STEP IT ACADEMY

Database Access Technology

ADO.NET

Lesson №4

Asynchronous operations

Contents

New async and await tools

General principals

With the appearance of the .NET Framework 4.5 platform, the substantial changes have occurred in the implementation of the asynch methods of programming. The asynch methods to access data sources, which were considered in the previous lesson, remain valid for the .NET Framework 4.5 as well. But now one shouldn't specify Asynchronous Processing=true in the connection string. However, this .NET Framework version offers you a number of new asynchronous tools, which we are going to get acquainted with now. The async and await specifiers are these tools. The methods written using these specifiers are called asynchronous.

If we describe these innovations briefly, two features should be noted:

- Compiler executes most of the work on writing the asynchronous code;
- Structure of the written asynchronous code looks like the synchronous one.

The signature of the asynchronous method that uses new tools now looks as follows:

```
async Task<int> Method1Async()
{
    int result;
    // Your actions
    return result;
}
```

or so:

```
async Task Method2Async()
{
    // Your actions
}
```

Let's note the main points:

- Async specifier should be specified before the method;
- Void, Task, or Task<T> should be a return value type, where Task is used, if the method should return void, and Task<T> is used, if the method should return the value of the T type;
- The method name should end with an "Async" suffix.

Such methods should be called with the await specifier:

```
//Keyword await used with a method that returns
//a Task<int>.
int result = await Method1Async ();

//Keyword await used with a method that returns a Task.
await Method2Async ();
```

Generally speaking, the above calls of the async methods have also an expanded form that is more useful in cases, when we should have a mechanism to impact on the awaited task while it's running. The expanded calls of our two methods can look as follows:

```
//Calls to Method1Async
Task<int> returnedTaskInt = Method1Async();
int result = await returnedTaskInt;
//Calls to Method1Async
Task returnedTaskVoid = Method2Async();
await returnedTaskVoid;
```

We can call methods with the await specifier only inside methods, anonymous methods, and lambda expressions specified by the async specifier.

Now, we are going to consider in detail how these new tools are applied. When should we use async and await?

These tools are useful in the situations, when your code should execute the blocking actions. It's an access to the remote network resources, it's a work with input/output threads, it's a conversion of binary objects, and, of course, web.

We should especially note cases, when an application works actively with UI elements created in a primary thread. In each of these situations, it's desirable to get rid of the blocking programming model and use asynchrony. I want to say once again that we can use the "classic" asynchronous model considered in the previous lesson. We have a good alternative now, which has a good chance to become the main model.

Let's review the use of async and await for executing an action that has nothing to do with using data providers and consequently, with ADO.NET. Let's simply consider the asynchronous read from a file. This simple example will allow you to understand the principle of using the async and await specifiers. And after getting acquainted with the principle, we will apply this technique to execute our tasks.

Let's assume that it's necessary to read data from a file in our application and display the read information in TextBox of the main application window. In order to execute this action we have created the following method:

```
// The async specifier and Async suffix in the method
// name tell us that this method is asynchronous,
// and this means that we will be able to use the
// await calls in this method a path to the file that
// should be read is transferred to the method at the
// input

async private Task GetDataAsync(string filename)
{
    // prepare an array to receive the read data
    byte[] data=null;

    // create a thread for reading
    using (FileStream fs = File.Open(filename,
                                      FileMode.Open))
    {
        // create an array for reading
        data = new byte[fs.Length];
        // call the ReadAsync method built into the
        // FileStream async class the method name
        // makes it clear that it is asynchronous,
        // thus, one should call it with the await
        // specifier
        await fs.ReadAsync(data, 0, (int)fs.Length);
    }
    // convert the read data from the byte array to a
    // string and display in the Result text field
   Result.Text = System.Text.Encoding.UTF8.
                                      GetString(data);
}
```

This method clearly demonstrates what it should execute. Let's now talk about how this method will execute its work using async and await.

This method clearly demonstrates what it should execute. Let's now talk about how this method will execute its work using async and await.

