Технология доступа к базам данных

ADO.NET

Урок №2

Отсоединенный режим работы

Содержание

Отсоединенный режим работы

В первом уроке вы рассмотрели использование присоединенного режима работы с БД. При этом способе работы мы создавали объект DbConnection, передавая ему строку подключения. Затем от имени этого объекта вызывали метод Open(), открывая подключение к БД. Дальше мы выполняли какие-либо действия, понимая, что подключение остается открытым до вызова метода Close().

Такой способ работы является оптимальным с точки зрения быстродействия, но не оптимальным, с точки зрения нагрузки на сервер. Так как серверу приходится держать открытыми подключения, по которым большую часть времени не выполняется никакого взаимодействия с БД.

В этом уроке вы изучите альтернативный способ работы с БД — отсоединенный режим. Этот режим базируется на использовании двух классов: DataSet и DbDataAdapter.

Для понимания этих двух классов сначала надо рассмотреть класс DataTable.

Табличные данные (DataTable)

Реляционные базы данных — это всегда таблицы. Поэтому неудивительно, что в .NET Framework есть ряд классов, предназначенных для хранения и отображения

табличных данных. Это такие графические элементы управления, как DataGridView и ListView и такой класс, как DataTable. Этот последний класс будет нам очень полезен в дальнейшей работе. Поэтому, давайте рассмотрим основные принципы его использования.

Для этого создадим новое приложение, но уже не консольное, а с окном. Разместим в окне текстовое поле, куда будем вводить запрос, кнопку и элемент DataGridView, в котором будем отображать результаты выполнения запросов. Для кнопки создадим обработчик события click. При нажатии на эту кнопку наше приложение должно будет выполнить введенный в текстовое поле запрос select, а результаты выполнения этого запроса приложение будет отображать в DataGridView. Класс DataTable значительно упростит получение и отображение результатов выполненного запроса. Рассмотрение этого класса и есть главной целью нашего приложения. Ради этого мы еще раз воспользуемся присоединенным режимом работы.

> *Мы не будем создавать для этого приложения новую БД, а продолжим использовать созданную в первом уроке. Для этого нам надо будет в новом приложении использовать ту же строку подключения, с которой мы подключались к нашей БД в первом уроке.*

Запустите Visual Studio 2015 и создайте приложение Windows Forms с именем TestDataTable и с таким окном:

Рис. 1. Главное окно приложения

Добавьте в созданное приложение строку подключения из приложения первого урока.

DataTable — типичный табличный элемент. Работа с ним требует соблюдения правил работы с табличными элементами. Прежде всего, вы должны понимать, что после создания объекта DataTable мы получаем заготовку таблицы, изначально пустую. Прежде, чем делать что-либо с любым табличным элементом, надо сформировать колонки желаемой таблицы. Для этого надо указать, сколько в таблице будет колонок, как они будут называться. Возможно, еще надо указать, какого типа данные будут располагаться в колонках, и какой ширины должны быть колонки.

Базовая структура DataTable проста. Есть свойство Columns, элементы которого имеют тип DataColumn и описывают колонки таблицы. Тип DataColumn, в свою

очередь, содержит важные свойства, определяющие колонки таблицы. Это свойства ColumnName, DataType, AllowDBNull и DefaultValue. Для создания в таблице новой колонки, достаточно добавить в Columns строковое имя колонки. До формирования коллекции Columns делать что-либо со строками нельзя.

Еще в классе DataTable есть свойство Rows, элементы которого имеют тип DataRow и представляют собой строки таблицы.

Вспомните, как мы извлекали имена полей таблицы БД при считывании ее данных из элемента SqlDataReader в приложении в первом уроке. Используя этот же прием, мы будем формировать колонки объекта DataTable так, чтобы он соответствовал любой прочитанной таблице. Если наш запрос select прочитал таблицу Books, то объект DataTable будет повторять структуру таблицы Books. Если наш запрос прочитает таблицу Authors, то объект DataTable будет повторять структуру этой таблицы. Другими словами, объект DataTable позволяет нам динамически воссоздавать структуру любой прочитанной таблицы. Давайте рассмотрим примитивный пример. Создадим вручную в элементе DataTable структуру таблицы Authors и добавим несколько строк.

```
DataTable table = new DataTable(); //создана пустая
                                   //таблица
table.Columns.Add(«id»);//создана новая колонка id
table.Columns.Add(«FirstName»);//создана новая
                               //колонка FirstName
table.Columns.Add(«LastName»); //создана новая
                               //колонка LastName
```

```
//теперь можно заносить в таблицу строки
//каждая строка – это объект типа DataRow
DataRow row = table.NewRow();
//метод NewRow() создает строку соответствующую
//таблице, от имени которой он вызван
//в нашем случае row будет массивом из трех элементов
//потому, что мы уже сформировали в table три колонки
//заполним элементы объекта row подходящими данными
//и занесем в таблицу

row[0] = 1;
row[1] = "Francis";
row[2] = "Becon";
table.Rows.Add(row);//в таблицу добавлена новая строка
//другие строки добавляются аналогично
```

Разобрав этот пример создания объекта DataTable, вы готовы понять код, который надо добавить в обработчик кнопки нашего приложения. Отметьте, что объект conn в нашем приложении создан и инициализирован глобально в конструкторе окна, поэтому доступен в приведенном примере кода.

```
private void show_Click(object sender, EventArgs e)
{
    SqlCommand comm = new SqlCommand();
    comm.CommandText = "select * from Authors";
    comm.Connection = conn;
    conn.Open();

    table = new DataTable();
    reader = comm.ExecuteReader();
    int line = 0;

    do
    {
```

```
        while (reader.Read())
        {
            //на первой итерации формируем колонки
            if (line == 0)
            {
                for (int i = 0; i < reader.FieldCount; i++)
                {
                    table.Columns.Add(reader.GetName(i));
                }
                line++;
            }
            //поскольку колонки уже готовы,
            //то на каждой итерации создаем
            //и заполняем очередную строку
            DataRow row = table.NewRow();
            for (int i = 0; i < reader.FieldCount; i++)
            {
                row[i] = reader[i]; //заполняем строку
                                    //из reader
            }
            table.Rows.Add(row);    //добавляем
                                    //очередную строку
        }
    } while (reader.NextResult());
    //да здравствует привязка
    dataGridView1.DataSource = table;

    conn.Close();
    reader.Close();
    }
```

Как вы видите, в этом коде особым образом отмечена строка добавления объекта DataTable в dataGridView1. Здесь используется привязка через свойство DataSource, которая избавляет нас от необходимости подгонять структуру элемента dataGridView1 к структуре нашей таблицы.

Добавьте в созданное приложение приведенный выше обработчик. Создайте в конфигурационном файле строку подключения к БД, созданной в первом уроке. Приведите весь код приложения к такому виду:

```
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.Data.SqlClient;
using System.IO;
using System.Configuration;

namespace AdoNetSample2
{
    public partial class Form1 : Form
    {
        private SqlDataReader reader;
        private DataTable table;
        private SqlConnection conn;

        string cs = "";

        public Form1()
        {
            InitializeComponent();
            conn = new SqlConnection();
            cs = ConfigurationManager.
                 ConnectionStrings["MyConnString"].
                 ConnectionString;
            conn.ConnectionString = cs;
        }

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

```
            SqlCommand comm = new SqlCommand();
            comm.CommandText = tbRequest.Text;
            comm.Connection = conn;
            dataGridView1.DataSource = null;
            conn.Open();

            table = new DataTable();
            reader = comm.ExecuteReader();
            int line = 0;
            do
            {
               while (reader.Read())
               {
                  if (line == 0)
                  {
                     for (int i = 0; i <
                          reader.FieldCount; i++)
                     {
                        table.Columns.Add(reader.
                              GetName(i));
                     }
                     line++;
                  }
               DataRow row = table.NewRow();
               for (int i = 0; i <
                    reader.FieldCount; i++)
               {
                   row[i] = reader[i];
               }
               table.Rows.Add(row);
            }
         } while (reader.NextResult());
         dataGridView1.DataSource = table;
      }
      catch(Exception ex)
      {
          MessageBox.Show("Probably wrong request
                           syntax");
      }
```

