Database Access Technology

ADO.NET

Lesson №2

Disconnected mode

Contents

Disconnected mode

In the first lesson you have considered the application of the connected mode when working with the database. When using this way of operation, we have created the DbConnection object by transferring the connection string to it. Then we have called the Open() method on behalf of this object opening the database connection. Next, we have performed some actions, understanding that the connection remains open until calling the Close() method.

This way of operation is optimal in terms of the performance, but suboptimal in terms of the server load. Since the server has to hold open all the connections, through which there is no interaction with the database most of the time.

In this lesson you are going to study an alternative way of working with the database – the disconnected mode. This mode is based on using two classes: DataSet and DbDataAdapter. To understand these two classes, at first one should consider the DataTable class.

DataTable

Relational databases are always related to tables. Therefore, no wonder, that in the .NET Framework there are a number of classes designed for data table storage and display. There are such graphic controls as DataGridView and ListView and the DataTable class. This last class will be very useful for us in

our future work. Therefore, let's consider the main principles of its application.

To do this, create a new application, not a console application, but that with a window. Let's place a text field in this window, where we will enter a query, a button and a DataGridView element, wherein we will display the query results. Create a click event handler for the button. When clicking on this button, our application will have to execute the select query entered into the text field and display the results of this query in DataGridView. The DateTable class will greatly simplify obtaining and displaying the results of the executed query. The main purpose of our application is to consider this class. In odred to do this, we will use the connected mode once again.

> *We will not create a new database for this application, but we will continue to use the database created in the first lesson. To do this, in the new application we should use the same connection string, that was used to connect to our database in the first lesson.*

Run Visual Studio 2015 and create a Windows Forms application named TestDataTable with the following window (Fig.1).

Add the connection string from the application of the first lesson to the created application. DataTable is a typical table element. When working with it, one should follow the rules of the table elements operation. First of all, you should understand that after creating a DataTable object, we obtain a table template that is empty initially.

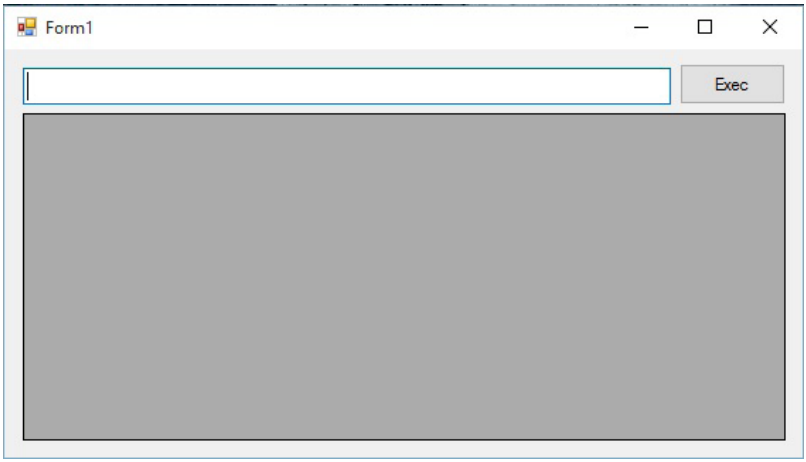

Fig. 1. Main application window.

Before doing something with any table element, one should form columns of the desired table. To do this, one should specify how many columns the table will contain and how they will be named. Perhaps, in addition to this, one should specify a type of data that will be placed in the columns, and width of the columns.

The DataTable base structure is simple. There is a Columns property whose elements have the DataColumn type and describe the table columns. The DataColumn type, in turn, contains important properties, defining the table columns. These are ColumnName, DataType, AllowDBNull and DefaultValue. To create a new table column, it's enough to add a string column name to Columns. One shouldn't do anything with strings until the Columns collection formation..

In the DataTable class there is one more Rows property, whose elements have the DataRow type and they are table strings.

Recall how we have extracted the field names of the database table when reading its data from the SqlDataReader element in the application of the first lesson. Using this technique, we will form the columns of the DataTable object so, that it will correspond to any read table. If our select query has read the Books table, then the DataTable object will repeat the Books table structure. If our query has read the Authors table, then the DataTable object will repeat the structure of this table. In other words, the DataTable object allows us to recreate the structure of any read table dynamically. Let's consider a simple example. We will create the Authors table structure in the DataTable element manually and add several strings.

```
DataTable table = new DataTable(); //an empty table
                                   //was created
table.Columns.Add("id");//a new id column was created
table.Columns.Add("FirstName");//a new FirstName
                               //column was created
table.Columns.Add("LastName"); //a new LastName
                               //column was created
//now one can enter rows into the table
//every row is an object of the DataRow type
DataRow row = table.NewRow();
//the NewRow() method creates a row corresponding to
//the table on behalf of which it was called
// in our case row will be an array of three elements
// because we have already formed three columns in
// the table fill the row object elements with the
// suitable data and enter into the table

row[0] = 1;
row[1] = "Francis";
row[2] = "Becon";
table.Rows.Add(row);// a new row was added to the table
        // other rows are added similarly
```

After examining the example of the DataTable object creation, you are ready to understand the code which should be added to the button handler of our application. Note that in our application the conn object was created and initialized in the window constructor globally, therefore it is available in the following code example.

```
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())
        {
           //at first iteration we form columns
           if (line == 0)
           {
              for (int i = 0; i < reader.FieldCount; i++)
              {
                 table.Columns.Add(reader.GetName(i));
              }
              line++;
           }
           //since the columns are already ready,
  //then at each iteration we create and fill the next row

           DataRow row = table.NewRow();
           for (int i = 0; i < reader.FieldCount; i++)
           {
```

```
                row[i] = reader[i]; //fill the row from
                                    //reader
            }
            table.Rows.Add(row);    //add the next row
        }
    } while (reader.NextResult());
    //welcome to binding
    dataGridView1.DataSource = table;

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

As you see, the row of adding the DataTable object to dataGridView1 is marked in a special way in this code. Here, one uses binding through the DataSource property that removes the necessityto fit the dataGridView1 element structure to our table structure.

Add the above handler to the created application. In the configuration file create a connection string to the database of the first lesson. Bring the whole application code to the following form:

```
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();
             }
         }
    }
}
```

Run the application, enter any select query to our database into the text field and click the button. You should see the query result in the dataGridView1 element:

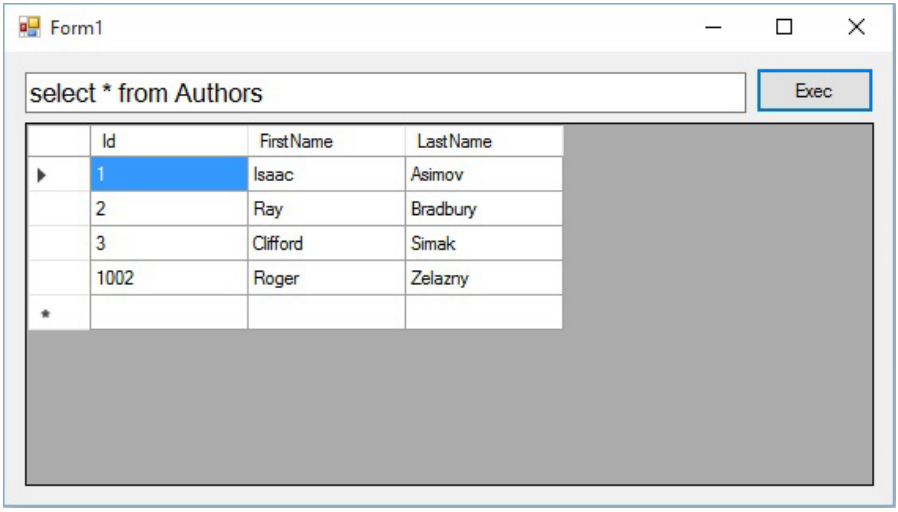

Fig. 2. Application execution

Note, that the table field names are displayed in the first dataGridView1 string. It's a result of our nested for loop executed at first iteration in the while loop.

As you can see, the while loop wherein the entries are extracted from SqlDataReader is enclosed in the do-while loop, controlled by the NextResult() method. It means that we can enter several queries separated by the semicolon into the text field. But herewith, we should take into account the following feature of this code. The table structure is formed at first iteration. That is, when processing the results of the first query. Therefore, if the resulting table of the first query exceeds the tables of other queries by the number of fields, everything will be output correctly. But if the first table

contains fewer fields than the result of the next query, it will cause an error. In our case, in the Books table there are more fields than in the Authors table. Therefore, if we enter the queries in such a sequence "select * from Books; select * from Authors;" they will be executed correctly. If these two queries are reversed, it will cause an error. In order to process this situation, the try-catch-finally block is added to the code. In your homework, you should make sure that such an error does not arise.

Now, when we have got acquainted with the DataTable class, let's proceed to the consideration of the DataSet and DbDataAdapter classes.

Data local storage (DataSet)

DataSet is a container capable of storing data from one or several tables in the memory. This class is similar to the DbDataReader class, but the important difference is that the DataSet class is used when there is no open database connection, while DbDataReader requires the open connection.

In the DataSet class the Tables property is defined, it's a collection, whose elements are objects of the DataTable type.

Since this class is capable of storing data of several tables, and the tables can be related, then it's necessary to store the information about the relationships somewhere. To store the information about the relationships, the Relations property is used. Relations propertyis a collection whose elements are objects of the Relation type.

This class allows the application to load a set of the required data from the database. Even data from several different tables.

This data, of course, is the data returned by any select queries. After uploading data into the DataSet object, the application can disconnect from the server and continue to work with this data locally. Then, if it's necessary, the application can connect to the server again and return the database data that were changed locally.

The DataSet object is usually created by the constructor without parameters. But data loading into this object occurs automatically when executing the special method, which we will get acquainted with in the next section.

Let us conduct the imaginary experiment. Abstract from the fact how the data arose in the DataSet object. Let's assume that in our object there are results of the select * from Authors and select * from Books queries. How can one work with this data?

Since there are two queries, then in the Tables collection there will be two elements, two tables. One can refer to them either by indexes or by names. The "table" and "table1" names will be the default names and so on. But these defaults can be changed. In order to imagine how we can refer to the DataSet data, look at the following code examples.

```
//create a ds object
DataSet ds = new DataSet();
// fill the ds object with the results of our two
// queries in any way

// we extract the first table by index from ds
DataTable dt1 = ds.Tables[0];
// we extract the second table by index from ds
DataTable dt2 = ds.Tables[1];
```

```
// we extract the first table by name from ds
DataTable dt3 = ds.Tables["table"];
// we extract the second table by name from ds
DataTable dt4 = ds.Tables["table1"];

// we  extract the author's LastName from the first
// row of the dt1 table by indexes
string lastName1 = dt1.Rows[0][1].ToString();
// we  extract the author's LastName from the DataSet
// object directly
string lastName2 = (string)ds.Tables[0].Row[0].
["LastName"];
```

Database connection (DbDataAdapter)

The DbDataAdapter class is the main character in the disconnected mode when working with the server. This class is responsible for opening a database connection, obtaining data from a database or uploading data into a database and for closing the connection. If one formulates the actions of this class more precisely, then it can be said that this class controls the data transfer from a database to a DataSet object and from a DataSet object to a database. The DbDataAdapter class works together with the DataSet class. The last one is a container for the first one.

In all fairness, one should say that the DbCommand and DbDataReader classes are used somewhere inside the DbDataAdapter class. But everything is done automatically, and if I haven't told you about this, you wouldn't have a possibility to guess.

The DbDataAdapter class has four main properties of the DbCommand type:

- SelectCommand (for reading the database data);

- InsertCommand (for adding the database data);
- UpdateCommand (for changing the database data);
- DeleteCommand (for deleting the database data).

Each of these properties should store the relevant query which will be executed when connecting to a database. It's not necessary to initialize all of these four properties. But the SelectCommand property should always be initialized. We are going to talk about different initialization methods of these properties below on this page.

In addition to these four class properties, the Fill() and Update() methods also play the important role.

> *Don't associate the Update() method name with the update SQL query. This method changes the database state by executing the SQL queries that are entered into the InsertCommand, UpdateCommand and DeleteCommand properties.*

When the Fill() method is called, it executes the query entered into the SelectCommand property. The data formed as a result of this query is entered into the DataSet (or DataTable) object. Note, that when using the DbDataAdapter class, we shouldn't execute the connection opening and closing. If the connection is closed, DbDataAdapter will open it and close it immediately when it will finish the action with the database. If the connection is open when calling Fill() or Update(), it will remain open after the DbDataAdapter completion.

Schematically the DbDataAdapter operation can be described so:

```
//to create a DbDataAdapter object one should have
//the select query and DbConnection object
SqlConnection conn = new SqlConnection(@"Data Source=
                (localdb)\v11.0;InitialCatalog=
                Library;Integrated Security=SSPI");
                String selectSQL =
                "SELECT * FROM Authors";
//create a DbDataAdapter object
SqlDataAdapter da = new SqlDataAdapter(selectSQL, conn);

//read this row explanation below
SqlCommandBuilder cmdBldr = new SqlCommandBuilder(da);

//create a DataSet object for local storage of the
//database data
DataSet ds = new DataSet();

//the select query of the SelectCommand property
//executes the call of the Fill() method
// and enters the read data into the DataSet object

da.Fill(ds);
```

Separately one should tell about the code string that is highlighted in yellow. Strictly speaking, all of the four command properties of the DbDataAdapter object should be filled: SelectCommand, InsertCommand, UpdateCommand and DeleteCommand. We will enter the queries into each of these properties in different ways. The SelectCommand query works as an input query for the application, it transfers data from the database to the application. Queries in other three command properties work as output ones synchronizing the application data with the database.

If we shouldn't interfere in the data synchronization process explicitly, we can do in such a way. Initialize the SelectCommand property explicitly, using the required query. For other three command properties we should create special default queries. To create these default queries, the SqlCommandBuilder class is used. The application of this class is very simple – one should only create its object by transferring the adapter object which requires such command queries to the constructor.

We are going to consider how to control the synchronization process explicitly after getting acquainted with the next section of this lesson.

Example of using DbDataAdapter

Let's create one more application wherein we are going to consider how to use the DbDataAdapter object in practice. As in the previous case, it will be a Windows Forms application named TestDataAdapter. Similarly in the window of this application we will place a text field where we will enter the queries and a DataGridView element, which will display the query results. We will need two buttons. Let's sign them like Fill and Update. We will create click handlers for both buttons. Everything is like in the previous case. But now we will work with the DbDataAdapter class.

Enter the connection string to our Library database into the application configuration file. Like in the previous application, add the reference to the System.Configuration namespace. Run our application and make sure that the main window looks like it's shown in the picture.

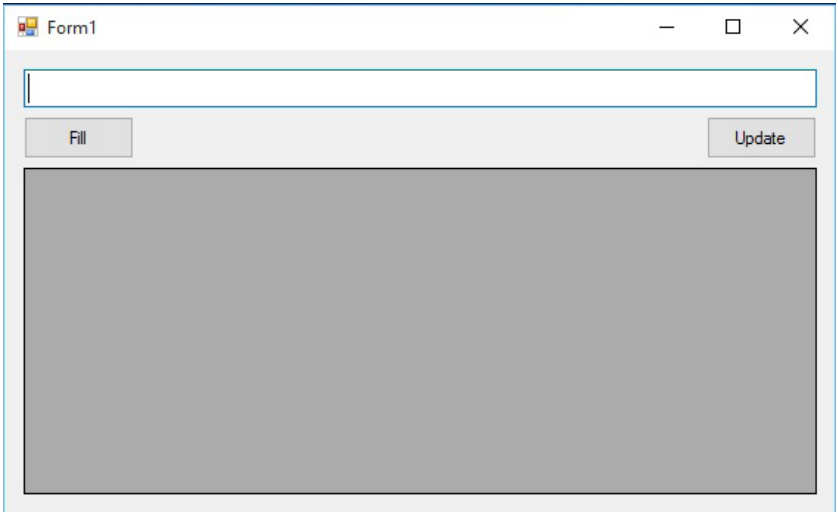

Fig. 3. Application main window

Bring the application code into line with the instance shown below.