As a rule, the async method should contain one or several await statements. However, if there are no such statements in the async method, the compiler will not qualify this as an error. It will output the warning message in case you simply forgot to use await. There is an await call in our async GetDataAsync() method. You understand that in the code, in turn, the call of our GetDataAsync() method should be somewhere. Since our method is asynchronous, it should be called with the await specifier. Let's present the code of calling our asynchronous method and strings that are adjacent to this call:

```
//some actions before calling the method
Console.WriteLine("Before async call");
await GetDataAsync("1.txt");
Console.WriteLine("After async call");
//some actions after calling the method
```

Note, that this call with the await specifier will demand the calling method to have async. That is, the above three code strings should be located inside of some async method.

And now attention. Let's consider our code implementation, beginning with the string:

```
await GetDataAsync("1.txt");
```

The control is transferred to the called method, and the following strings highlighted in yellow are executed successively in it:

```
async private Task GetDataAsync(string filename)
{
    byte[] data=null;

    using (FileStream fs = File.Open(filename,
                                FileMode.Open))
    {
        data = new byte[fs.Length];
        await fs.ReadAsync(data, 0, (int)fs.Length);
    }
    Result.Text = System.Text.Encoding.UTF8.
                                GetString(data);
}
```

In order to understand what happens after calling the ReadAsync() method, one should know the following. The async method returns the control in two cases: when it completes the work, and when *some await call occurs* in it. Therefore, after calling ReadAsync(), the control returns immediately to the called method (*because there is an await call there)* – and the "After async call" string is displayed on the console. When ReadAsync() completes its work, the control will be transferred to the method that has called ReadAsync() to the string located right after the await call. That is, the control will be transferred to the GetDataAsync() method (*because the async method completes its work)*. And at this moment the read information will be entered into the Result text field.

It is important to understand that the await specifier does NOT block the thread wherein it is called. Instead, await *specifies that the compiler should add the currently executed code that follows the await call to the run queue right after completing the called (awaited) task. Then the control is returned to the thread that called the async method. When the awaited task is completed, the execution of the added code that contains the async method continuation will start automatically from the point where it has called the await.*

So, let's sum up what the async method is. If you have marked some method as "async", this means that you can use the await calls inside it. This specifier signals to the compiler that this method should be compiled in a special way: so that it is possible to suspend it and then resume its execution. The await calls in the method are suspension/resumption points. Actually, the async method is not asynchronous by definition. If there are no await calls in the async method, then such a method will be executed as synchronous. There is an opinion that only async method with the Task return value type can be executed as synchronous if there are no await calls. But the async method with the Task<T> return value type will be executed asynchronously even if there are no await calls. This is wrong. The absence or presence of the await calls is a watershed for the synchronous or asynchronous execution of the async method. But even if there are await calls inside the async method, it can be executed as synchronous. This will happen in case if the compiler sees that the data, which should be obtained as a result of the await call, is ready and one shouldn't wait for them.

You should understand such simple statement: if there is an await call in the method, it doesn't mean that this method will be suspended in the point of this call until the called asynchronous operation will be completed. The async method would become blocking! If there is a call in the await method, it means that if the results of the called asynchronous operation are not ready yet, all the method strings followed after the await call are added by the compiler for execution AFTER completing the awaited asynchronous operation. And the control returns immediately to the method that called the current async method. When the asynchronous operation is completed, the execution of the method that called it will be continued immediately.

If there are await calls in the method, they divide this method into parts that can be executed separately. The part located before the await call will be executed first. After that, any other code will be executed, and after that, the next part of the async method located after the await call will be executed.

The most surprising fact about the async methods is that they do NOT create additional threads and they are NOT executed in additional threads.

Asynchrony doesn't mean an execution in the separate thread. Asynchrony means NON synchrony, that is, there is no blocking. The execution in a separate thread is one of the ways to avoid blocking calls. But it's not an only way.