```
            finally
            {
                // Close the connection
                if (conn != null)
                {
                    conn.Close();
                }
                if (reader != null)
                {
                    reader.Close();
                }
            }
        }
    }
}
```

Запустите приложение, введите в текстовое поле какой-либо запрос select к нашей БД и нажмите кнопку. Вы должны увидеть в элементе dataGridView1 результат выполнения запроса:

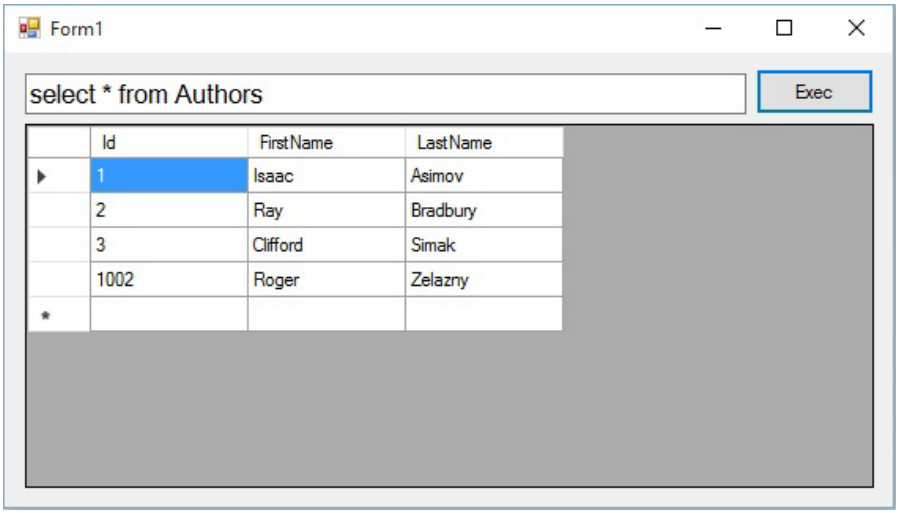

Рис. 2. Выполнение приложения

Обратите внимание, что в первой строке dataGridView1 отображены имена полей таблицы. Это результат работы нашего вложенного цикла for, выполняющегося на первой итерации в цикле while.

Как можно видеть цикл while, в котором происходит извлечение записей из SqlDataReader, заключен в цикл do-while, управляемый методом NextResult(). Это значит, что мы можем заносить в текстовое поле несколько запросов, разделенных «;». Но при этом надо учитывать следующую особенность этого кода. Структура таблицы формируется на первой итерации. Т.е, при обработке результатов первого запроса. Поэтому, если результирующая таблица первого запроса по количеству полей будет превосходить таблицы остальных запросов, все будет выведено нормально. Но если первая таблица будет содержать меньше полей, чем результат следующего запроса, то произойдет ошибка. В нашем случае, в таблице Books больше полей, чем в таблице Authors. Поэтому, если ввести запросы в такой очередности «select * from Books; select * from Authors;» — они выполняться нормально. А если переставить эти два запроса местами — произойдет ошибка. Для обработки этой ситуации в код добавлен try-catch-finally блок. О том, чтобы такая ошибка не возникала, вы позаботитесь в домашнем задании.

Теперь, после знакомства с классом DataTable, перейдем к рассмотрению классов DataSet и DbDataAdapter.

Локальное хранение данных из БД (DataSet)

DataSet представляет собой контейнер, способный хранить в оперативной памяти данные из одной или нескольких таблиц. Этот класс подобен классу DbDataReader, но важное отличие заключается в том, что DataSet используется при отсутствии открытого подключения к БД, в то время, как DbDataReader требует наличия открытого подключения.

В классе DataSet определено свойство Tables, представляющее собой коллекцию, элементами которой являются объекты типа DataTable.

Поскольку этот класс способен хранить данные нескольких таблиц, а таблицы могут быть связанными, то необходимо где-то хранить и информацию о связях. Для хранения информации о связях используется свойство Relations. Relations — это тоже коллекция, элементами которой являются объекты типа Relation.

Этот класс позволяет приложению загрузить из БД набор необходимых данных. Даже данных из нескольких разных таблиц. Эти данные конечно же являются данными, возвращаемыми какими-либо запросам select. После загрузки данных в объект DataSet, приложение может отключиться от сервера и продолжить работу с этими данными локально. Затем, при необходимости, приложение может снова подключиться к серверу и вернуть обратно в БД данные, которые были изменены локально.

Как правило, создается объект DataSet конструктором без параметров. А загрузка данных в этот объект

происходит автоматически при выполнении специального метода, с которым мы познакомимся в следующем разделе.

Давайте проведем мысленный эксперимент. Абстрагируемся от того, каким образом в объект DataSet попали данные. Будем считать, что в нашем объекте находятся результаты выполнения запросов select * from Authors и select * from Books. Как можно работать с такими данными?

Поскольку запросов два, то в коллекции Tables будет два элемента, две таблицы. К ним можно обращаться либо по индексам, либо по именам. Именами по умолчанию будут «table», «table1» и т.д. Но эти умолчания можно изменять. Посмотрите на приведенные примеры кода, чтобы получить представление о том, как можно обращаться к данным в DataSet.

```
//создаем объект ds
DataSet ds = new DataSet();
//каким-либо способом заполняем объект ds
//результатами наших двух запросов

//извлекаем из ds первую таблицу по индексу
DataTable dt1 = ds.Tables[0];
//извлекаем из ds вторую таблицу по индексу
DataTable dt2 = ds.Tables[1];

//извлекаем из ds первую таблицу по имени
DataTable dt3 = ds.Tables[«table»];
//извлекаем из ds вторую таблицу по имени
DataTable dt4 = ds.Tables[«table1»];

//извлекаем LastName автора из первой строки таблицы
//dt1 по индексам
string lastName1 = dt1.Rows[0][1].ToString();
```

```
//извлекаем LastName автора прямо из объекта DataSet
string lastName2 = (string)ds.Tables[0].Row[0].
["LastName"];
```

Подключение к БД (DbDataAdapter)

Главным действующим лицом в отсоединенном режиме работы с сервером является класс DbDataAdapter. Этот класс отвечает за открытие подключения к БД, получение данных из БД или загрузку данных в БД и за закрытие подключения. Если сформулировать действия этого класса точнее, то можно сказать, что этот класс управляет переносом данных из БД в объект DataSet и из объекта DataSet обратно в БД. Класс DbDataAdapter работает в паре с классом DataSet. Последний является контейнером первого.

Справедливости ради надо сказать, что где-то внутри класс DbDataAdapter использует классы DbCommand и DbDataReader. Но все это делается автоматически и, если бы я вам не сказал об этом, у вас не было бы возможности об этом догадаться.

У класса DbDataAdapter есть четыре основных свойства типа DbCommand:

- SelectCommand (для чтения данных из БД),
- InsertCommand (для добавления данных в БД),
- UpdateCommand (для изменения данных в БД),
- DeleteCommand (для удаления данных из БД)

В каждом из этих свойств должен храниться соответствующий запрос, который будет выполняться при подключении к БД. Не обязательно инициализировать все эти четыре свойства. Но свойство SelectCommand

должно быть инициализировано всегда. О разных способах инициализации этих свойств поговорим немного ниже на этой странице.

Наряду с этими четырьмя свойствами класса важную роль играют два метода Fill() и Update().

> *Не ассоциируйте имя метода Update() с SQL запросом update. Этот метод изменяет состояние БД, выполняя для этого SQL запросы, занесенные в свойства InsertCommand, UpdateCommand и DeleteCommand.*

Когда вызывается метод Fill() он выполняет запрос, занесенный в свойство SelectCommand. Данные, сформированные в результате выполнения этого запроса, заносятся в объект DataSet (или DataTable). Обратите внимание, что при использовании класса DbDataAdapter нам не надо выполнять открытие подключения и закрытие подключения. Если подключение не открыто, DbDataAdapter откроет его сам и сразу же сам закроет, когда завершит действие с БД. Если подключение на момент вызова Fill() или Update() будет открыто, оно так и останется открытым после завершения использования DbDataAdapter. Схематично работу с DbDataAdapter можно описать так:

```
//для создания объекта DbDataAdapter надо иметь
//запрос select и объект DbConnection
SqlConnection conn = new SqlConnection(@"Data Source=
                (localdb)\v11.0;InitialCatalog=
                Library;Integrated Security=SSPI");
                String selectSQL =
                "SELECT * FROM Authors";
```

```
//создаем объект DbDataAdapter
SqlDataAdapter da = new SqlDataAdapter(selectSQL, conn);

//объяснение этой строки почитайте ниже
SqlCommandBuilder cmdBldr = new SqlCommandBuilder(da);

//создаем объект DataSet для локального хранения
//данных из БД
DataSet ds = new DataSet();