```
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");
        }
    }
}
```

Let's look what the DbDataAdapter class can do. Run the application, enter the select * from Authors query into the text field and click the Fill button. When clicking this button, the required objects will be created: SqlConnection, DataSet and SqlDataAdapter. Then the Fill() method will be executed, and it will enter the result of the specified select query into the DataSet object. The result of this query will be placed in the Tables property, and the first table named "mybook" will be formed there. The table name can be specified in the optional second parameter of the Fill() method. If we executed several select queries, the result of the second one would be in the table named "mybook1", the result of the third one would be in the table named "mybook2" and so on.

It makes sense to specify the names in the Fill() method, if you don't want to deal with indexes and memorize what table is placed in Tables[0], and what table is placed in Tables[1]. If we didn't specify the table name, two strings highlighted in yellow would look as follows:

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

In addition to this, even if we didn't specify the table names in the Fill() method, all the same, we could refer to the DataSet tables by "table", "table1" names and so on. These names are assigned to the DataSet tables by default. After clicking the Fill button, you will see the filled dataGridView1. Although outwardly it looks like the work with our previous application, there is a principal difference.

There is the overloaded Fill() method version, that allows you to extract a part of strings returned by the select query from the database. Such a call da.Fill(ds, 0, 10, "mybook") will enter the first 10 strings into DataSet and place them in the Tables collection with the "mybook" index. However, keep in mind that the overload of this method all the same extracts all the query strings from the database, but further the unnecessary strings are discarded. Therefore there is no sense to talk about the optimality.

Now when you are observing the Authors table data in the window of your application, you are disconnected from a database. All the read database data are placed now in the DataSet object and are available for you locally. You can do with it whatever you want and then you can return them to the database by changing your database content in such a way.

Navigate to the last string of the dataGridView1 element and enter the name and surname of the new author. One shouldn't enter anything into the identifier field. The value of this field is formed automatically. After adding a name and surname of a new author into dataGridView1, the window of your application can look as follows (Fig. 4).

Now click the Update button. Believe it or not, but a new string has been just added into the Authors table of your database. However, I don't urge you to believe. I urge to check my statement. Click the Fill button again to re-read the data from the Authors table. In the table you will see a new string with a new value of the identifier assigned to this string (Fig. 5).

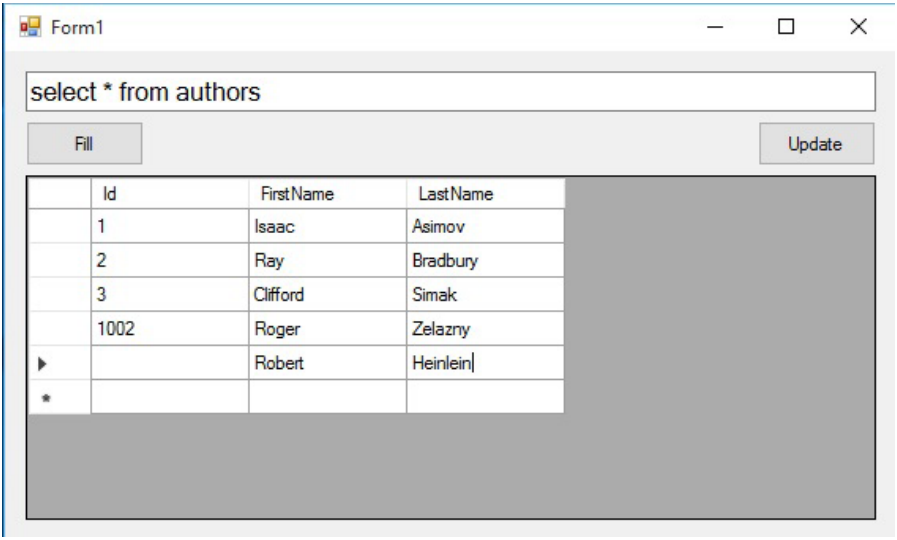

Fig. 4. Adding a new author

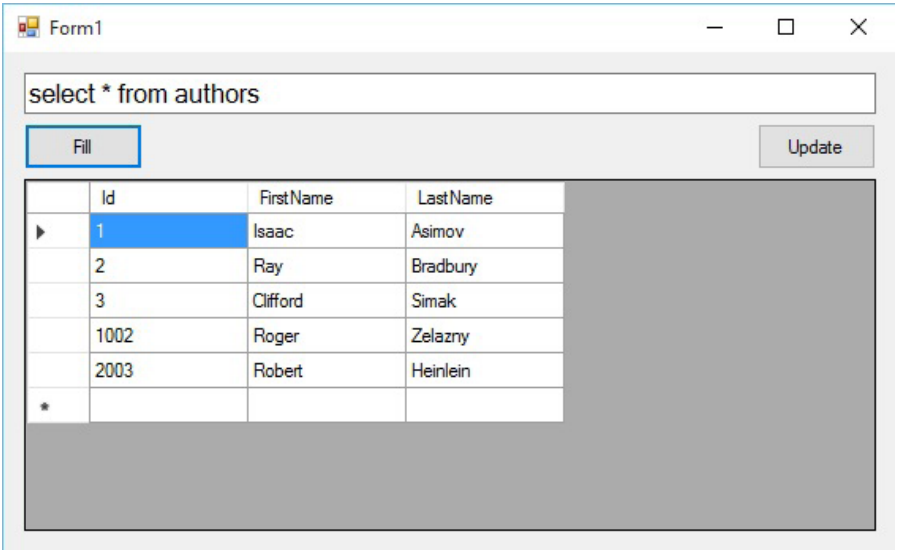

Fig. 5. Reading the changed data

So, working with local data (while the application was disconnected from the database), you added a new entry into DataSet. Then you called the Update() method of the adapter object. This method connected to the database and synchronized its local data with the table, from which this data was read earlier! After completing the synchronization, the Update() method disconnected from the database.

Similarly you could change any value in any field or even delete any string. All the changes would be synchronized with the database when executing the Update() method.

If you want to know how it occurs, wait a bit. We are going to consider all of this soon. Now it's very important to do the following experiment.

Run our application again and execute such a query: select title, price, pages from books. Navigate to any dataGridView1 string and change the value of any field, for example, price. After entering a new price, click the Update button. At this time something went wrong. If we translate the explanation of the arisen exceptional situation, you will see a message that the dynamic database synchronization is not supported for the select query, that doesn't return any information about the primary key of the table. If you think about it, it's logical. We excluded the id field output from the select query. And when trying to synchronize the data in DataSet with the data in the database table, the SqlDataAdapter object doesn't understand where each string in DataSet was read from and where this string should be returned to in the database. Let's memorize this result.

In order to make possible the automatical synchronization of the local data in DataSet with the data in the database tables by the Update() method, the local data should be entered into DataSet by the query that returns the primary key or another unique key of the table.

Using SqlCommandBuilder

The last example allows us to formulate the following principle of the SqlDataAdapter class application. When calling the Fill() method, the query stored in the SelectCommand property is executed. The execution of this query leads to the DataSet object filling. After this, the disconnection from the server occurs and it's time to work with local data. Then in any moment the Update() method is called. It leads to the execution of queries stored in the InsertCommand, UpdateCommand and DeleteCommand properties. The execution of these queries leads to changing the database content.

It should be considered in more detail. We create a query for the SelectCommand property by ourselves. And where do the queries in the InsertCommand, UpdateCommand and DeleteCommand properties come from? They are entered into SqlDataAdapter by the SqlCommandBuilder class when creating an object of this class. Let's look at these queries. It's not difficult to do.

Add the following strings to the code of our application after creating the SqlCommandBuilder object:

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

and you will see the queries created by the SqlCommandBuilder class for these three command properties. My queries look like this:

```
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)))
```

You see that these are parameterized queries, configured to change the database with the exact match of parameters. We don't know yet, where the parameters for the execution of these queries come from. If it seems to you that these queries are not very optimized, you are right. Imagine the effectiveness of these queries for the tables with dozens of fields, most of which, moreover are not indexed. Therefore, note, that if the performance is critical for your application, it's better to prepare these queries by yourself, but don't entrust their creation to the SqlCommandBuilder class. As always in life, if it is necessary to do something important – do it yourself, and do not entrust it to others.

The queries created by the SqlCommandBuilder class depend on the select query stored in the SelectCommand property. Therefore, if you changed the select query in the SelectCommand property, you should restructure the queries in three other command properties. In order to do this, one should call the RefreshSchema( ) method of the SqlCommandBuilder object.

Remember the main limitations of the SqlCommandBuilder class:

- *The synchronization queries created by it are not optimized;*
- *This class allows you to create the modification queries only for data obtained using the single-table select query, that is, if the multiple-table query was entered into your SelectCommand property, you shouldn't rely on SqlCommandBuilder;*
- *This class doesn't support the stored procedures.*

User logic in the database synchronization

Let's consider how we can enter our own queries into the InsertCommand, UpdateCommand and DeleteCommand properties. DbCommand is a type of each of these properties. We can already work with this type. You should create such an object, initialize its CommandText property and enter this object into the relevant SqlDataAdapter command property.

Let's consider the example of creating the own query for the UpdateCommand property. Creating the queries for other properties occurs similarly. Very often we shouldn't allow the user to change any values in the table. For example, it's logically

to allow you to change only the book price. Therefore, our user query will be specialized, allowing you to change only the price.

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

//create parameters for the Update query
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";

//insert the created SqlCommand object into the
//UpdateCommand SqlDataAdapter property
da.UpdateCommand = UpdateCmd;
```

We have already considered the SqlParameter type, but then we haven't talked about its SourceVersion and SourceColumn properties. These properties define the value that should be entered into the parameter. For the update query there are two versions of the table string to be changed:

- Initial string that is in the table now;
- Updated string with changes that should be entered into the table.

  In order to be able to distinguish such strings, the

SourceVersion property is used. Its values are defined in the DataRowVersion enumeration. The DataRowVersion.Original value corresponds to the first string, the DataRowVersion. Current value corresponds to the second string. In our example we specify that we enter the price value into the parameter from the "new" modified string (DataRowVersion.Current), and to compare the strings we choose id from the initial string (DataRowVersion.Original). DataRowVersion.Current is the default value of this property.

The SourceColumn property specifies from which field the value is selected.

To initialize the adapter command properties one can use not only the SQL queries but the stored procedures also. In most cases the stored procedures are the preferred option. Let's look how we can use the stored procedures in the command properties of the SqlDataAdapter object. Let's assume that there is the following stored procedure in our database:

```
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
```

Let's consider how the UpdateCommand property can be initialized by this stored procedure and how one can prepare the parameters.

```
//create a SqlCommand object and initialize it by the
//stored procedure
SqlCommand updateCommand = new
SqlCommand("UpdateBooks", conn);
updateCommand.CommandType = CommandType.
                            StoredProcedure;

// create parameters for  the stored procedure
// in order to do this, create a reference of the
// Parameters type collection and relate it
// with the Parameters property of the created
// updateCommand object it will be convenient to
 // add parameters through such a reference

SqlParameterCollection cparams;

cparams = updateCommand.Parameters;

// we add parameters for the stored procedure
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");

//initialize the adapter command UpdateCommand
//property
da.UpdateCommand = updateCommand;
```

Similarly one can create the stored procedure for other adapter command properties. Now, when calling the Update() method, not the SQL queries but the stored procedures will be executed, that is more optimal in terms of the performance and security.

Sequence of executing the database changes (DataViewRowState)

We have already considered the Update() method execution in detail. But there are some more features that should be considered. Let's assume that your application has fulfilled the Fill() method, read the database data, disconnected from the server and begun to process the read data locally. When processing, some strings were changed, some of them were added and deleted. How do you think, what will be the sequence of database changes when calling the Update() method? And whether this sequence matters?

If you think over these questions, you will have to come to conclusion, that the sequence is important, of course. Moreover, we should have a tool, allowing us to control the sequence of the executed database changes. Let's get acquainted with the method of controling the database changes.

When calling the Update() method, the strings are processed one by one. Each string has the DataViewRowState

property, whose available values are defined in the enumeration with the same name. The value of this property depends on actions, executed with the string. If any string was deleted, the value of this property of the string would be equal to DataViewRowState.Deleted, for the added string this value would be equal to DataViewRowState.Added, for the changed string this value would be equal to DataViewRowState.ModifiedCurrent, for the string that remained unchanged this property would be equal to DataViewRowState.Unchanged. Using this property, we will write a code, which will allow us to execute the synchronization of the local data from our application with the database in the necessary sequence.

The Select method, defined in the DataTable class and allowing you to select the strings with the specified value of the DataViewRowState property from the table, will help us to do this. At first, from the DataSet object we will select the table data which we want to synchronize with our database. Then, instead of the single call of the Update() method, we will call this method several times by transferring to it only strings with the specified type of changing, only the deleted strings or only the added strings for processing. We can select these strings using the Select() method that returns the array of the DataRow type. Thus, we will execute the synchronization in the sequence that is required in our application.