The asynchronous methods are executed in their own execution context (in the thread wherein they are called) creating an "alternate multitasking". In other words, async methods insert the control transfer into their working thread in the moments, when the blocking can arise. And when the potentially blocking task is completed, the control returns to the string after calling this task. Just as the callback methods do. But herewith, we should NOT create delegates and callback methods, and transfer data between threads! Besides, if you remember that the process of creating a new thread and its synchronization with the main thread is resource-intensive, you will understand that the async methods can be more effective than new ones. It's useful to remind you again that you can create an async method using only a method with the void, Task, and Task<T> return type value.

Some information about the Task class

Does all of the above mean that async and await don't work with multithreading? No, it doesn't. If you need multithreading, you can create it using the async methods. To do this you should use the Task class. There is the Task.Run<T>() method in this class, which queues the code strings transferred to it to execute them in the ThreadPool. Let's consider how it can be done. Assume that there is a synchronous method, which execution is time consuming:

```
static string SlowMethod(string file)
{
    Thread.Sleep(3000);
    //reading file
    return string.Format("File {0} is read", file);
}
```

This method can be converted by any asynchronous method in order to avoid the arising blocking. For example, as follows:

```
static Task<string> SlowMethodAsync(string file)
{
    return Task.Run<string>(() =>
    {
        return SlowMethod(file);
    });
}
```

Look, the SlowMethodAsync() is asynchronous, we have specified this in its name. But we didn't specify the async specifier before this method. For there are no await calls inside SlowMethodAsync().Why is it asynchronous? Because it doesn't call blocking. Why doesn't it call blocking? Because it executes the work (calls the SlowMethod()) in the additional ThreadPool.

But SlowMethodAsync()should be called with the await specifier!

```
private async static void CallMyAsync()
{
    string result = await SlowMethodAsync("BigFile.txt");
    //сюда можно добавить и другие вызовы нашего метода
    //string result1 = await SlowMethodAsync("BigFile1.txt");
   //string result2 = await SlowMethodAsync("BigFile2.txt");
    Console.WriteLine(result);
}
```

This example shows that async and await can be used with the real multithreading too. You should choose the specific asynchrony either with or without the additional threads. You should choose the most suitable option in each specific situation.

Use of the Task class provides us with some more useful opportunities. Before, it was impossible to interrupt the execution of the method running in the ThreadPool. Now, we can do it. In order to stop the running task, you should get acquainted with the CancellationTokenSource class. Before calling the asynchronous method that uses ThreadPoll, you should transfer CancellationToken to it, wherein the cancel of the method executed in another thread is already programmed:

```
private async static void CallMyAsync1()
{
    try
    {
        var cts = new CancellationTokenSource();
        cts.CancelAfter(TimeSpan.FromSeconds(3));

           Task<string> t1 = SlowMethodAsync1("BigFile.
                                 txt", cts.Token);
           string result = await t1;
        Console.WriteLine(result);
    }
    catch (OperationCanceledException ex)
    {
        Console.WriteLine(ex.Message);
    }
}
```

The call of the CancelAfter() method will lead to the stoppage of SlowMethod1() that runs in the separate thread after the specified time period.

You should change two other methods accordingly:

```
static string SlowMethod1(string file,
                                            CancellationToken token)
{
      Thread.Sleep(3000);
      //reading file
      token.ThrowIfCancellationRequested();
      return string.Format("File {0} is read", file);
}

static Task<string> SlowMethodAsync1(string file,
CancellationToken token)
{
      return Task.Run<string>(() =>
      {
             return SlowMethod1(file, token);
      });
}
```

Async and await for the ADO.NET

Now, let's consider async and await in relation to the work with data providers. A number of asynchronous methods was added to the .NET Framework 4.5 in order to use them in the ADO.NET. Here are the most demanded of them:

- SqlConnection.OpenAsync
- SqlCommand.ExecuteNonQueryAsync
- SqlCommand.ExecuteReaderAsync
- SqlCommand.ExecuteScalarAsync
- SqlDataReader.NextResultAsync
- SqlDataReader.ReadAsync

A much longer list of new asynchronous methods is on this page: https://msdn.microsoft.com/en-us/library/hh211418(v=vs.110).aspx.

