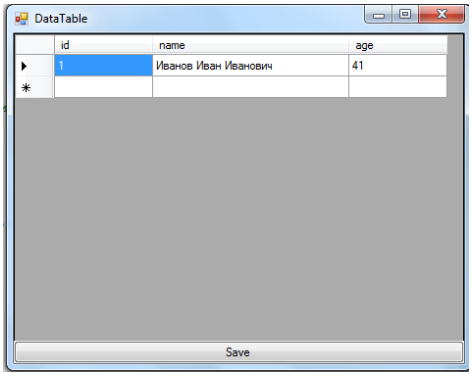


Добавим в конструктор формы инициализацию источника данных и таблицы и напишем обработчик для кнопки **Save**:

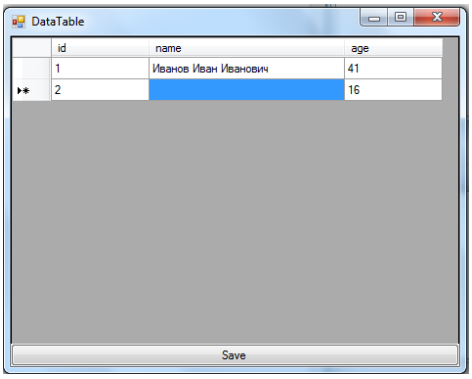
```
public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
            //Создаем таблицу с именем human
            DataTable human = dataSet1.Tables.Add("human");
            //Добавляем столбцы в таблицу
            human.Columns.Add("id", typeof(Int32));
            human.Columns.Add("name", typeof(String));
            human.Columns.Add("age", typeof(Int32));
            //Добавляем ограничение первичного ключа
```

```
            human.Constraints.Add("PK_Human", human.Columns["id"], true);
            //Делаем поле id счетчиком
            human.Columns[0].AutoIncrement = true;
            //Добавляем ограничение уникальности на поле name
            Constraint c = new UniqueConstraint(human.Columns["name"]);
            human.Constraints.Add(c);
            //задем полю age значение по умолчанию равное 16
            human.Columns["age"].DefaultValue = 16;

            //добавляем строку в таблицу
            human.Rows.Add(1, "Иванов Иван Иванович", 41);

            //Привяжем DataGridView к источнику данных
            dataGridView1.DataSource = dataSet1;
            //указываем отображаемую в текущий момент таблицу
            dataGridView1.DataMember = "human";
           //задим ширину второй колонки равной 200
            dataGridView1.Columns[1].Width = 200;


        }

        private void Save_Click(object sender, EventArgs e)
        {
            dataSet1.WriteXml("1.xml");
        }

    }
```

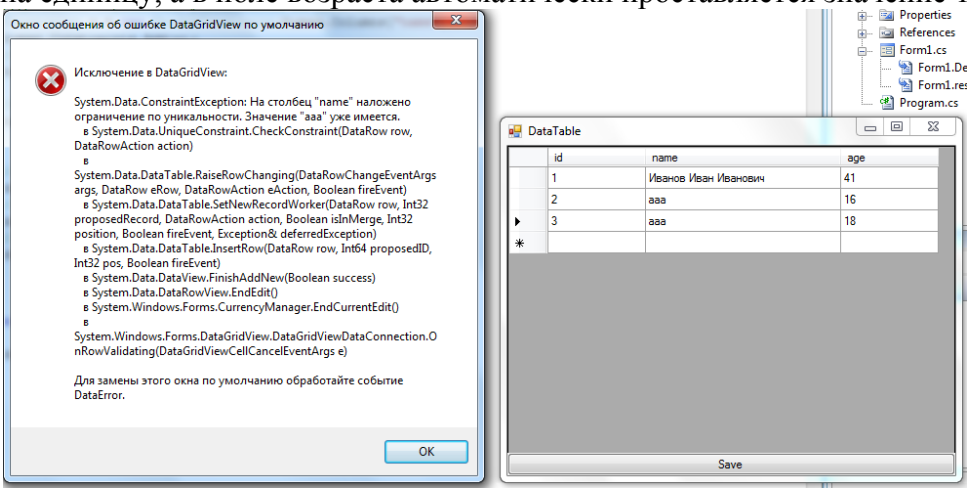
Вот что получилось:



Как видим, при запуске проекта сразу появляется первая запись, добавленная программно в конструкторе.



При добавлении новой записи, значение первичного ключа автоматически увеличивается на единицу, а в поле возраста автоматически проставляется значение 16.