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

//at first we will process the deleted rows
da.Update(table.Select(null, null,
                        DataViewRowState.Deleted));

//now we will process the modified rows
da.Update(table.Select(null, null, DataViewRowState.
                       ModifiedCurrent));

//and now we will process the added rows
da.Update(table.Select(null, null, DataViewRowState.
                        Added));
```

Note, that right after calling the Fill() method, all the strings, entered into DataSet, have the RowState property value that is equal to DataViewRowState.Unchanged. The AcceptChangesDuringFill boolean property of the SqlDataAdapter object is responsible for this. By default the value of this property is equal to True and all the read database strings obtain RowState that is equal to DataViewRowState.Unchanged.

If you set the AcceptChangesDuringFill equal to False, then right after calling the Fill() method they will have the RowState property equal to DataViewRowState.Inseretd. If you need to apply all the changes, you can call the AcceptChanges() method of the DataSet object, that will transfer the RowState property to the DataViewRowState.Unchanged value. The following code example shows the changes of the string states:

```
DataSet set = new DataSet();
da.Fill(ds, "mybooks");
//by default after Fill() each string in "mybooks"
//has RowState = Unchanged
```

```
//delete everything from the "mybooks" table
ds.Tables["mybooks"].Clear();

//modify the adapter property
//AcceptChangesDuringFill = false


da.AcceptChangesDuringFill = false;

//call the Fill() method again to fill the "mybooks" table
da.Fill(ds, "Orders");
//now after Fill() each string in "mybooks" has
//RowState = Inserted

//call of AcceptChanges changes the RowState property
//value for all the strings
ds.AcceptChanges();
//now each string in "mybooks" has RowState =
//Unchanged
```

Displaying the tables (TableMappings)

Let's consider the situation when the SelectCommand adapter command property was initialized by the following string:

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

There is no Publisher table in our database. But, let's imagine that there is such a table. It would be more useful for our example if we had more than two tables. If you want, add such a table to the database, but it may remain unchanged and can be confined to the imaginary experiment. Next, we will execute the standard call of the Fill() method:

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

After calling the Fill() method, there will be three tables in the Tables property of our DataSet. Each of these tables will contain the result of one of the select queries. Remember, how these tables will be named in the Tables property. By default the first table will be named "Table", the second table will be named "Table1", and the third table will be named "Table2". If you, for example, want to display the data from the Books table in DataGridView, you will have to write the following code:

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

or the following one:

```
dataGridView1.DataSource = set.Tables["Table1"];
```

In another words, you should remember, that the query of the Books table was entered into the adapter as the second one or you should remember that the table with the results of this query is named "Table1". And what to do if there are many tables? And what to do if we have weak imagination? Would you like the data from the Authors database table to be placed in DataSet of the "Authors" table, the data from the Books database table to be placed in DataSet of the "Books" table and so on? Then look how it can be done.

The adapter has the public DataTableMappingCollection TableMappings { get; } property.

The type of this DataTableMappingCollection property is

defined in the System.Data.Common namespace. It's a collection whose elements are objects of the DataTableMapping type. The TableMappings property allows us to bring the database table names in line with the table names in DataSet. You will see soon that this property allows you to specify not only the match between the table names, but something else. But at first we will consider the display of the table names

Our goal is to create a table with the same Authors name for the database Authors table in DataSet, and similarly for the other tables. For this purpose we can do the following:

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

Now, after executing the Fill() method, the results of each of our three queries will be placed in DataSet in the tables with the names that match the names of the initial database tables.

These actions can be written more fully:

```
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);
```

Here, naturally, such a question arises: can we execute the displays not only for the table names, but for the column names of the tables as well? Yes, we can. For this purpose there is the DataTableMapping object property that is named ColunmMappings:

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

By default, each column returned by the select query matches the column with the same name in the table created in DataSet. You can change such behavior in two ways. For example, if you specify aliases for the fields in the select query:

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

the columns created in DataSet will be named according to the aliases.

However, you can use the more flexible mechanism, that is provided by the ColumnMappings property. Let's consider the example for one select query, wherein we want to replace both the table name and the column names:

```
string strSQL = "SELECT id, FirstName,
                 LastName FROM Authors";
SqlDataAdapter da = new SqlDataAdapter(strSQL, conn);
//display the table names
DataTableMapping dtm = da.TableMappings.Add("Table",
                "Authors");
//display the column names
dtm.ColumnMappings.Add("id", "Num");
dtm.ColumnMappings.Add("FirstName", "Name");
dtm.ColumnMappings.Add("LastName", "Surname");
da.Fill(ds);
```

You understand perfectly that since both properties TableMappings and ColumnMappings are collections, then we can not only add new elements but delete the existing ones in them. There are Remove() and RemoveAt() methods for this purpose.

Now in the application you can work with the data read from the database more conveniently.

Working with graphic information

All the created examples of the applications are illustrations showing the different aspects of using ADO.NET. Let's create an application, which will contain certain client logic. We will continue to work with our database Library, but we will make some changes. Let's add the table wherein we will store the pictures for our books. In order to be able to store an arbitrary number of pictures for any book in the Books table, we will create a new table.