Let's consider how these methods are applied in practice. Run Visual Studio 2015 and create a new project named LibraryTest4. We will continue to work with our Library database, but now we will refer to it using async and await. Moreover, let's use the DbProviderFactory class in this project. It's optional, but it will be useful in term of repeating the reviewed information. This project will execute the actions which you are already familiar with. It will connect to the database specified in the connection string and execute the query in it, which is input by a user in the main application window. However, now our application will add the "WAITFOR DELAY '00:00:05';" query before the user's query in order to simulate the long processing and give us an opportunity to make sure that the main thread is not blocked.

Add the database connection string to the configuration file. Add the reference to the System.Configuration namespace to our project and make the application code look like this:

```
namespace AdoNetSample4
{
    public partial class Form1 : Form
    {
        DbConnection conn = null;
        DbProviderFactory fact = null;
        string connectionString = "";
        public Form1()
        {
            InitializeComponent();
            button1.Enabled = false;
        }
        private async void button1_Click(object sender,
                                         EventArgs e)
        {
            conn.ConnectionString = connectionString;

            await conn.OpenAsync();

            DbCommand comm = conn.CreateCommand();
            comm.CommandText = "WAITFOR DELAY '00:00:05';";

            comm.CommandText += textBox1.Text.ToString();
            DataTable table = new DataTable();
            using (DbDataReader reader = await comm.
                                       ExecuteReaderAsync())
  {
                int line = 0;

                do
                {
                    while (await reader.ReadAsync())
                    {
                        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] = await reader.
                              GetFieldValueAsync<Object>(i);
                        }
                        table.Rows.Add(row);
                    }
                } while (reader.NextResult());
            }
            //output the query results
            dataGridView1.DataSource = null;
            dataGridView1.DataSource = table;
        }
        ///<summary>
        ///When the window is uploaded,select the
        ///factory for the System.Data.SqlClient
        ///provider, and call the method for receiving
        ///the connection string
        ///from the configuration file

        ///</summary>
        ///<param name="sender"></param>
        ///<param name="e"></param>
        private void Form1_Load(object sender, EventArgs e)
        {
            fact = DbProviderFactories.
                     GetFactory("System.Data.SqlClient");
            conn = fact.CreateConnection();
            connectionString =
                    GetConnectionStringByProvider("System.
                    Data.SqlClient");
            if(connectionString == null)
            {
               MessageBox.Show("There is no required
                               connection string in the
                               configuration file");
            }
        }
        ///<summary>
        ///According to the provider name, this method
        ///reads the connection string from the
        ///configuration file and returns it, if this
        ///string is in the configuration file :)
        ///</summary>
        ///<param name="providerName"></param>
        ///<returns></returns>
        static string GetConnectionStringByProvider(
                                 string providerName)
        {
              string returnValue = null;
       //read all the connection strings from App.config
              ConnectionStringSettingsCollection
              settings = ConfigurationManager.
              ConnectionStrings;

              //find and return the connection string
              //for providerName
              if (settings != null)
              {
                  foreach (ConnectionStringSettings cs
                           in settings)
              {
                  if (cs.ProviderName == providerName)
                  {
                     returnValue = cs.ConnectionString;
                     break;
                  }
              }
           }
           return returnValue;
        }

        ///<summary>
        ///control the button accessibility
        ///</summary>
        ///<param name="sender"></param>
        ///<param name="e"></param>
        void textBox1_TextChanged(object sender,
                                  EventArgs e)
        {
            if (textBox1.Text.Length > 3)
                button1.Enabled = true;
            else
                button1.Enabled = false;
        }
     }
}
```

Let's focus on the code strings highlighted in yellow. There are four calls with the await specifier in our handler of the button. Therefore, we specified the async specifier near the handler. Void is a return value type of the handler that's why we can apply the async specifier to the handler. If you remember, each await call informs the compiler that the method (our handler) should be divided into parts, which can be executed separately, and we understand that our handler will be divided into five parts.