При попытке добавить две разные записи с одинаковым значением в поле **name** выдается сообщение о нарушении ограничения уникальности.

После сохранения содержимого источника данных получаем **XML** файл следующего содержания:

```
<?xml version="1.0" standalone="yes"?>
<NewDataSet>
  <human>
    <id>1</id>
    <name>Иванов Иван Иванович</name>
    <age>41</age>
  </human>
  <human>
    <id>2</id>
    <name>aaa</name>
    <age>16</age>
  </human>
</NewDataSet>
```

5. Класс DataRow.

Класс **DataRow** описывает одну запись таблицы. Используйте объект **DataRow** и его свойства и методы для извлечения и оценки, вставки, удаления и обновления

значений в **DataTable**. Коллекция **DataRowCollection** представляет фактические объекты **DataRow** в **DataTable**.

Основные методы класса **DataRow**:

Метод	Описание
AcceptChanges	Фиксирует все изменения, внесенные в строку после создания или после последнего вызова метода **AcceptChanges.**
Delete	Удаляет объект **DataRow.**
ClearErrors	Очищает ошибки для строки. Это относится к свойству **RowError** и ошибкам, установленным с помощью метода **SetColumnError**.
GetChildRows	Получает дочерние строки объекта **DataRow**.
GetColumnError	Получает описание ошибки для столбца.
GetColumnsInError	Получает массив столбцов с ошибками.
GetParentRow	Получает родительскую строку объекта **DataRow.**
GetParentRows	Получает родительские строки объекта **DataRow.**
IsNull	Возвращает истину, если указанный столбец содержит значение null.
RejectChanges	Выполняется откат всех изменений, внесенных в строку с момента ее загрузки или после последнего вызова метода **AcceptChanges.**
SetAdded	Изменяет значение свойства **Rowstate** объекта **DataRow** на **Added.**
SetColumnError	Задает описание ошибки для столбца.
SetModified	Изменяет значение свойства **Rowstate** объекта **DataRow** на **Modified.**
SetNull	Задает значение **null** указанному параметру **DataColumn.**
SetParentRow	Задает родительскую строку объекта **DataRow.**

Основные свойства класса **DataRow**:

Метод	Описание
HasErrors	Возвращает истину, если есть ошибки в строке.
Item	Возвращает или задает данные, сохраненные в указанном столбце.
ItemArray	Возвращает или задает все значения для этой строки с помощью массива.
RowError	Возвращает или задает описание ошибки для строки.
RowState	Получает текущее состояние строки, относящееся к ее отношению к коллекции **DataRowCollection.**
Table	Получает объект **DataTable,** для которого эта строка имеет схему.

6. Класс DataColumn.

Класс **DataColumn** описывает столбец таблицы. Коллекция **DataColumnCollection** содержит объекты **DataColumn**, которые описывают схему **DataTable**.
DataColumn — базовый строительный блок для создания схемы **DataTable**. Построение схемы выполняется посредством добавлением одного или нескольких объектов **DataColumn** в **DataColumnCollection**.

Основные методы класса **DataColumn**:

Метод	Описание
SetOrdinal	Изменяет порядковый номер или положение **DataColumn** на указанный порядковый номер или положение.
ToString	Возвращает выражение **Expression** столбца, если такое существует.

Основные свойства класса **DataColumn**:

Метод	Описание
AllowDBNull	Возвращает истину, если столбец допускает нулевые значения.
AutoIncrement	Возвращает или задает значение, показывающее, увеличивать ли автоматически значение столбца для новых строк, добавляемых в таблицу.
AutoIncrementSeed	Возвращает или задает начальное