//вызов метода Fill() выполняет запрос select из
//свойства SelectCommand и заносит прочитанные
//данные в объект DataSet
da.Fill(ds);
```

Отдельно надо рассказать о строке кода, выделенной желтым цветом. Строго говоря, заполненными должны быть все четыре командных свойства объекта DbDataAdapter: SelectCommand, InsertCommand, UpdateCommand и DeleteCommand. Запросы в каждое из этих свойств мы будем заносить разными способами. Запрос в SelectCommand работает как входной запрос для приложения, он переносит данные из БД в приложение. Запросы в остальных трех командных свойствах работают, как выходные, синхронизируя данные из приложения с БД.

Если нам не надо явно вмешиваться в процесс синхронизации данных, можно поступать таким образом. Явно инициализировать свойство SelectCommand необходимым запросом. А для остальных трех командных свойств создать специальные запросы по умолчанию. Для создания этих запросов по умолчанию используется

класс SqlCommandBuilder. Применение этого класса очень простое — надо только создать его объект, передав в конструктор объект адаптера, для которого надо создать такие командные запросы.

Как явно управлять процессом синхронизации мы рассмотрим после знакомства со следующим разделом этого урока.

Пример использования DbDataAdapter

Давайте создадим еще одно приложение, в котором на практике рассмотрим использование объекта DbDataAdapter. Как и в предыдущем случае, это будет приложение Windows Forms с именем TestDataAdapter. Таким же образом расположим в окне этого приложения текстовое поле, куда будем вводить запросы

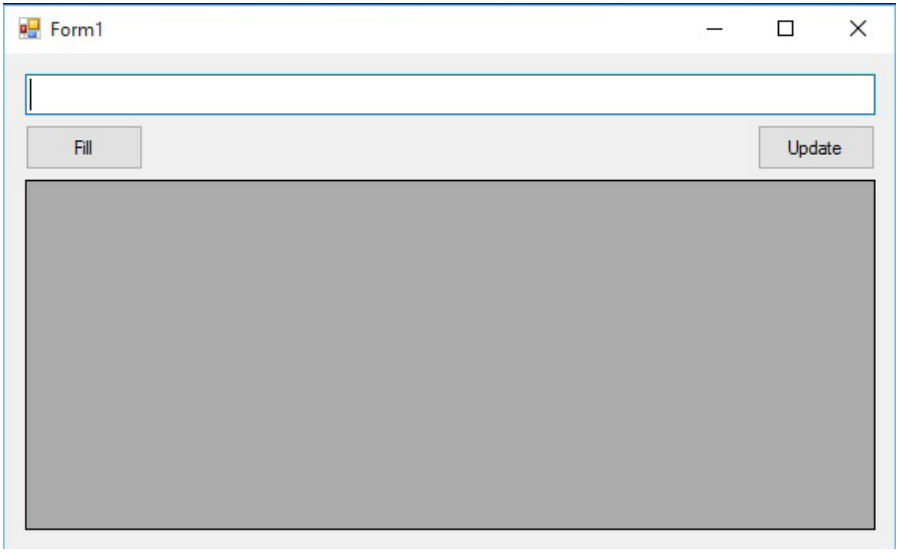

Рис. 3. Главное окно приложения

и элемент DataGridView, в котором будем отображать результаты выполнения запросов. Кнопок нам понадобится две. Подпишем их Fill и Update. Для обеих кнопок создадим обработчики события click. Все как в предыдущем случае. Но сейчас мы будем работать с классом DbDataAdapter.

Занесите в файл конфигурации приложения строку подключения к нашей БД Library. Как и в предыдущем приложении, добавьте ссылку на пространство имен System.Configuration. Запустите наше приложение и убедитесь, что главное окно выглядит таким образом, как на рисунке.

Приведите код приложения в соответствие с образцом, показанным ниже.

```
using System.Data.SqlClient;
using System.Configuration;

namespace AdoNetSample3
{
    public partial class Form1 : Form
    {
        private SqlConnection conn=null;
        SqlDataAdapter da = null;
        DataSet set = null;
        SqlCommandBuilder cmd = null;
        string cs = "";

        public Form1()
        {
            InitializeComponent();
            conn = new SqlConnection();
```

```
            cs = ConfigurationManager.
                 ConnectionStrings["MyConnString"].
                 ConnectionString;
            conn.ConnectionString = cs;
        }

        private void show_Click(object sender,
                                EventArgs e)
        {
            try {
                SqlConnection conn =
                              new SqlConnection(cs);
                set = new DataSet();
                string sql = tbRequest.Text;
                da = new SqlDataAdapter(sql, conn);
                dataGridView1.DataSource = null;
                cmd = new SqlCommandBuilder(da);
                da.Fill(set, "mybook");
                dataGridView1.DataSource =
                                set.Tables["mybook"];
            }
            catch(Exception ex)
            {
            }
            finally
            {
            }
        }
        private void button1_Click(object sender,
                EventArgs e)
        {
            da.Update(set, "mybook");
        }
    }
}
```

Давайте посмотрим, что умеет делать класс DbDataAdapter. Запустите приложение, введите в текстовое

поле запрос select * from Authors, и нажмите кнопку Fill. При нажатии на эту кнопку будут созданы необходимые объекты: SqlConnection, DataSet и SqlDataAdapter. Затем будет выполнен метод Fill(), который занесет результат выполнения указанного запроса select в объект DataSet. Результат выполнения этого запроса разместится в свойстве Tables, сформировав там первую таблицу с именем «mybook». Имя таблицы можно задать в необязательном втором параметре метода Fill(). Если бы мы выполнили несколько запросов select, то результат второго оказался бы в таблице с именем «mybook1», третьего — в таблице «mybook2» и т.д.

Задавать имена в методе Fill() имеет смысл, если вы не хотите иметь дело с индексами и запоминать какая таблица располагается в Tables[0], а какая — в Tables[1]. Если бы мы не указали имя для таблицы, то две строки, выделенные желтой заливкой, выглядели бы так:

```
da.Fill(set);
dataGridView1.DataSource = set.Tables[0];
```

Кроме того, даже если бы мы не указали имя таблицам в методе Fill(), то все-равно, могли бы обращаться к таблицам в DataSet по именам «table», «table1» и т.д. Эти имена присваиваются таблицам в DataSet по умолчанию. После нажатия на кнопку Fill вы увидите заполненный dataGridView1. Хотя внешне это очень похоже на работу с нашим предыдущим приложением, разница здесь принципиальная.

Существует перегруженная версия метода Fill(), позволяющая извлечь из БД часть строк, воз-

вращенных запросом select. Такой вызов da.Fill (ds, 0, 10, «mybook») занесет в DataSet первые 10 строк и разместит их в коллекции Tables с индексом «mybook». Однако имейте в виду, что эта перегрузка метода все равно излекает из БД все строки запроса, но ненужные затем отбрасывает. Поэтому говорить об оптимальности здесь не стоит.

Сейчас, когда вы наблюдаете данные из таблицы Authors в окне своего приложение, вы уже отключены от БД. Все прочитанные из БД данные расположены сейчас в объекте DataSet и доступны вам локально. Вы можете делать с этими данными все, что хотите, а затем можете вернуть их в БД, изменив таким образом содержимое своей БД.

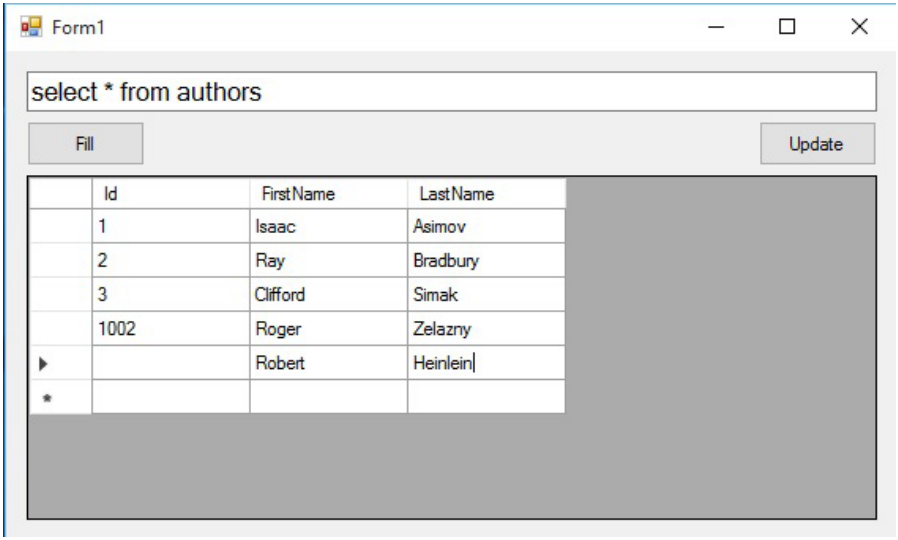

Рис. 4. Добавление нового писателя

Перейдите в последнюю строку элемента dataGridView1 и занесите имя и фамилию нового писателя. В поле для идентификатора заносить ничего не надо. Значение этого поля формируется автоматически. После добавления в dataGridView1 имени и фамилии нового писателя, окно вашего приложения может выглядеть рис. 4.

Теперь нажмите кнопку Update. Хотите верьте, хотите нет, но в вашу БД, в таблицу Authors, только что добавилась новая строка. Впрочем, верить я вас не призываю. Призываю проверить мое утверждение. Снова нажмите кнопку Fill, чтобы повторно прочитать данные из таблицы Authors. Вы увидите в таблице новую строку уже с новым значением присвоенного этой строке идентификатора.

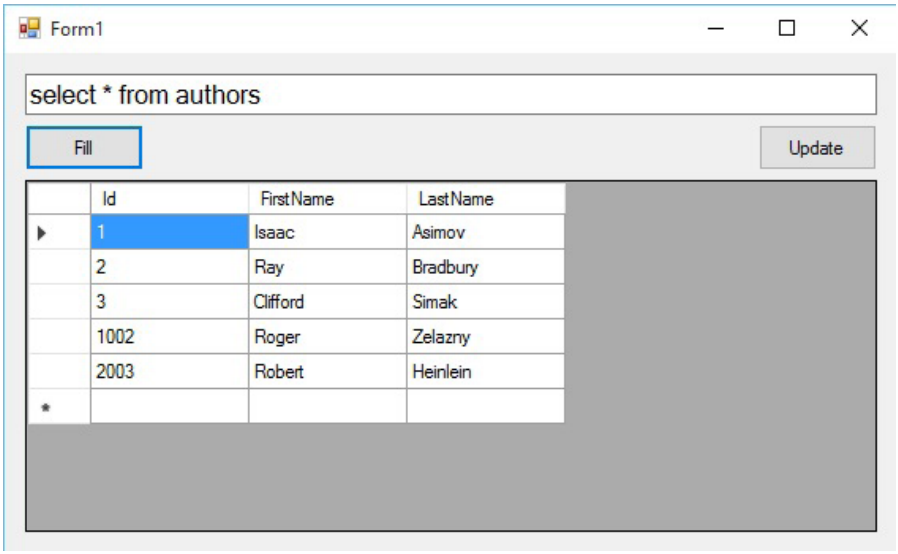

Рис. 5. Чтение измененных данных

Итак, работая с локальными данными (в то время, как приложение было отключено от БД), вы добавили

в DataSet новую запись. Затем вызвали метод Update() объекта адаптера. Этот метод подключился к БД и синхронизировал свои локальные данные с той таблицей, из которой эти данные были ранее прочитаны! Выполнив синхронизацию, метод Update() отключился от БД.

Точно также вы могли изменить любое значение в любом поле или даже удалить какую-либо строку. Все изменения были бы синхронизированы с БД при выполнении метода Update().

Если вам интересно знать, как это происходит, потерпите немного. Мы скоро все это разберем. Сейчас важно проделать следующий эксперимент.

Запустите наше приложение снова и выполните такой запрос: select title, price, pages from books. Станьте в любую строку dataGridView1 и измените значение любого поля, например цену. После занесения новой цены, нажмите кнопку Update. В этот раз что-то пошло не так. Если перевести объяснение возникшей исключительной ситуации, то вы увидите сообщение о том, что динамическая синхронизация БД не поддерживается для запроса select, не возвращающего никаких сведений о первичном ключе таблицы. Если вдуматься, то это логично. Мы исключили из запроса select вывод поля id и при попытке синхронизировать данные в DataSet с данными в таблице БД, объект SqlDataAdapter просто не понимает, откуда была прочитана каждая строка, расположенная сейчас в DataSet и куда эту строку возвращать в БД. Давайте запомним этот результат.

Чтобы метод Update() мог автоматически синхронизировать локальные данные в DataSet с данными в таблицах БД, локальные данные должны быть занесены в DataSet запросом, возвращающим первичный ключ или другой уникальный ключ таблицы.

Применение SqlCommandBuilder

Последний пример позволяет нам сформулировать следующий принцип использования класса SqlDataAdapter. При вызове метода Fill() выполняется запрос, хранящийся в свойстве SelectCommand. Выполнение этого запроса приводит к заполнению объекта DataSet. После этого происходит отключение от сервера и наступает период работы с локальными данными. Затем в какой-то момент вызывается метод Update(). Это приводит к выполнению запросов, хранящихся в свойствах InsertCommand, UpdateCommand и DeleteCommand. Выполнение этих запросов приводит к изменению содержимого БД.

Вот на этом надо остановиться подробнее. Запрос для свойства SelectCommand мы создаем сами. А откуда появляются запросы в свойствах InsertCommand, UpdateCommand и DeleteCommand? Они заносятся в SqlDataAdapter классом SqlCommandBuilder при создании объекта этого класса. Давайте посмотрим на эти запросы. Сделать это не сложно. Добавьте в код нашего приложения, после создания объекта SqlCommandBuilder такие строки:

```
Debug.WriteLine(cmd.GetInsertCommand().CommandText);
Debug.WriteLine(cmd.GetUpdateCommand().CommandText);
Debug.WriteLine(cmd.GetDeleteCommand().CommandText);
```

и вы увидите запросы, созданные классом SqlCommandBuilder для этих трех командных свойств. У меня эти запросы выглядят так:

```
INSERT INTO [books] ([AuthorId], [Title], [PRICE],
[PAGES]) VALUES (@p1, @p2, @p3, @p4)

UPDATE [books] SET [AuthorId] = @p1, [Title] =
@p2, [PRICE] = @p3, [PAGES] = @p4 WHERE (([Id] =
@p5) AND ([AuthorId] = @p6) AND ([Title] =
@p7) AND ((@p8 = 1 AND [PRICE] IS NULL) OR ([PRICE] =
@p9)) AND ((@p10 = 1 AND [PAGES] IS NULL) OR ([PAGES]
= @p11)))

DELETE FROM [books] WHERE (([Id] = @p1) AND
([AuthorId] = @p2) AND ([Title] = @p3) AND
((@p4 = 1 AND [PRICE] IS NULL) OR ([PRICE] = @p5))
AND ((@p6 = 1 AND [PAGES] IS NULL) OR ([PAGES] =
@p7)))
```

Вы видите, что это параметризированные запросы, настроенные на изменение БД при точном соответствии параметров. Мы пока не знаем, откуда берутся параметры для выполнения этих запросов. Если вам кажется, что эти запросы не очень оптимизированы, то вы правы. Представьте себе эффективность этих запросов для таблиц с десятками полей, большинство из которых, к тому же, не индексированы. Поэтому, отметьте для себя, что если быстродействие критично для вашего приложения, то лучше эти запросы подготовить самостоятельно, а не поручать их создание классу SqlCommandBuilder. Как всегда в жизни, если надо сделать что-то важное — сделай это сам, а не поручай другим.

Запросы, создаваемые классом SqlCommandBuilder, зависят от запроса select, хранящегося в свойстве SelectCommand. Поэтому если вы изменили запрос select в свойстве SelectCommand, вы должны перестроить запросы в трех остальных командных свойствах. Чтобы сделать это, надо просто вызвать метод RefreshSchema( ) объекта SqlCommandBuilder.

Запомните главные ограничения класса SqlCommandBuilder:

- *созданные им запросы синхронизации не оптимизированы;*
- *этот класс позволяет создавать запросы модификации только для данных, полученных с помощью однотабличного запроса select, т.е. если в вашем свойстве SelectCommand занесен многотабличный запрос, вы не должны полагаться на SqlCommandBuilder;*
- *этот класс не поддерживает хранимые процедуры;*

Пользовательская логика в синхронизации БД

Рассмотрим, как можно заносить в свойства InsertCommand, UpdateCommand и DeleteCommand свои собственные запросы. Тип каждого из этих свойств DbCommand. Работать с этим типом мы уже умеем. Нам надо создать такой объект, инициализировать его свойство CommandText и занести этот объект в соответствующее командное свойство SqlDataAdapter.

Рассмотрим пример создания собственного запроса для свойства UpdateCommand. Создание запросов для

других свойств происходит аналогично. Очень часто нам не надо предоставлять пользователю возможность изменять любые значения в таблице. Например, для книги логично разрешить изменять только цену. Поэтому наш пользовательский запрос будет специализированным, позволяющим изменять только цену.