The connection string to the database server is the first of the highlighted strings:

```
await conn.OpenAsync();
```

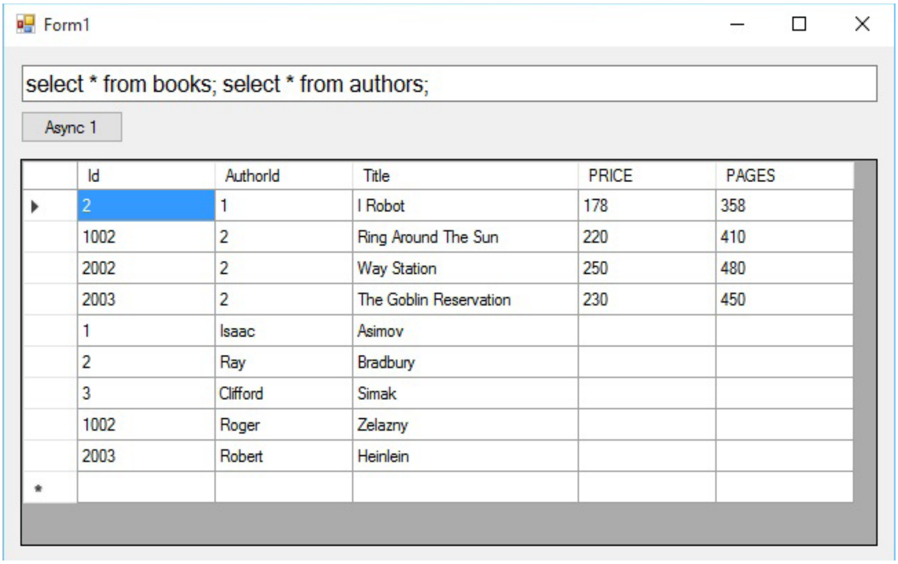
At this point the handler execution can be suspended and the control will be returned to the main thread, that is, a user will be able to execute other actions using the main application window. We have taken care of that the button is blocked when executing the query and a user can't input new queries until the current query process is completed. By the way, our procedure of connecting to the database server can be executed in the synchronous mode.

Next, we add the query that calls the five-second stop of the server to the DbCommand object. After that, in the await mode, we call the ExecuteReaderAsync() method, which should execute all queries added to the CommandText property, create a DbDataReader object, enter the query results into this object, and return the filled DbDataReader object to the object that calls. This operation is the most resource-intensive and time-consuming in our project. Especially considering our five seconds ☺ But, as you can see, if you run the application, there will be no blocking of the main thread. That is, we see that this complex operation is executed asynchronously.

Next, we have two more await calls, but they are much easier, than the ExecuteReaderAsync() execution. The below picture shows our application window after executing two queries that are input by a user and a hidden delay query that is added by the application. While these queries were being executed, the application window was responding to all the actions: movements, changing a window size and DataGridView columns, and others. That is, the window was not blocked (Fig.1).

So, the application is created and executes what we expect from it. But we have already done and seen all this. So how does this application differ from the previous asynchronous application?

This application differs from the previous one because it works with data providers asynchronously, but herewith, doesn't create the additional threads! Consequently, this application is more effective than the previous multithread one, because creating the additional threads complicates the application very much in terms of writing a code and consuming resources and time needed for execution.



select * from books; select * from authors;

	Id	AuthorId	Title	PRICE	PAGES
▶	2	1	I Robot	178	358
	1002	2	Ring Around The Sun	220	410
	2002	2	Way Station	250	480
	2003	2	The Goblin Reservation	230	450
	1	Isaac	Asimov		
	2	Ray	Bradbury		
	3	Clifford	Simak		
	1002	Roger	Zelazny		
	2003	Robert	Heinlein		
*					

Fig. 1. Query execution using async and await

Configuration file encryption

You have become accustomed to use the configuration file in our applications. You have to agree that this is a convenient way to store connection strings. In some situations it's the only way to keep the opportunity of changing the credentials for connecting to the data sources without the need to recompile the application. However, you certainly have noted for yourself the drawback of this approach. This drawback is that credentials for connecting to the database are stored in the text format in the open form, and consequently, they can be accessible for viewing.