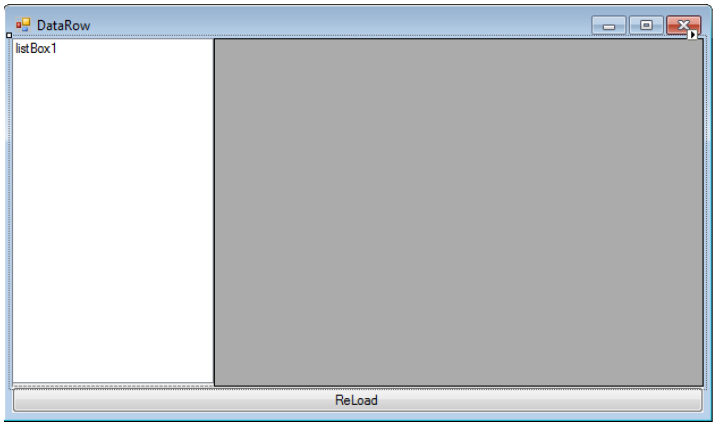
	значение для столбца, свойство которого **AutoIncrement** установлено на **true**.
AutoIncrementStep	Возвращает или задает инкремент для столбца, свойство **AutoIncrement** которого установлено на **true**.
Caption	Возвращает или задает заголовок для столбца.
ColumnName	Возвращает или задает имя столбца в **DataColumnCollection**.
DataType	Возвращает или задает тип данных, хранимых в столбце.
DefaltValue	Возвращает или задает значение по умолчанию для столбца при создании новых столбцов.
Expression	Возвращает или задает выражение, используемое для фильтрации строк, расчета значений в столбце либо создания составного столбца.
MaxLength	Возвращает или задает максимальную длину текстового столбца.
Ordinal	Возвращает положение столбца в коллекции **DataColumnCollection**.
Prefix	Возвращает или задает префикс **XML**, который является псевдонимом пространства имен класса **DataTable**.
ReadOnly	Возвращает или задает значение, указывающее на допустимость изменения столбца после добавления строки в таблицу.
Table	Получает **DataTable**, к которому принадлежит столбец.
Unique	Возвращает или задает значение, показывающее, должны ли значения в каждой строке столбца быть уникальными.

Давайте рассмотрим пример, в котором источник данных забирает данные из **XML** файла. Данные представлены двумя таблицами **gruppa**(**id**, **name**) и **student**(**id**, **fio**, **age**, **id_gruppa**). Будем вычитывать данные о группе в **ListBox** и при выделении соответствующей группы в окне **DataGridView** отображать всех студентов этой группы.

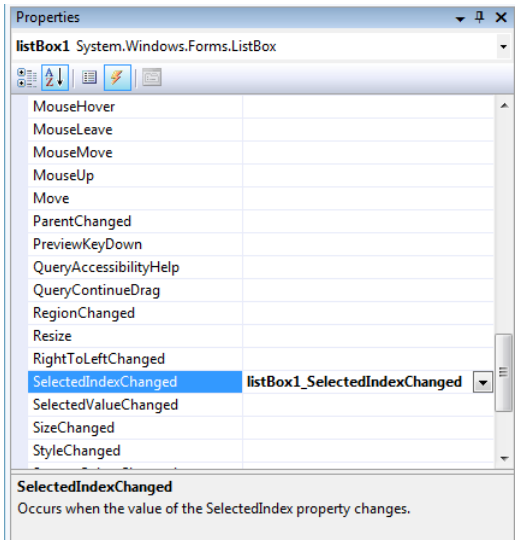
Для начала разработаем **XML** файл (не забудьте сохранить его в кодировке **UTF-8**):

```
<?xml version="1.0" encoding="UTF-8"?>
<NewDataSet>
  <student>
    <id>1</id>
     <fio>Иванов Иван Иванович</fio>
    <age>21</age>
    <id_gruppa>1</id_gruppa>
  </student>
  <student>
    <id>2</id>
    <fio>Куров Евгений Вадимович</fio>
    <age>31</age>
    <id_gruppa>1</id_gruppa>
  </student>
<student>
    <id>3</id>
    <fio>Сидоров Петр Иванович</fio>
    <age>25</age>
    <id_gruppa>1</id_gruppa>
</student>
<student>
    <id>4</id>
    <fio>Кривая Юлия Петровна</fio>
    <age>18</age>
    <id_gruppa>2</id_gruppa>
  </student>
<student>
    <id>5</id>
    <fio>Светлая Наталья Ивановна</fio>
    <age>22</age>
    <id_gruppa>2</id_gruppa>
  </student>
 <gruppa>
   <id>1</id>
   <name>СТВ-18-1</name>
 </gruppa>
 <gruppa>
   <id>2</id>
   <name>СТУ-18-1</name>
 </gruppa>
 <gruppa>
   <id>3</id>
   <name>СТД-18-1</name>
 </gruppa>
</NewDataSet>
```

Теперь создадим новый проект и разместим на нем кнопку «**Reload**», панель и **DataSet**. Привяжем кнопку по нижнему краю (свойство **Dock** установим в **Bottom)** , а панель по всей клиентской области (свойство **Dock** установим в **Fill)**. На панели расположим **ListBox** и **DataGridView**. Привяжем **ListBox** по левой стороне (свойство **Dock** установим в **Left)**, а **DataGridView** по всей клиентской области (свойство **Dock** установим в **Fill)**. Установим у **DataGridView** свойство **SelectionMode** в **FullRowSelect**.