```
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)
)
```

We will store the pictures for different books in this table. The picture will be bound to a particular book using the relation with the Books table. As you see by the Picture field type, I propose to store not the path to the picture, but namely the binary content in the table.

Here one should make a small digression. We often face with the following superstition: one shouldn't store the binary content in the database tables, it's necessary to store only the

paths to the pictures. Let's discuss this statement. Why did it appear? Proponents of this superstition believe that the tables become "very big" in this case and are processed "very slowly". Yes, there is such a fact. Especially, if we enter the binary content of 1.5–2.0 GB into each table string. Because varbinary(MAX) allows you to store such volumes.

But if you behave decently and limit yourself to the pictures of a few megabytes, nothing terrible will happen. And very often it's enough to deal with even smaller pictures.

Let's evaluate the pros and cons of each approach objectively. It makes sense to store the binary data in the file system but not in the tables in cases, when the disk space is limited for a database, for example, by an expensive hosting. Or when the binary data, which your application works with, should be available for other applications.

Storage of binary data solves the data integrity issues. This data is always synchronized. The binary objects in the tables are archived at backup and you shouldn't take any further actions for their transfer at database transfer.

If all these arguments didn't persuade you, and somebody else is afraid of the varbinary type, here is one more argument in favor of this type. You remember, of course, that any database must ensure its data integrity. It's one of the most important database functions. For this purpose many limitations, triggers and many other things were created. Now let's imagine that you store your pictures in the folder on the disc, but the paths to these pictures are stored in the table. If the pictures in your folder are modified, deleted, renamed, how will the database even not interfere with it, but simply know about it? In no way.

And where is the integrity? Why do I need the fast table that contains invalid data? Do as you wish, but varbinary is cool!

To make it easy, let's forbid the user to enter the king-size pictures into our database. Let's allow the user to choose the picture of any size, and our application will create a reduced copy for the chosen picture, that doesn't exceed, for example, 300 pixels in height and width and will enter this smaller copy into the table. I hope everybody understands that we plucked the value of 300 pixels "from the air"?

So. Add a new table to our database. We have already done this action earlier. Let's repeat it once again. In the SQL Server Object Explorer window we expand the Tables node of our database and select the Add New Table option from the context menu… We insert the query for creating our new Pictures table into the N-SQL window (located in the bottom of the window).

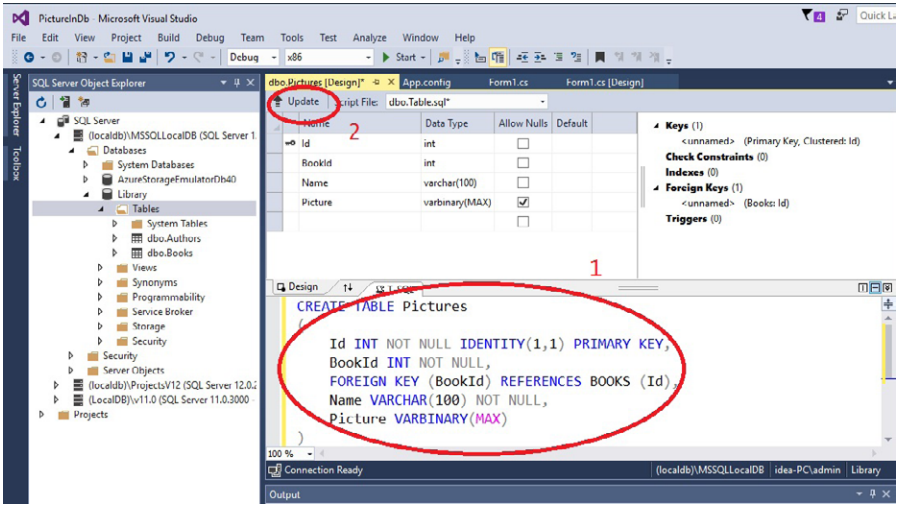

Fig. 6. Adding a new table

Click the Update button on the top toolbar and then click on the Udpate Database button in the appeared window. We have created a new table and have related it to the Books table.

Now we run VisualStudio 2015 and create a new Windows Forms application. We will go on to use the modified database Library. Although the database was changed, we can connect to it like in the previous projects. Therefore, copy the content of the configuration file of any of our previous projects to the configuration file of the created application. The newly created application window can look as follows.

Fig. 7. Main application window

It's not necessary to create a toolbar and make the window like in the picture. But one should provide the execution of the following actions:

- loading a new database picture bound to any book – the Load Picture button and the text field on the toolbar, wherein one should specify the book id, are responsible for this;
- outputting the information about the specific entry of the Pictures table from the database – the Show One button is responsible for this, herewith one should specify the entry id of the Pictures table in the text field on the toolbar;
- outputting the information about all the entries of the Pictures table – the Show All button is responsible for this;
- displaying the results of the select queries;
- Displaying the graphic information from the varbinary (MAX) field.

The query results are displayed in the DataGridView element at the bottom left part of the window, the picture of the Show One action execution is displayed in the PictureBox element in the right part of the window. Bring the code application interface to the form. The code is commented enough and understandable. For the real using one should add the serious validation and the user interface. For us this application is a demonstration of the ADO.NET opportunities and it performs this task.

```
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);
    //database connection
}
///<summary>
///In this handler the user can select a
/// picture to load into the database.
///After selecting a picture, the
///LoadPicture() method is called
///wherein the picture is converted to a byte
///array and is entered into the database

///</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>
        ///In this method we create and execute the
        ///insert parameterized query wherein
        ///a reduced copy of the selected picture is
        ///entered into the database
        /// The reduced copy is created in the
        ///CreateCopy() method
        ///</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>
        /// In this method the picture orientation is
        /// analyzed and the copy of this picture
        /// is created so that the maximum picture
        /// size (height and width) will not exceed
        /// of 300 pixels
        /// Herewith the picture proportions are not
        /// distorted
        ///</summary>
        ///<returns> a byte array that contains a
            smaller copy </returns>

        private byte[] CreateCopy()
        {
            Image img = Image.FromFile(fileName);
            int maxWidth = 300, maxHeight = 300;
            //Sizes were chosen arbitrarily
            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);
            //picture in the memory
```