Security issues are a separate big conversation topic. Now, we will consider it only partially. First of all, you should keep in mind other ways of storing and using connecting credentials that differ from the configuration file. If you are afraid to store the credentials for connecting to the server in the configuration file, don't store them there. Don't store these credentials anywhere. Simply create a dialog box in the application, where a user will have to enter the required data at each connection. However in this case, users will have to know these connecting credentials. I want to offer you now to consider an intermediate way to comply with the security. This method boils down to the encrypted connection string that is stored in the configuration file. That is, the user shouldn't know the connecting credentials, and one can't take a look at these credentials in the configuration file. In some cases, such a solution will be optimal.

In order to implement such approach we should use the RSAProtectedConfigurationProvider or DPAPIProtectedConfigurationProvider class. Each of these classes uses its own method to encrypt the specified section of the configuration file. For the rest, they work similarly. They read the section of the ConnectionStringsSection configuration file, which stores the connection string, encrypt its content using its own algorithm, and record back to the file as encrypted. Each time you read this section, it's decrypted automatically and the application gets access to the data provider which is specified in the connection string. Let's see how this approach works in practice. Create a new Windows Forms application. Place two text fields in the application window, where we will display the connection string of the configuration file at different stages of the application execution. Next, in order to make sure that the section encryption in the configuration file doesn't violate the database access, in the window place a button and DataGridView, using which we will execute the test connection and display results of the test query execution. Add the database connection string to the configuration file:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <startup>
        <supportedRuntime version="v4.0"
         sku=".NETFramework,Version=v4.5" />
    </startup>
    <connectionStrings>
    <add name="MyLibrary"
    providerName="System.Data.SqlClient"
    connectionString=
    "Data Source=(localdb)\v11.0;Initial
         Catalog=Library; Integrated Security=SSPI;"
    />
    </connectionStrings>
</configuration>
```

Pay attention that MyLibrary is a connection string name. We will refer to this name in the application code.

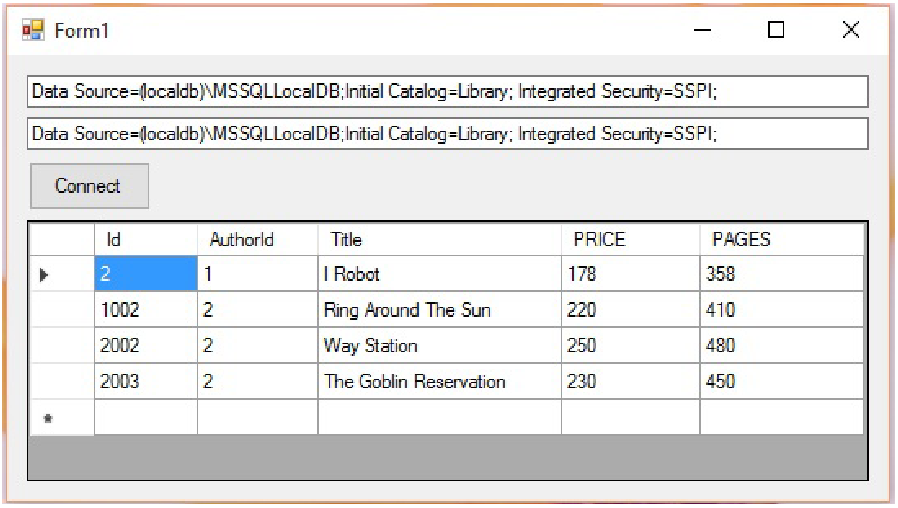
Bring the code of the created project to the following form:

```
namespace AppConfigDecrypt
{
   public partial class Form1 : Form
   {

//not encrypted connection string
       string connStringNotEncrypted = "";
```

```
//encrypted connection string
      string connStringWithEncryption = "";
      public Form1()
      {
          InitializeComponent();
      //read the connection string before encryption
          connStringNotEncrypted =
          ConfigurationManager.
          ConnectionStrings["MyLibrary"].
          ConnectionString;