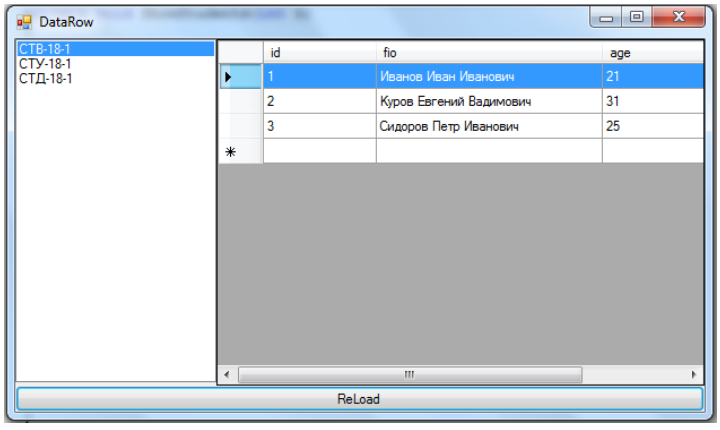
Напишем вспомогательные функции **InitDataSet** для инициализации источника данных (создание таблиц, установка ограничений и т.д.) и **ShowStudents** для отображения всех студентов выбранной группы. А также напишем конструктор, обработчик на кнопку **Reload** и обработку события **SelectedIndexChanged** у **ListBox**.

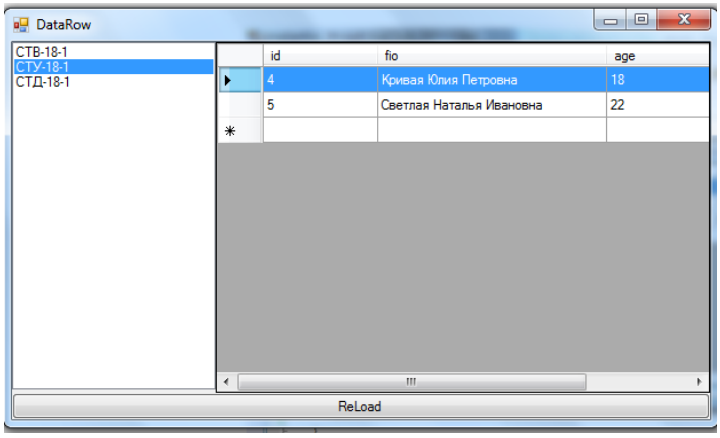



```
public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
            try
            {
                InitDataSet();
            }
            catch(System.Exception e1)
            {
                MessageBox.Show(e1.Message);
            }
        }

         private void InitDataSet()
        {
            //Создаем таблицу с именем student
```

```
            DataTable student = dataSet1.Tables.Add("student");
            //Добавляем столбцы в таблицу
            student.Columns.Add("id", typeof(Int32));
            student.Columns.Add("fio", typeof(String));
            student.Columns.Add("age", typeof(Int32));
            student.Columns.Add("id_gruppa", typeof(Int32));

            //Создаем таблицу с именем gruppa
            DataTable gruppa = dataSet1.Tables.Add("gruppa");
            //Добавляем столбцы в таблицу
            gruppa.Columns.Add("id", typeof(Int32));
            gruppa.Columns.Add("name", typeof(String));

            //Добавляем ограничения первичных ключей и делаем из счетчиком
            student.Constraints.Add("PK Student", student.Columns["id"],
true);
            student.Columns[0].AutoIncrement = true;
            gruppa.Constraints.Add("PK_Gruppa", gruppa.Columns["id"], true);
            gruppa.Columns[0].AutoIncrement = true;
             //также исключим возможность пустого значения в ключах
            student.Columns["id"].AllowDBNull = false;
            gruppa.Columns["id"].AllowDBNull = false;

            /*Добавляем ограничение уникальности на поле name таблицы gruppa
             и поле fio таблицы student        */
            Constraint c = new UniqueConstraint(gruppa.Columns["name"]);
            gruppa.Constraints.Add(c);
            c = new UniqueConstraint(student.Columns["fio"]);
            student.Constraints.Add(c);

            //Добавляем внешний ключ
            student.Constraints.Add("FK_Student", gruppa.Columns["id"],
student.Columns["id_gruppa"]);
           //скажем что внешний ключ не может содержать пустые строки
            student.Columns["id_gruppa"].AllowDBNull = false;
             //Добавляем отношение между этими таблицами
            dataSet1.Relations.Add("Stud_Gruppa",gruppa.Columns["id"],
student.Columns["id_gruppa"]);

             //Привяжем ListBox к источнику данных
            listBox1.DataSource = dataSet1.Tables["gruppa"];
            //указываем отображаеый столбец
            listBox1.DisplayMember = "name";

            //Вычитываем данные в таблицы из XML документа
            dataSet1.Tables["gruppa"].ReadXml("..\\..\\1.xml");
            /*
             Таблица студентов вычитывается автоматически, т.к. мы указали
отношение
             */

            ShowStudents(0);

        }

         private void ReLoad_Click(object sender, EventArgs e)
         {
             try
```

```
            {
                dataSet1 = new DataSet();
                 InitDataSet();
            }
            catch(System.Exception e1)
            {
                MessageBox.Show(e1.Message);
            }

        }

        private void ShowStudents(int n)
        {
            DataTable stud = dataSet1.Tables["student"].Clone();
            DataRow parent =dataSet1.Tables["gruppa"].Rows[n];
            DataRow[] childs = parent.GetChildRows("Stud_Gruppa");
            foreach (DataRow row in childs)
            {

stud.ImportRow(row);//.Rows.Add(row[0],row[1],row[2],row[3]);
            }
            dataGridView1.DataSource = stud;
            //задим ширину второй колонки равной 200
            dataGridView1.Columns[1].Width = 200;

        }

         private void listBox1_SelectedIndexChanged(object sender, EventArgs
e)
        {
            if (listBox1.SelectedIndex != -1)
            {
                ShowStudents(listBox1.SelectedIndex);
            }
        }
 }
```