```
SqlCommand UpdateCmd = new SqlCommand("Update Books
          set Price = @pPrice where id = @pId", conn);

//создаем параметры для запроса Update
UpdateCmd.Parameters.Add(new SqlParameter("@pPrice",
          SqlDbType.Int));
UpdateCmd.Parameters["@pPrice"].SourceVersion =
          DataRowVersion.Current;
UpdateCmd.Parameters["@pPrice"].SourceColumn =
          "Price";

UpdateCmd.Parameters.Add(new SqlParameter("@pId",
          SqlDbType.Int));
UpdateCmd.Parameters["@pId"].SourceVersion =
          DataRowVersion.Original;
UpdateCmd.Parameters["@pId"].SourceColumn = "id";

//вставляем созданный объект SqlCommand в свойство
//UpdateCommand SqlDataAdapter
da.UpdateCommand = UpdateCmd;
```

Мы с вами уже рассматривали тип SqlParameter, но тогда не говорили о его свойствах SourceVersion и SourceColumn. Эти свойства определяют какое значение надо заносить в параметр. Для запроса update существует две версии изменяемой строки таблицы:

- Исходная строка, находящаяся в таблице сейчас;

- Обновленная строка, с изменениями, которые должны быть занесены в таблицу;

Чтобы можно было различать такие строки, применяется свойство SourceVersion. Его значения определены в перечислении DataRowVersion. Первой строке соответствует значение DataRowVersion.Original, второй — DataRowVersion.Current. В нашем примере мы указываем, что значение цены в параметр заносим из «новой» измененной строки (DataRowVersion.Current), а для сравнения строк выбираем id из исходной строки (DataRowVersion.Original). По умолчанию значение этого свойства DataRowVersion.Current.

Свойство SourceColumn указывает, из какого поля выбирается значение.

Для инициализации командных свойств адаптера можно использовать не только SQL запросы, а также и хранимые процедуры. Хранимые процедуры в большинстве случаев являются предпочтительным вариантом. Посмотрим, как можно использовать хранимые процедуры в командных свойствах объекта SqlDataAdapter. Предположим, что в нашей БД есть такая хранимая процедура:

```
CREATE PROCEDURE UpdateBooks
    @pId int,
    @pAuthorId int,
    @pTitle nchar(100),
    @pPrice int,
    @pPages int

AS
SET NOCOUNT ON
```

```
Update Books
set
AuthorId = @pAuthorId,
Title = @pTitle,
Price = @pPrice,
Pages = @pPages
Where  ip = @pId

return
```

Рассмотрим, как этой хранимой процедурой инициализировать свойство UpdateCommand и как подготовить параметры.

```
//создадим объект SqlCommand и инициализируем
//его хранимой процедурой
SqlCommand updateCommand = new
SqlCommand("UpdateBooks", conn);
updateCommand.CommandType = CommandType.
                            StoredProcedure;

//создадим параметры для хранимой процедуры
//для этого создадим ссылку типа коллекции
//Parameters и свяжем ее со свойством Parameters
//созданного объекта updateCommand
//через такую ссылку будет удобно добавлять параметры

SqlParameterCollection cparams;

cparams = updateCommand.Parameters;

//добавляем параметры для хранимой процедуры
cparams.Add("@pid", SqlDbType.Int, 0, "id");
cparams["@pid"].SourceVersion =
                            DataRowVersion.Original;
```

```
cparams.Add("@pAuthorId", SqlDbType.Int, 8, "AuthorId");
cparams.Add("@pTitle", SqlDbType.NChar, 100, "Title");
cparams.Add("@pPrice", SqlDbType.Int, 8, "Price");
cparams.Add("@pPages", SqlDbType.Int, 8, "Pages");

//инициализируем командное свойство UpdateCommand
//адаптера
da.UpdateCommand = updateCommand;
```

Таким же способом можно создать хранимые процедуры для остальных командных свойств адаптера. Теперь, при вызове метода Update() будут выполняться не SQL запросы, а хранимые процедуры, что является более оптимальным с точки зрения быстродействия и безопасности.

Очередность выполнения изменений в БД (DataViewRowState)

Мы уже подробно рассмотрели выполнение метода Update(). Однако есть еще несколько особенностей, которые надо рассмотреть. Предположим, ваше приложение выполнило метод Fill(), прочитало данные из БД, отключилось от сервера и занялось локальной обработкой прочитанных данных. В ходе такой обработки какие-то строки были изменены, какие-то добавлены и удалены. Как вы думаете, в какой очередности будут выполняться изменения в БД при вызове метода Update()? И имеет ли значение, в какой очередности эти изменения выполнять?

Если вы подумаете над этими вопросами, то должны будете придти к выводу, что очередность конечно же важна. Более того, у нас должен быть инструмент, позволяющий управлять очередностью выполняемых изменений в БД.

Давай познакомимся со способом управления изменениями в БД.

При вызове метода Update() строки обрабатываются по одной. У каждой строки есть свойство DataViewRowState, допустимые значения которого определены в перечислении с таким же именем. Значение этого свойства зависит от действий, выполненных со строкой. Если какую-то строку удалили, то значение этого свойства у строки будет равно DataViewRowState.Deleted, для добавленной строки это свойство будет равно DataViewRowState.Added, для измененной — DataViewRowState.ModifiedCurrent, для строки, которая не была изменена, это свойство будет равно DataViewRowState.Unchanged. Используя это свойство мы напишем код, который позволит нам выполнять синхронизацию локальных данных из нашего приложения с БД в той очередности, которая нам нужна.

В этом нам поможет метод Select, определенный в классе DataTable и позволяющий выбирать из таблицы строки с заданным значением свойства DataViewRowState. Сначала выберем из объекта DataSet данные таблицы, которую мы хотим синхронизировать с нашей БД. Затем, вместо однократного вызова метода Update(), вызовем этот метод несколько раз, передавая ему для обработки только строки с указанным типом изменения, только удаленные, или только добавленные. Отобрать эти строки мы можем с помощью метода Select(), который возвращает массив типа DataRow. Таким образом, мы выполним синхронизацию в той очередности, которая требуется в нашем приложении.

```
DataTable table = set.Tables["mybook"];

//сначала обработаем удаленные строки
da.Update(table.Select(null, null,
                        DataViewRowState.Deleted));

//теперь обработаем измененные строки
da.Update(table.Select(null, null, DataViewRowState.
                       ModifiedCurrent));

//теперь обработаем добавленные строки
da.Update(table.Select(null, null, DataViewRowState.
                        Added));
```

Отметьте для себя, что сразу после вызова метода Fill(), все строки, занесенные в DataSet, имеют значение свойства RowState равное DataViewRowState.Unchanged. За это отвечает булевское свойство AcceptChangesDuringFill объекта SqlDataAdapter. По умолчанию значение этого свойства равно True и все прочитанные из БД строки получают RowState равное DataViewRowState.Unchanged. Если выставить свойство AcceptChangesDuringFill равным False, то сразу после вызова метода Fill() будут иметь свойство RowState равное DataViewRowState.Inseretd. Если вам надо применить все проделанные изменения, вы можете вызвать метод AcceptChanges() объекта DataSet, который переведет свойство RowState к значению DataViewRowState.Unchanged. Следующий пример кода демонстрирует изменения состояния строк:

```
DataSet set = new DataSet();
da.Fill(ds, "mybooks");
//по умочлчанию после Fill() каждая строка
//в «mybooks» имеет RowState = Unchanged

//удалим все из таблицы «mybooks»
ds.Tables["mybooks"].Clear();

//изменим свойство адаптера
//AcceptChangesDuringFill = false
da.AcceptChangesDuringFill = false;

//снова вызовем Fill(), чтобы заполнить таблицу «mybooks»
da.Fill(ds, "Orders");
//сейчас после Fill() каждая строка в «mybooks» имеет
//RowState = Inserted

//вызов AcceptChanges изменяет значение свойства
//RowState для всех строк
ds.AcceptChanges();
//теперь каждая строка в «mybooks» имеет
//RowState = Unchanged
```

Отображение таблиц (TableMappings)

Рассмотрим ситуацию, когда мы инициализировали командное свойство SelectCommand адаптера такой строкой:

```
string strSQL = "SELECT * FROM Authors;" +
                "SELECT * FROM Books;" +
                "SELECT * FROM Publishers";
```

В нашей БД нет таблицы Publisher. Но, давайте представим, что такая таблица есть. Нам для примера полезнее, чтобы таблиц было больше двух. Если хотите — тогда добавьте в БД такую таблицу, но можно этого не делать

и ограничиться мысленным экспериментом. Дальше выполним стандартный вызов метода Fill():

```
da = new SqlDataAdapter(strSQL, conn);
ds = new DataSet();
da.Fill(ds);
```

После вызова метода Fill() в свойстве Tables нашего DataSet будут находиться три таблицы. Каждая из таблиц будет содержать результат выполнения одного из запросов select. Вспомните, какие будут имена у этих таблиц в свойстве Tables. По умолчанию, первая таблица будет называться «Table», вторая — «Table1», третья — «Table2». Если вы захотите, например, отобразить в DataGridView данные из таблицы Books, вам надо будет написать такой код:

```
dataGridView1.DataSource = set.Tables[1];
```

или такой:

```
dataGridView1.DataSource = set.Tables[“Table1”];
```

Другими словами, вам надо помнить, что запрос для таблицы Books вы занесли в адаптер вторым или же надо будет помнить, что имя таблицы с результатами этого запроса — «Table1». А если таблиц много? А если воображение слабое? Хотели бы вы, чтобы данные из таблицы БД Authors располагались в DataSet в таблице «Authors», данные из таблицы Books — в DataSet в таблице «Books» и т.д.? Тогда смотрите, как это можно сделать.

У адаптера есть свойство public DataTableMappingCollection TableMappings { get; }

Тип этого свойства DataTableMappingCollection определен в пространстве имен System.Data.Common. Это коллекция, элементами которой являются объекты типа DataTableMapping. Свойство TableMappings позволяет нам задать соответствие имен таблиц в БД с именами таблиц в DataSet. Скоро вы увидите, что это свойство позволяет задавать не только соответствие между именами таблиц, а и еще кое-что. Но сначала рассмотрим отображение имен таблиц.

Наша цель заключается в том, чтобы для таблицы БД Authors в DataSet была создана таблица с таким же именем Authors, и точно также для других таблиц. Для этого мы можем поступить таким образом:

```
da.TableMappings.Add("Table", "Authors");
da.TableMappings.Add("Table1", "Books");
da.TableMappings.Add("Table2", "Publishers");
da.Fill(ds);
```

Теперь после выполнения метода Fill() результаты каждого из трех наших запросов будут располагаться в DataSet в таблицах с именами, совпадающими с именами исходных таблиц в БД. Эти же действия можно записать в более развернутом виде:

```
DataTableMapping dtm1 = new DataTableMapping();
dtm1.SourceTable = "Table";
dtm1.DataSetTable = "Authors";
da.TableMappings.Add(dtm1);
```

```
DataTableMapping dtm2 = new DataTableMapping();
dtm2.SourceTable = "Table1";
dtm2.DataSetTable = "Books";
da.TableMappings.Add(dtm2);

DataTableMapping dtm3 = new DataTableMapping();
dtm3.SourceTable = "Table2";
dtm3.DataSetTable = "Publishers";
da.TableMappings.Add(dtm3);
da.Fill(ds);
```

Здесь закономерно возникает вопрос, можем ли мы выполнять отображения не только для имен таблиц, а также для имен столбцов таблиц? Да. Для этого существует свойство объекта DataTableMapping, которое называется ColunmMappings:

```
public DataColumnMappingCollection ColumnMappings { get; }
```

По умолчанию каждому столбцу возвращаемому запросом select соответствует столбец с таким же именем в таблице, создаваемой в DataSet. Вы можете изменить такое поведение двумя способами. Например, если укажете в запросе select псевдонимы для полей:

```
"SELECT id as Num, FirstName as Name, LastName as
Surname FROM Authors;"
```

то созданные в DataSet столбцы будут именованы согласно псевдонимам.

Однако можно использовать более гибкий механизм, предоставляемый свойством ColumnMappings. Рассмотрим пример для одного запроса select, в котором мы хотим заменить и имя таблицы и имена столбцов:

```
string strSQL = "SELECT id, FirstName,
                 LastName FROM Authors";
SqlDataAdapter da = new SqlDataAdapter(strSQL, conn);
//Отображение имен таблиц
DataTableMapping dtm = da.TableMappings.Add("Table",
                "Authors");
//отображение имен столбцов
dtm.ColumnMappings.Add("id", "Num");
dtm.ColumnMappings.Add("FirstName", "Name");
dtm.ColumnMappings.Add(«LastName», «Surname»);
da.Fill(ds);
```

Вы прекрсано понимаете, что поскольку оба свойства TableMappings и ColumnMappings являются коллекциями, то в них можно не только добавлять новые элементы, а и удалять существующие. Для этого есть методы Remove() и RemoveAt().

Теперь вы можете более комфортно работать в приложении с данными, прочитанными из БД.

Работа с графической информацией

Все созданные нами до сих пор примеры приложений являются просто иллюстрациями, демонстрирующими разные аспекты использования классов ADO.NET. Давайте сейчас создадим приложение, в котором будет содержаться определенная клиентская логика. Продолжим работу с нашей БД Library, но внесем в нее ряд изменений. Добавим таблицу в которой будем хранить картинки для наших книг. Чтобы для любой книги в таблице Books можно было хранить произвольное количество картинок, создадим новую таблицу.