      //display the connection string before encryption
          textBox1.Text = connStringNotEncrypted;

      //execute the encryption
          EncryptConnSettings("connectionStrings");

      //read the connection string after encryption
          connStringWithEncryption =
          ConfigurationManager.ConnectionStrings
          ["MyLibrary"].ConnectionString;
          //display the connection string after encryption
          textBox2.Text = connStringWithEncryption;
      }

      private static void
                 EncryptConnSettings(string section)
      {
      //create an object of our configuration file
      //AppConfigDecrypt.exe is a name of the executed
      //file of your application if you have another
      //name, bear this in mind

         Configuration objConfig = ConfigurationManager.
         OpenExeConfiguration(GetAppPath() +
         "AppConfigDecrypt.exe");
         //get access to the ConnectionStrings section
         //of our configuration file
```

```
            ConnectionStringsSection conSringSection
            = (ConnectionStringsSection)objConfig.
            GetSection(section);
            //if the section is not encrypted, encrypt it
            if (!conSringSection.SectionInformation.
            IsProtected)
            {
               conSringSection.SectionInformation.
                   ProtectSection(
                   "RsaProtectedConfigurationProvider");
               conSringSection.SectionInformation.
                   ForceSave = true;
               objConfig.Save(ConfigurationSaveMode.
                   Modified);
            }
         }

         //get the path to the folder, wherethe
         //configuration file is
         private static string GetAppPath()
         {
            System.Reflection.Module[] modules =
                System.Reflection.Assembly.
                GetExecutingAssembly().GetModules();
            string location = System.IO.Path.
 GetDirectoryName(modules[0].FullyQualifiedName);
            if ((location != "") && (location[location.
                Length - 1] != '\\'))
                location += '\\';
            return location;
        }

        private void button1_Click(object sender, EventArgs e)
        {
            SqlConnection conn=null;
            SqlDataAdapter da = null;
            DataSet set = null;
            SqlCommandBuilder cmd = null;
            string connString = "";
           //re-read the connection string from the
           //encrypted section

           connString = ConfigurationManager.
               ConnectionStrings["MyLibrary"].
               ConnectionString;
           //redisplay the connection string in the
           //application window
           textBox2.Text = connString;

           try
           {
              conn = new SqlConnection(connString);
              set = new DataSet();
              string sql = "select * from books";
              da = new SqlDataAdapter(sql, conn);
              dataGridView1.DataSource = null;

              cmd = new SqlCommandBuilder(da);