7. Класс DbDataAdapter.

Класс **DbDataAdapter** предоставляет набор команд **SQL** и подключение базы данных, которые используются для заполнения объекта **DataSet** и обновления источника данных. Класс абстрактный. Для создания экземпляра класса используйте класс **SqlDataAdapter** из пространства имен **System.Data.SqlClient**, класс **OleDbDataAdapter** из пространства имен **System.Data.OleDb**, класс **OdbcDataAdapter** из пространства имен **System.Data.Odbc**.

Основные методы класса **DbDataAdapter**:

Метод	**Описание**
AddToBatch	Добавляет интерфейс **IDbCommand** к текущему пакету.
ClearBatch	Удаляет все объекты **IDbCommand** из пакета.
ExecuteBatch	Выполняет текущий пакет.
Fill	Заполняет объект **DataSet** или объект **DataTable**.
FillSchema	Добавляет объект **DataTable** в объект **DataSet** и настраивает схему для получения соответствия со схемой в источнике данных.
GetBatchedParemeter	Возвращает интерфейс **IDataParameter** из одной из команд в текущем пакете.
GetFillParameters	Получает параметры, заданные пользователем при выполнении оператора **SQL SELECT**.
InitializeBatching	Инициализирует пакетную обработку для объекта **DbDataAdapter**.
Update	Выполняет соответствующие команды **INSERT, UPDATE** или **DELETE** для каждой вставленной, обновленной или

	удаленной строки в объекте **DataSet**.

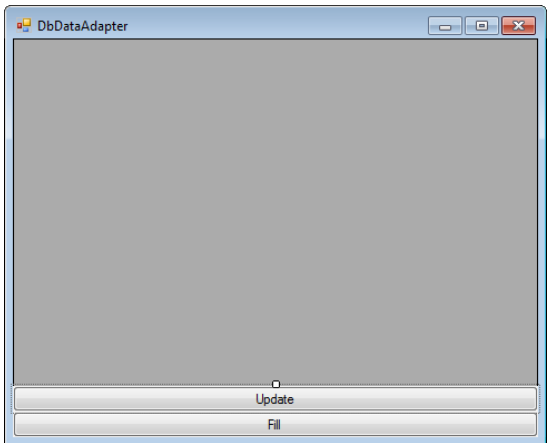
Основные свойства класса **DbDataAdapter**:

Метод	**Описание**
AcceptChangesDuringFill	Возвращает или задает значение, указывающее, вызывается ли метод **AcceptChanges** в объекте **DataRow** после его добавления к объекту **DataTable** при выполнении любой из операций **Fill**.
AcceptChangesDuringUpdate	Возвращает или задает, вызывается ли метод **AcceptChanges** при вызове метода **Update**.
DeleteCommand	Возвращает или задает команду для удаления записей из набора данных.
InsertCommand	Возвращает или задает команду **SQL** для вставки новых записей в источник данных.
MissingMappingAction	Определяет действие, выполняемое, если входные данные не соответствуют таблице или столбцу.
MissingSchemaAction	Определяет действие, которое должно быть выполнено, если существующая схема **DataSet** не соответствует входным данным.
SelectCommand	Возвращает или задает команду, используемую для выбора записей в источнике данных.
UpdateCommand	Возвращает или задает команду, используемую для обновления записей в источнике данных.