```
CREATE TABLE Pictures
(
    Id INT NOT NULL IDENTITY(1,1) PRIMARY KEY,
    BookId INT NOT NULL,
    FOREIGN KEY (BookId) REFERENCES BOOKS (Id),
    Name VARCHAR(100) NOT NULL,
    Picture VARBINARY(MAX)
)
```

В этой таблице будем хранить картинки для разных книг. С помощью связи с таблицей Books будем привязывать картинку к конкретной книге. Как вы видите по типу поля Picture, я предлагаю хранить в таблице не путь к картинке, а именно бинарное содержимое.

Здесь надо сделать небольшое отступление. Мы часто сталкиваемся с таким суеверием: в таблицах БД нельзя

хранить бинарное содержимое, а надо хранить только пути к картинкам. Давайте обсудим это утверждение. Чем оно вызвано? Обычно, сторонники этого суеверия говорят, что таблицы в этом случае становятся «очень большими» и обрабатываются «очень медленно». Да, такой факт имеет место. Особенно, если заносить в каждую строку таблицы бинарное содержимое размером в 1.5–2.0 Гигабайта. Ведь varbinary(MAX) позволяет хранить такие объемы.

Но если вести себя скромно и ограничиваться картинками в несколько Мегабайт, то ничего страшного не произойдет. А очень часто бывает достаточно иметь дело с еще меньшими картинками.

Давайте объективно оценим плюсы и минусы каждого из подходов. Хранение бинарных данных в файловой системе, а не в таблицах имеет смысл в таких случаях, когда у вас ограничен размер дискового пространства для базы данных, например, дорогостоящий хостинг. Или же, когда бинарные данные, с которыми работает ваше приложение, должны быть доступны еще и другим приложениям;

Хранение бинарных данных в таблицах решает вопросы целостности данных. Такие данные все время синхронизированы. Бинарные объекты в таблицах архивируются при бэкапе и не надо предпринимать никаких дополнительных действий для их переноса при переносе БД.

Если эти аргументы вас не убедили, и кто-то еще боится типа varbinary, вот вам еще один аргумент в пользу

использования этого типа. Вы, конечно же, помните, что любая БД просто обязана обеспечивать целостность своих данных. Это одна из важнейших функций БД. Для этого созданы ограничения, триггеры и много чего другого. Теперь представим, что вы храните свои картинки в папке на диске, а в таблице храните пути к этим картинкам. Если картинки в вашей папке будут изменены, удалены, переименованы, то каким образом БД не то, что помешает этому, а как она просто сможет узнать об этом? Никак! И где здесь целостность? Зачем мне быстрая таблица, в которой недостоверные данные? Вы как хотите, а varbinary — это круто!

Но, чтобы никто сильно не переживал, давайте запретим пользователю заносить в нашу БД очень большие картинки. Сделаем так, что выбрать пользователь сможет картинку любого размера, а наше приложение создаст для выбранной картинки уменьшенную копию, не превышающую по ширине и высоте, например, 300 пикселей, и занесет в таблицу эту уменьшенную копию. Надеюсь, все понимают, что значение 300 взято «с потолка»?

Итак. Добавляем в нашу БД новую таблицу. Это действие мы уже выполняли. Повторим еще раз. В окне SQL Server Object Explorer раскрываем узел Tables нашей БД и выбираем из контекстного меню опцию Add New Table... Вставляем в окно N-SQL (располагается в нижней части рабочего окна) запрос для создания нашей новой таблицы Pictures.

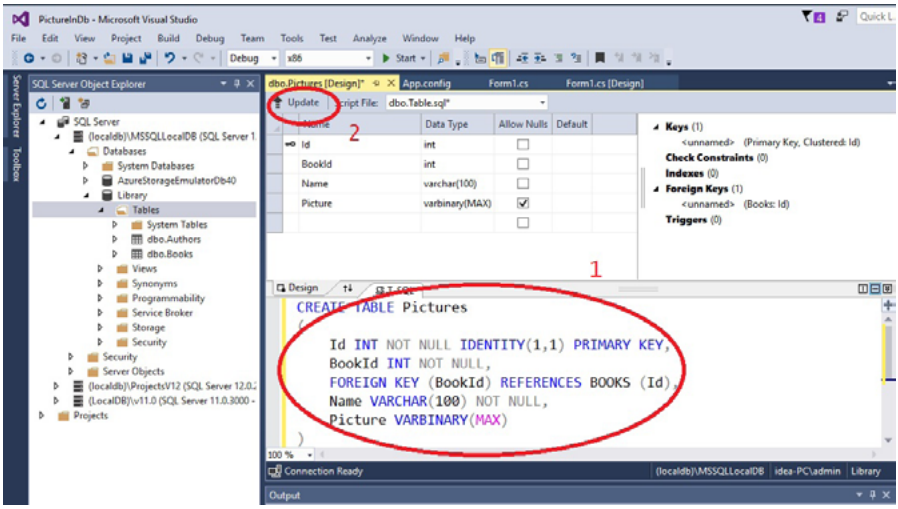

Рис. 6. Добавление новой таблицы

Нажимаем на верхней панели кнопку Update и в окне, которое появляется после этого, жмем кнопку Udpate Database. Новая таблица создана и связана с таблицей Books.

Теперь запускаем VisualStudio 2015 и создаем новое Windows Forms приложение. Мы будем продолжать использовать измененную БД Library. Хотя БД изменена, способ подключения к ней остался таим же, как в предыдущих проектах. Поэтому в конфигурационный файл созданного приложения скопируйте содержимое из конфигурационного файла любого из наших предыдущих проектов. Окно нового созданного приложения может выглядеть так.

Рис. 7. Главное окно приложения

Не обязательно создавать панель инструментов, и делать окно именно таким, как на рисунке. Но надо обеспечить выполнение таких действий:

- загрузка в БД новой картинки, привязанной к какой-либо книге — за это отвечает кнопка Load Picture и текстовое поле на панели инструментов, в котором надо указать id книги;
- вывод из БД информации о конкретной записи из таблицы Pictures — за это отвечает кнопка Show One, при этом в текстовом поле на панели инструментов надо указать id записи из таблицы Pictures;
- вывод из БД информации обо всех записях из таблицы Pictures — за это отвечает кнопка Show All;

- отображение результатов выполнения запросов select;
- отображение графической информации из поля varbinary(MAX);

В нижней левой части окна в элементе DataGridView отображаются результаты выполнения запросов, а в правой части окна в элементе PictureBox — отображается картинка при выполнении действия Show One.

Приведите код приложения к указанному виду. Код достаточно комментирован и понятен. Для реального применения надо еще добавить более серьезную валидацию и улучшить пользовательский интерфейс. Но для нас это приложение является демонстрацией возможностей ADO.NET и эту задачу оно выполняет.

```
namespace PictureInDb
{
    public partial class Form1 : Form
    {
        SqlConnection conn = null;
        SqlDataAdapter da = null;
        DataSet ds = null;
        string fileName = "";
        string conn_string = ConfigurationManager.
               ConnectionStrings["MyConnString"].
               ConnectionString;

        public Form1()
        {
            InitializeComponent();
            this.Text = "Picture Library";
            conn = new SqlConnection(conn_string);
            //подключение к БД
        }
```

```
        ///<summary>
        ///В этом обработчике пользователь может
        ///выбрать картинку для загрузки в БД.
        ///После выбора картинки вызывается метод
        ///LoadPicture() в котором картинка
        ///преобразовывается в байтовый
        ///массив и заносится в БД
        ///</summary>
        ///<param name="sender"></param>
        ///<param name="e"></param>
        private void btnLoad_Click(object sender,
                                   EventArgs e)
        {
            OpenFileDialog ofd = new OpenFileDialog();
            ofd.Filter = "Graphics File|*.bmp;*.gif;*.
                         jpg;*.png";
            ofd.FileName = "";
            if (ofd.ShowDialog() == System.Windows.
                         Forms.DialogResult.OK)
            {
                fileName = ofd.FileName;
                LoadPicture();
            }
        }

        ///<summary>
        ///В этом методе создается и выполняется
        ///параметризированный запрос
        ///insert, который заносит уменьшенную копию
        ///выбранной картинки в БД
        ///Уменьшенная копия создается в методе
        ///CreateCopy()
        ///</summary>
        private void LoadPicture()
        {
            try
            {
```

```
            byte[] bytes;
            bytes = CreateCopy();
            conn.Open();
            SqlCommand comm = new SqlCommand("insert
                    into Pictures (bookid,
                    name, picture) values
                    (@bookid, @name, @picture);",
                    conn);

            if (tbIndex.Text == null ||
               tbIndex.Text.Length == 0) return;
            int index = -1;
            int.TryParse(tbIndex.Text, out index);
            if (index == -1) return;

            comm.Parameters.Add("@bookid",
                 SqlDbType.Int).Value = index;
            comm.Parameters.Add("@name",
                 SqlDbType.NVarChar, 255).
                 Value = fileName;
            comm.Parameters.Add("@picture",
                 SqlDbType.Image, bytes.Length).
                 Value = bytes;
            comm.ExecuteNonQuery();
            conn.Close();
        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.Message);
        }
        finally
        {
            if (conn != null) conn.Close();
        }
    }

    /// <summary>
    /// В этом методе анализируется ориентация
```

```
        ///картинки и создается копия этой картинки
        ///таким образом, чтобы максимальный размер
        ///картинки (высота или ширина) не превышал
        ///300 пикселей
        ///Пропорции картинки при этом не искажаются
        ///</summary>
        ///<returns>байтовый массив, содержащий
        ///уменьшенную копию </returns>

        private byte[] CreateCopy()
        {
            Image img = Image.FromFile(fileName);
            int maxWidth = 300, maxHeight = 300;
            //размеры выбраны произвольно
            double ratioX = (double)maxWidth /
                            img.Width;
            double ratioY = (double)maxHeight /
                            img.Height;
            double ratio = Math.Min(ratioX, ratioY);

            int newWidth = (int)(img.Width * ratio);
            int newHeight = (int)(img.Height * ratio);

            Image mi = new Bitmap(newWidth, newHeight);
            //рисунок в памяти
            Graphics g = Graphics.FromImage(mi);
            g.DrawImage(img, 0, 0, newWidth, newHeight);
            MemoryStream ms = new MemoryStream();
            //поток для ввода|вывода байт из памяти
            mi.Save(ms, ImageFormat.Jpeg);
            ms.Flush();//выносим в поток все данные
            //из буфера
            ms.Seek(0, SeekOrigin.Begin);
            BinaryReader br = new BinaryReader(ms);
            byte[] buf = br.ReadBytes((int)ms.Length);
            return buf;
        }
```

```
        ///<summary>
        ///в этом методе выполняется запрос,
        ///выводящий все записи
        ///таблицы Pictures
        ///</summary>
        ///<param name="sender"></param>
        ///<param name="e"></param>
        private void btnAll_Click(object sender,
                                  EventArgs e)
        {
            try
            {
                da = new SqlDataAdapter("select *
                     from Pictures;", conn);
                SqlCommandBuilder cmb =
                          new SqlCommandBuilder(da);
                ds = new DataSet();
                da.Fill(ds, "picture");
                dgvPictures.DataSource =
                          ds.Tables["picture"];
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message);
            }
        }

        ///<summary>
        ///в этом методе выполняется запрос,
        ///выводящий запись
        ///таблицы Pictures по заданному id
        ///</summary>
        ///<param name="sender"></param>
        ///<param name="e"></param>
        private void btnIndex_Click(object sender,
                            EventArgs e)
        {
```