              da.Fill(set, "Books");
              dataGridView1.DataSource = set.
Tables["Books"];
           }
           catch (Exception ex)            {
              MessageBox.Show(ex.Message);
           }
        }
      }
}
```

Run the application and click the Connect button. You will see a window similar to the below one. Let's talk about what happens in this application.



Data Source=(localdb)\MSSQLLocalDB;Initial Catalog=Library; Integrated Security=SSPI;

Data Source=(localdb)\MSSQLLocalDB;Initial Catalog=Library; Integrated Security=SSPI;

Id	AuthorId	Title	PRICE	PAGES
2	1	I Robot	178	358
1002	2	Ring Around The Sun	220	410
2002	2	Way Station	250	480
2003	2	The Goblin Reservation	230	450

Fig. 2. Configuration file encryption

Right after running the application, before clicking the Connect button, you see a regular connection string to our database in both text fields. All is clear with the string in the first text field. We output it there before encrypting, so we see it in the normal not encrypted form. But to the second text field, we output the connection string that has been read from the configuration file after calling the EncryptConnString() method, wherein the encryption is executed. Why in this case do we see the connection string in the open form again? Maybe, the connection string wasn't encrypted by the application. Navigate to the folder, which contains the exe.file of your application – the configuration file with the .exe.config extension. Open this configuration file in notepad. In my case, it looks as follows:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <startup>
      <supportedRuntime version="v4.0"
                  sku=".NETFramework,Version=v4.5" />
      </startup>
   <connectionStrings configProtectionProvider=
              "RsaProtectedConfigurationProvider">
      <EncryptedData Type="http://www.w3.org/2001/04/
              xmlenc#Element"
         xmlns="http://www.w3.org/2001/04/xmlenc#">
         <EncryptionMethod Algorithm="http://www.
              w3.org/2001/04/xmlenc#tripledes-cbc" />
         <KeyInfo xmlns="http://www.w3.org/2000/09/
              xmldsig#">
            <EncryptedKey xmlns="http://www.
                   w3.org/2001/04/xmlenc#">
               <EncryptionMethod Algorithm="http://
                   www.w3.org/2001/04/xmlenc#rsa-
                   1_5" />
               <KeyInfo xmlns="http://www.
                      w3.org/2000/09/xmldsig#">
                  <KeyName>Rsa Key</KeyName>
               </KeyInfo>
               <CipherData>
<CipherValue>tp4Kw1xrlACIAEGcEkU0ttlfAPVbT6U09zi/JThoGbK
Df+uF4vuFdOXY6KdprNpD8keLBycaaz50pWeX2oQOMIacs07o0zYmry
EM+tuZET5zdxSlasKGa6YApjfgH6/ALWae0izQLRBg+iesevyOmvj74/
FmxdypdpE2FMGn0ZA=</CipherValue>
               </CipherData>
            </EncryptedKey>
         </KeyInfo>
         <CipherData>
<CipherValue>6IhdBvExZfxjkmcHnqxhfCzAXQ4AwD5p9SKIVV+
M8lm0IZtwqMjz76rPv9vNqwF9ubDnsqNS1GrsuGOzzUKsB/926M2
4DKpOspSQwthYHVfF59hdxSmwCbhQTpJCR3F9zodBT/4MrgnSf7D+
TE368ddTcvS64BupV5WYo0kDYuDAP9++vfGEaPgbYIriPAz8dhro
SPn/Hpn3ZEH22eyBlQWYKymhRgCD7s1zGdAZJ/HHLtIdSnUdHgja4
3xcQIKE6N8buu+QqISeUkZL+IL3g13dt82A3MM45GppXtbmFEtSP
```

```
opYDzGH2HG2KMv70aPEkGnyvwnDxKviTbRMemczxBx5QjqQ/
YIX9kp9Q4eVTXAiJ9wKvw/NhhSxbwm6V2jswCiB/mIBVNkTVsu0a
0VhKYwkMddfyzPM0rFzFO3wCui7uS7TNCQsB+v8dWcYS78+sk+Ks
ER/TU6hLCldD0DYZTTN8ElnLo2DA2XhpjVqMziRwrmpJ9LSFRcyLI
H6qemJQYDTofSaShk=</CipherValue>
         </CipherData>
      </EncryptedData>
   </connectionStrings>
</configuration>
```

As you see, the connection string is encrypted and there are no confidential data accessible for viewing. It is simply decrypted automatically, when reading the encrypted connection string in the application, in order to use it properly further.

If you see the result of the test query in the window, this confirms the fact that the connection string is fully functional. But it's encrypted.

You should take into account that such a way of the configuration file encryption depends on the machines. In other words, if you try to use the configuration file, encrypted on one computer, on another computer, you will not succeed. Encryption should be executed on the computer, where it is supposed to use the configuration file.

Home task

As a home task, you should finish the LibraryTest4 application of this lesson. But you should add the combo box and label for displaying small numerical values to the main application window. The application work logic should be changed so that, when executing the "select * from books" query, the results should be displayed in DataGridView, as before. Herewith, the combo box, wherein you should enter the authors' names, whose books are in the Books table, should be filled simultaneously. When selecting any author in the combo box, in the added label there should be displayed the number of books of the selected author in the Books table.

The combo box should be filled only in the case, if a user enters a single "select * from books" query in the window. Without clarifying the fields, without any conditions, without any other queries. When entering any other queries, the combo box should remain empty.

Filling the combo box with the authors' names and filling the label with the number of books should be executed asynchronously using async and await.