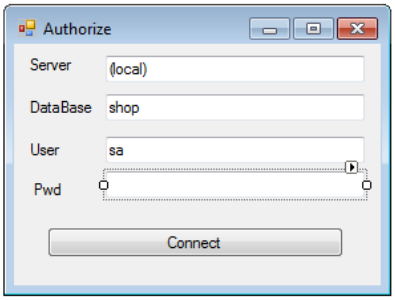
Давайте рассмотрим пример.
Для начала создадим базу данных **Shop** в СУБД **MS SqlServer 2008**. В ней создадим таблицу **Goods** и заполним ее данными.

```
create table goods
(
  id int primary key identity,
  name nvarchar(50),
  price float
)

insert into goods values('помидоры', 5.5);
insert into goods values('селедка', 20.1);
insert into goods values('бананы', 4.8);
```

Теперь создадим новый проект с двумя формами. На первой форме разместим две кнопки «**Update**» и «**Fill**», **DataGridView** и **DataSet**. Привяжем кнопки по нижнему краю (свойство **Dock** установим в **Bottom)** , а **DataGridView** по всей клиентской области (свойство **Dock** установим в **Fill)**. Установим у **DataGridView** свойство **SelectionMode** в **FullRowSelect**.



На второй форме расположим четыре текстовых поля «**Server**» «**DataBase**» «**User**» «**Pwd**», а также кнопку «**Connect**». Зададим для первых трех значения по умолчанию, а для поля **User** установим свойство **PasswordChar** равное «*».



Для второй формы разработаем свойства и напишем обработчик для кнопки:

```
public partial class Form2 : Form
    {
        public Form2()
        {
            InitializeComponent();
        }
        public string ServerName
        {
            get
            {
                return Server.Text;
            }
        }
        public string DBName
        {
            get
            {
                return DataBase.Text;
            }
```

```
        }
        public string UserName
        {
            get
            {
                return User.Text;
            }
        }
        public string Password
        {
            get
            {
                return Pwd.Text;
            }
        }

        private void button1_Click(object sender, EventArgs e)
        {
            DialogResult = DialogResult.OK;
        }
    }
```

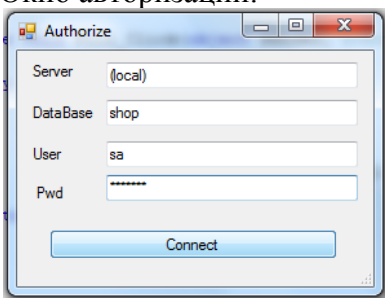
Для первой формы подключим пространства имен System.Data.Common и **System.Data.SqlClient** а также напишем конструктор и обработчики для кнопок:

```
public partial class Form1 : Form
    {
        DbDataAdapter adapter;
        DbConnection conn;
        public Form1()
        {
            InitializeComponent();
            //Запросим у пользователя параметры соединения
            Form2 form = new Form2();
            if (form.ShowDialog() == DialogResult.Cancel) Close();

            //Создаем соединение, открывать его не нужно
            conn = new SqlConnection();
            conn.ConnectionString = "Data Source="+form.ServerName+
                                    ";Initial Catalog="+form.DBName+
                                    ";User Id="+form.UserName+
                                    ";Password="+form.Password;
            //Создаем и формируем адаптер
            adapter = new SqlDataAdapter();
            //формируем команду на выборку
            DbCommand select = new SqlCommand();
            select.Connection = conn;
            select.CommandText = "Select * from goods";
            //добавляем команду к адаптеру
            adapter.SelectCommand = select;

            //формируем команду на добавление
            DbCommand insert = new SqlCommand();
            insert.Connection = conn;
            insert.CommandText = "insert into goods values(@n,@p)";
```