```
            Graphics g = Graphics.FromImage(mi);
            g.DrawImage(img, 0, 0, newWidth, newHeight);
            MemoryStream ms = new MemoryStream();
            //a stream to input/output bytes from the
            //memory
            mi.Save(ms, ImageFormat.Jpeg);
            ms.Flush();//bring all the buffer data to
                       //the stream
            ms.Seek(0, SeekOrigin.Begin);
            BinaryReader br = new BinaryReader(ms);
            byte[] buf = br.ReadBytes((int)ms.Length);
            return buf;
        }
        ///<summary>
        ///in this method we execute the query that
        ///outputs all the entries of the
        /// Pictures table
        ///</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>
    ///in this method we execute the query that
    ///outputs the Pictures table entry by the
    ///specified 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("Specify the book
                                        id!");
               return;
            }
            int index = -1;
            int.TryParse(tbIndex.Text, out index);
            if (index == -1)
            {
                MessageBox.Show("Specify the book
                     id in the correct format!");
                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>
        ///Forbid to input not numerical values into
        ///TextBox
        ///</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;
        }
    }
}
```

Let's check our application. In the Books table look at the id values of the books for which the pictures will be added to the Pictures table. Run the application, enter the selected id into the text field and click the LoadPicture button and then select the picture and click OK. If you forget to enter the book code, the application will tell you about it in the

dialog box. If you specify the incorrect book code, you will obtain a message about this. In this case – as a description of the exception.

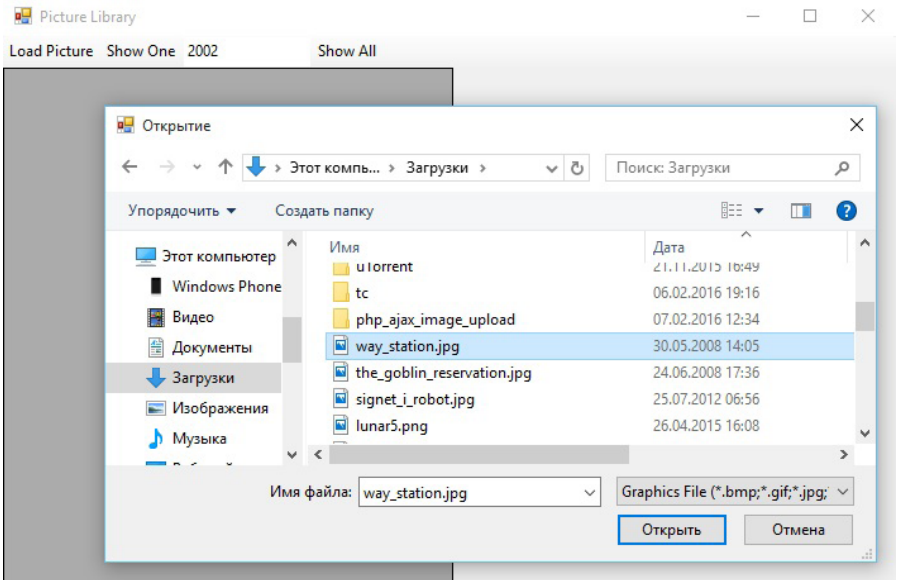

Fig. 8. Adding a picture

After adding several pictures, execute the ShowAll action. Perhaps all what you will see will be a suprise for someone. It turns out that the DataGridView element can display the graphical information from the binary content! In the picture you can see how it looks. For more convenient display of the pictures, the DataGridView cell sizes were changed a bit.

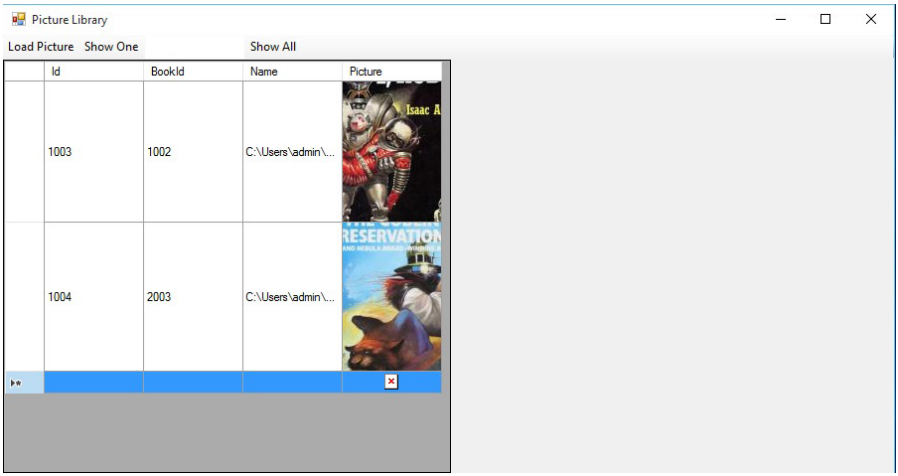

Fig. 9. ShowAll action execution

Having in front of eyes the ShowAll result, memorize id of any picture, enter it into the text field and click the ShowOne button. Now you can see the uploaded picture fully in the right part of the window.

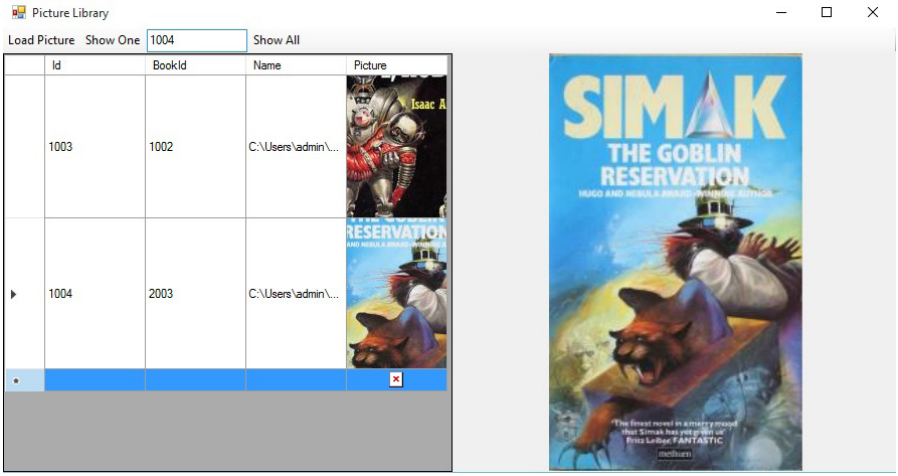

Fig. 10. ShowOne action execution

Preliminary conclusions

You have got acquainted with the basic ADO.NET opportunities. Now you should clearly understand what is the difference between connected and disconnected operation mode. You should understand how to apply the main classes of the System.Data namespace. We are going to consider some more interesting opportunities of this technology and see the remarkable results which the development of this technology led to.

Home task

Recall the TestDataTable application that was considered at the beginning of this lesson. There were problems with the sequence of the queries execution in that application. You are offered to rewrite this application by making the following changes:

- use the disconnected operation mode instead of the connected operation mode;
- instead of one DataGridView element that displays the results of all the queries, use the TabControl element, wherein the results of each input query will be displayed in a new tab;
- instead of the DataGridView element that displays the results, use the ListView element which allows viewing the query results in the different display modes (for example, in the details and list modes).

The Update() method shouldn't be used in this application. You should come up and create the switching method of the different display modes in ListView by yourself.

If one formulates the task briefly, it will sound like this. When inputting three different select queries into the command text field, the application should create three tabs. On each tab one should create a ListView element and display the results of one of the input queries in this element. One should provide the switching for viewing the query results in two different forms: in the table view and in the list view.