```
        try
        {
            if (tbIndex.Text == null ||
                        tbIndex.Text.Length == 0)
            {
               MessageBox.Show("Укажите id книги!");
               return;
            }
            int index = -1;
            int.TryParse(tbIndex.Text, out index);
            if (index == -1)
            {
                MessageBox.Show(«Укажите id книги
                          в правильном формате!»);
                return;
            }

            da = new SqlDataAdapter("select
                 picture from Pictures where id =
                 @id;", conn);
            SqlCommandBuilder cmb =
                 new SqlCommandBuilder(da);
            da.SelectCommand.Parameters.Add("@id",
                 SqlDbType.Int).Value = index;
            ds = new DataSet();
            da.Fill(ds);
            byte[] bytes = (byte[])ds.Tables[0].
                 Rows[0]["picture"];
            MemoryStream ms =
                 new MemoryStream(bytes);
            pbShowPictures.Image =
                 Image.FromStream(ms);
        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.Message);
        }
    }
```

```
        ///<summary>
        ///Запретить вводить в TextBox не числовые
        ///значения
        ///</summary>
        ///<param name="sender"></param>
        ///<param name="e"></param>
        private void tbIndex_KeyPress(object sender,
                KeyPressEventArgs e)
        {
            if ((e.KeyChar <= 48 || e.KeyChar >= 59)
               && e.KeyChar != 8)
                e.Handled = true;
        }
    }
}
```

Проверим наше приложение. Посмотрите в таблице Books значения id для тех книг, картинки для которых вы

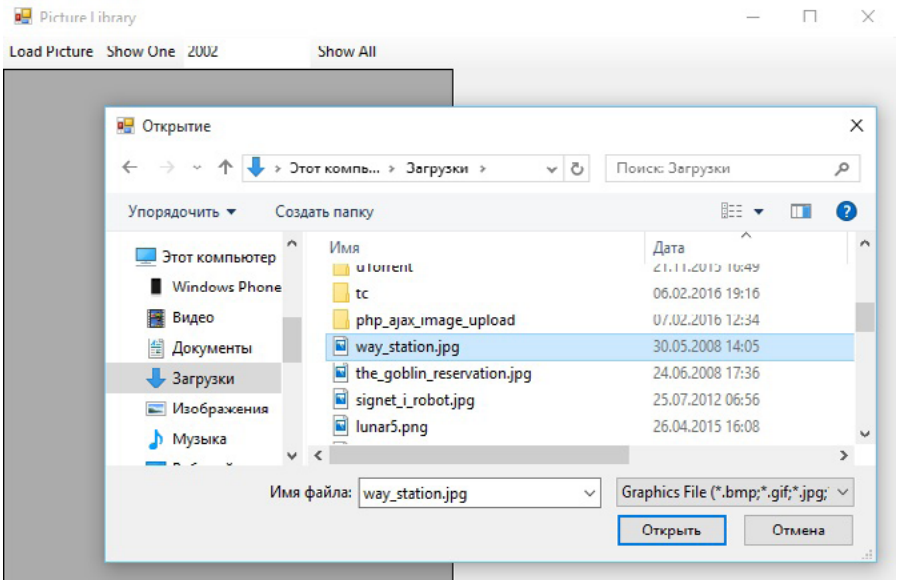

Рис. 8. Добавление картинки

хотите добавить в таблицу Pictures. Запустите приложение, введите в текстовое поле выбранный id и нажмите кнопку LoadPicture, затем выберите картинку и нажмите OK. Если вы забудете ввести код книги, приложение подскажет вам об этом в диалоговом окне. Если же вы укажете неверный код для книги, вы также получите сообщение об этом. В этом случае — как описание исключительной ситуации.

После добавления нескольких картинок выполните действие ShowAll. Возможно, то, что вы увидите, для некоторых будет сюрпризом. Оказывается, элемент DataGridView умеет отображать графическую информацию из бинарного содержимого! На приведенном рисунке вы видите, как это выглядит. Для более удобного отображения картинок я немного изменил размеры ячеек DataGridView.

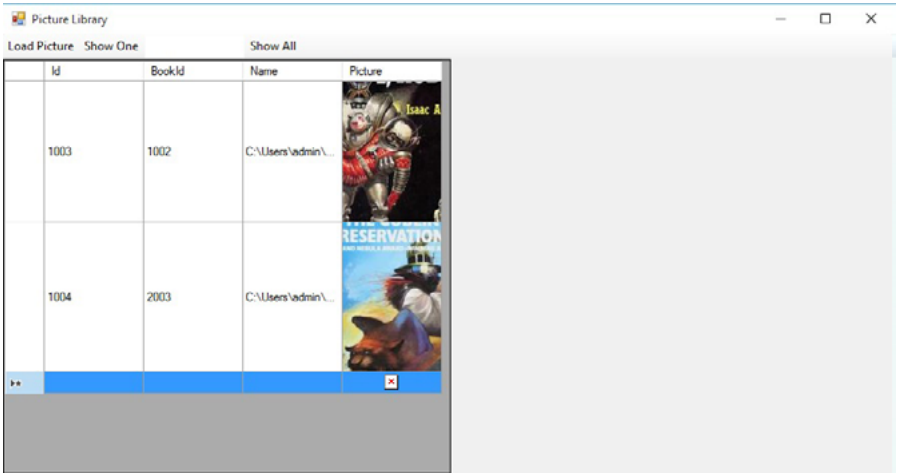

Рис. 9. Выполнение действия ShowAll

Имея перед глазами результат выполнения ShowAll, запомните id какой-либо картинки, занесите его в текстовое

поле и нажмите кнопку ShowOne. Теперь вы имеете возможность увидеть в правой части окна загруженную картинку полностью.

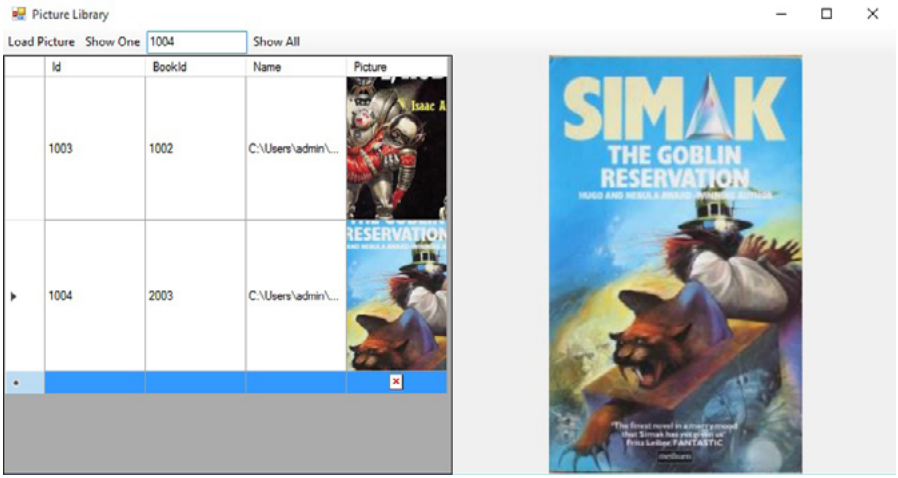

Рис. 10. Выполнение действия ShowOne

Предварительные итоги

Вы познакомились с базовыми возможностями ADO.NET. Теперь вы должны четко понимать, в чем заключается разница между присоединенным и отсоединенным режимами работы. Вы должны понимать, как применять основные классы из пространства имен System.Data. Впереди нам предстоит рассмотреть еще несколько интересных возможностей этой технологии и увидеть, к каким замечательным результатам привело развитие этой технологии.

Домашнее задание

Вспомните приложение TestDataTable, рассмотренное в начале этого урока. В том приложении были проблемы с очередностью выполнения запросов. Вам предлагается переписать это приложение, внеся следующие изменения:

- вместо присоединенного режима работы использовать отсоединенный;
- вместо одного элемента DataGridView, отображающего результаты выполнения всех запросов, использовать элемент TabControl, в котором результаты каждого введенного запроса будут отображаться в новой вкладке;
- вместо элемента DataGridView, отображающего результаты, использовать элемент ListView, позволяющий просматривать результаты запроса в разных режимах отображения (например, в режимах details и list);

Метод Update() в этом приложении использовать не надо. Способ переключения разных режимов отображения в ListView придумать и создать самостоятельно.

Если сформулировать задание коротко, то оно будет звучать так. При вводе в командное текстовое поле трех разных запросов select, приложение должно создать три вкладки. На каждой вкладке создать элемент ListView и отобразить в этом элементе результаты выполнения одного из введенных запросов. Обеспечить пользователю переключение для просмотра результатов запроса в двух разных видах: в табличном и списочном.