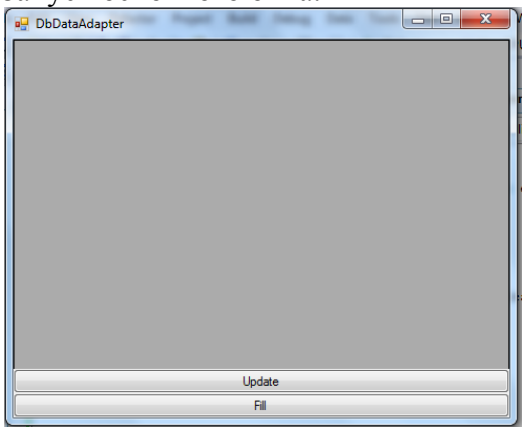
```
            //формируем параметры для команды
            DbParameter Name, Price;
            Name = new SqlParameter("n", SqlDbType.Text);
            Price = new SqlParameter("p", SqlDbType.Float);
            /*указываем из какий столбцов таблицы брать значения для
параметров*/
            Name.SourceColumn = "name";
            Price.SourceColumn = "price";
            //добавляем параметры к команде
            insert.Parameters.Add(Name);
            insert.Parameters.Add(Price);
            //добавляем команду к адаптеру
            adapter.InsertCommand = insert;

            //формируем команду на удаление
            DbCommand delete = new SqlCommand();
            delete.Connection = conn;
            delete.CommandText = "delete from goods where id=@id";
            //формируем параметр для команды
            DbParameter Id;
            Id = new SqlParameter("id", SqlDbType.Int);
            /*указываем из какого столбца таблицы брать значения для
параметра*/
            Id.SourceColumn = "id";
            //добавляем параметр к команде
            insert.Parameters.Add(Id);
            //добавляем команду к адаптеру
            adapter.DeleteCommand = delete;


        }

        private void Fill_Click(object sender, EventArgs e)
        {
            try
            {
                dataSet1.Clear();
                adapter.Fill(dataSet1);
                dataGridView1.DataSource = dataSet1.Tables[0];
            }
            catch (System.Exception e1)
            {
                MessageBox.Show(e1.Message);
            }
        }

        private void Update_Click(object sender, EventArgs e)
        {
            try
            {
                adapter.Update(dataSet1);

            }
            catch (System.Exception e1)
            {
                MessageBox.Show(e1.Message);
            }
        }
```

```
}
```

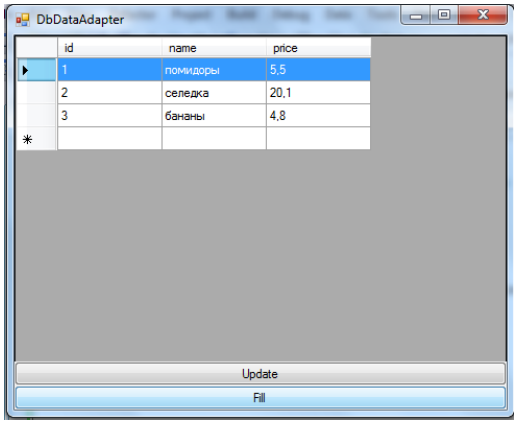
Анализируем то, что получилось:
Окно авторизации:



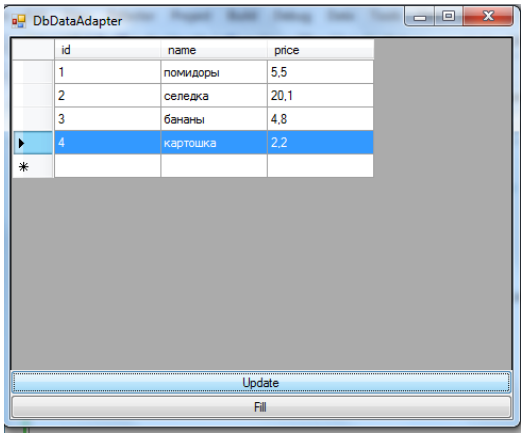
Запуск основного окна:



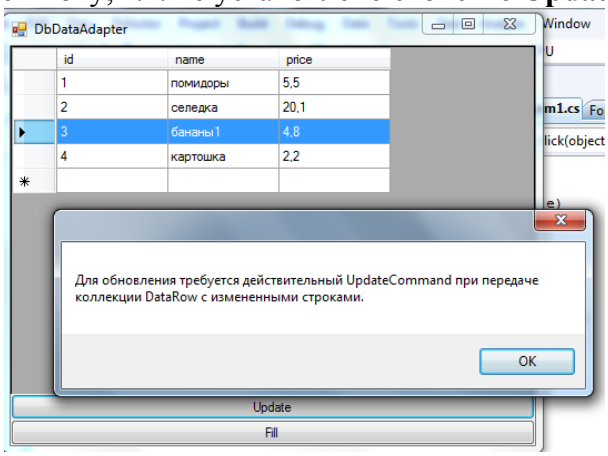
Нажата кнопка **Fill**:



Добавление новой записи:



При попытке изменить запись и сохранить изменения (кнопка **Update**) мы видим ошибку, т.к. не установлено свойство **UpdateCommand** у **DbDataAdapter**:



8. Класс SqlCommandBuilder.

Автоматически генерирует однотабличные команды, которые позволяют согласовать изменения, вносимые в объект **DataSet**, со связанной базой данных **SQL Server**. От этого класса нельзя наследоваться.

Основные методы класса **SqlCommandBuilder**:

Метод	Описание
ApplyParameterInfo	Разрешает реализации поставщика класса **DbCommandBuilder** обрабатывать дополнительные свойства параметров.
DeriveParameters	Извлекает сведения о параметрах из хранимой процедуры, указанной в объекте **SqlCommand**, и включает их в коллекцию параметров **Parameters** указанного объекта **SqlCommand**. Статический метод.
GetDeleteCommand	Получает автоматически созданный объект **SqlCommand**, который требуется

	для выполнения операций удаления в базе данных.
GetInsertCommand	Получает автоматически созданный объект **SqlCommand**, который требуется для выполнения операций вставки в базу данных.
GetParameterName	Получает имя параметра
GetSchemaTable	Возвращает таблицу схемы для объекта **DbCommandBuilder**.
GetUpdateCommand	Получает автоматически созданный объект **SqlCommand**, который требуется для выполнения операций редактирования в базе данных.
InitializeCommand	Сбрасывает свойства **CommandTimeout**, **Transaction**, **CommandType** и **UpdateRowSource** в объекте **DbCommand**.

Отредактируем пример из предыдущего раздела, изменим конструктор первой формы:

```
public Form1()
        {
            InitializeComponent();
            //Запросим у пользователя параметры соединения
            Form2 form = new Form2();
            if (form.ShowDialog() == DialogResult.Cancel) Close();

            //Создаем соединение, открывать его не нужно
            conn = new SqlConnection();
            conn.ConnectionString = "Data Source=" + form.ServerName +
                                    ";Initial Catalog=" + form.DBName +
                                    ";User Id=" + form.UserName +
                                    ";Password=" + form.Password;
            //Создаем и формируем адаптер
            adapter = new SqlDataAdapter();
            //формируем команду на выборку
            DbCommand select = new SqlCommand();
            select.Connection = conn;
            select.CommandText = "Select * from goods";
            //добавляем команду к адаптеру
            adapter.SelectCommand = select;

            /*Заполняем источник данных, это нужно сделать, иначе команды для
вставки,
             обновления и удаления не смогут быть сформированны, поскольку не
известна еще
             структура таблицы*/
            adapter.Fill(dataSet1);

            //формируем остальные команды
            //создаем SqlCommandBuilder
```

```
            SqlCommandBuilder builder = new
SqlCommandBuilder((SqlDataAdapter)adapter);
            //получаем команды
            adapter.InsertCommand = builder.GetInsertCommand();
            adapter.DeleteCommand = builder.GetDeleteCommand();
            adapter.UpdateCommand = builder.GetUpdateCommand();
}
```

Домашнее задание

1. Создайте приложение для хранения и редактирования пользователей. Необходимо хранить следующие данные о пользователе: логин, пароль, адрес, телефон, признак админа (логическая переменная). При добавлении нового пользователя, необходимо проверять, что логин с таким именем свободен. Для защиты пароля используйте функцию getHashCode у строки. Список пользователей отображается в ListBox. При выполнении двойного щелчка над записью, содержащей пользователя, открывается вторая форма для редактирования. Реализовать добавление и удаление пользователей. Добавление пользователя реализуется в новом окне. Также реализуйте вывод всех зарегистрированных пользователей с фильтрацией администраторов.

2. Создать базу (книги, авторы, издательство). На форме 2 чексбокса, 2 комбобокса, 1 кнопка и одна таблица для отображения данных. Изначально комбобоксы неактивны. Пользователь выбирает по какому признаку он будет искать книги (по автору, по издательству или по обоим полям сразу - чексбоксы). В зависимости от выбора пользователя становятся активными комбобоксы. В первом комбобоксе находятся списки всех авторов, во втором – всех издательств. Пользователь делает выбор, нажимает на кнопку «Поиск» и у него в таблице отображаются найденные книги.

3. Разработать приложение для магазина холодильников. Хранить товар, продавцов и покупателей. Все данные хранить в XML формате. Когда покупатель приобретает товар, оформляется чек. Чек сохраняется в XML формате по пути, указанному пользователем. Чек содержит дату, фио покупателя, фио продавца и купленные позиции.