Технология доступа к базам данных

ADO.NET

Урок №4

Асинхронный режим доступа

Содержание

Новые средства async и await

Общие принципы

С появлением платформы .NET Framework 4.5 в реализации асинхронных методов программирования произошли существенные изменения. Методы для асинхронного доступа к источникам данных, которые мы рассмотрели в предыдущем уроке, остаются в силе и для .NET Framework 4.5. Только теперь в строке подключения не надо указывать атрибут Asynchronous Processing=true. Однако данная версия .NET Framework предлагает ряд новых асинхронных инструментов, с которыми мы сейчас познакомимся. Этими инструментами являются спецификаторы async и await. Методы, написанные с использованием этих спецификаторов, называются асинхронными.

Если кратко описать эти нововведения, то надо отметить две характерные особенности:

- большую часть работы по написанию асинхронного кода компилятор берет на себя;
- структура написанного асинхронного кода внешне выглядит как синхронная;

Сигнатура асинхронного метода, использующего новые инструменты теперь выглядит так:

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

или так:

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

Отметим основные моменты:

- Перед методом должен быть указан спецификатор async;
- Тип возвращаемого значения должен быть void, Task или Task<T>, где Task используется, если метод должен вернуть void, а Task<T> — если метод должен вернуть значение типа T;
- Имя метода должно завершаться суффиксом Async;

Вызываться такие методы должны со спецификатором await:

```
//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 ();
```

Вообще то говоря, приведенные вызовы async методов имеют еще и развернутую форму, более полезную в тех

случаях, когда надо иметь механизм воздействия на запущенную ожидаемую задачу. Для наших двух методов развернутые вызовы могут выглядеть так:

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

Использовать вызов методов со спецификатором await можно только внутри методов, анонимных методов и лямбда-выражений, помеченных спецификатором async.

Сейчас мы подробно рассмотрим применение этих новых инструментов. Когда следует применять async и await?

Эти инструменты полезны в тех ситуациях, когда ваш код должен выполнять блокирующие действия. Это доступ к удаленным сетевым ресурсам, это работа с потоковым вводом-выводом, это преобразование бинарных объектов и конечно же - веб. Особенно надо выделить случаи, когда приложение активно работает с UI элементами, созданными в первичном потоке. В каждой из этих ситуаций желательно избавиться от блокирующей модели программирования и использовать асинхронность. Хочу еще раз сказать, что можно использовать «классическую» асинхронную модель, рассмотренную в предыдущем уроке. Сейчас у нас есть хорошая альтернатива, которая имеет все шансы стать основной моделью.

Давайте рассмотрим применение async и await для выполнения действия, не имеющего отношения к использованию

поставщиков данных, а следовательно — к ADO.NET. Просто рассмотрим асинхронное чтение из файла. Этот простой пример позволит понять принцип использования спецификаторов async и await. А познакомившись с принципом, мы уже затем применим эту технику для выполнения наших задач.

Предположим, что в нашем приложении надо прочитать данные из какого-либо файла и вывести прочитанную информацию в TextBox главного окна приложения. Для выполнения этого действия мы создали такой метод:

```
//спецификатор async и суффикс Async в имени метода
//говорят о том, что этот метод асинхронный, а это значит,
//что в этом методе мы сможем использовать await вызовы
//на вход методу передается путь к файлу, который
//надо прочитать
async private Task GetDataAsync(string filename)
{
    //готовим массив для приема прочитанных данных
    byte[] data=null;

    //создаем поток для чтения
    using (FileStream fs = File.Open(filename,
                                     FileMode.Open))
    {
        //создаем массив для чтения
        data = new byte[fs.Length];
        //вызываем встроенный в класс FileStream async
        //метод ReadAsync по имени этого метода ясно,
        //что он асинхронный, значит, вызывать его
        //надо со спецификатором await
        await fs.ReadAsync(data, 0, (int)fs.Length);
    }
    //преобразовываем прочитанные данные из байтового
    //массива в строку и отображаем в текстовом поле Result
```

```
    Result.Text = System.Text.Encoding.UTF8.
                                        GetString(data);
}
```

Этот метод наглядно демонстрирует, что он должен выполнять. Давайте теперь поговорим о том, как этот метод будет выполнять свою работу, благодаря использованию async и await.

Как правило, async метод должен содержать в себе одну или несколько инструкций await. Однако, если таковых в async методе не будет, компилятор не расценит это, как ошибку. Он выдаст предупреждающее сообщение, на случай, если вы просто забыли использовать await. В нашем async методе GetDataAsync() await вызов присутствует. Вы понимаете, что в коде, в свою очередь, где-то должен присутствовать вызов нашего метода GetDataAsync(). Поскольку наш метод асинхронный, то вызываться он должен со спецификатором await. Приведем код вызова нашего асинхронного метода и соседние с этим вызовом строки:

```
//какие-то действия до вызова метода
Console.WriteLine("Before async call");
await GetDataAsync("1.txt");
Console.WriteLine("After async call");
//какие-то действия после вызова метода
```

Отметьте, что этот вызов метода со спецификаторм await потребует наличия async у вызывающего метода. Т.е. приведенные три строки кода должны быть расположены внутри какого-либо async метода.

А теперь — внимание. Рассмотрим ход выполнения нашего кода, начиная со строки:

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

Управление передается вызванному методу, и в нем последовательно выполняются следующие строки, выделенные желтым цветом:

```
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);
}
```

Чтобы понять, что происходит после вызова метода ReadAsync() надо знать следующее. Async метод возвращает управление в двух случаях: когда он завершает работу, и когда в нем *происходит какой-нибудь await вызов*. Поэтому, после вызова ReadAsync() управление сразу возвращается в вызывающий метод (*т.к. здесь имеет место await вызов*) — и в консоль выводиться строка «After async call». Когда завершится работа ReadAsync() — контроль будет передан в метод, вызвавший ReadAsync(), в строку, расположенную сразу за await вызовом. Т.е., управление будет передано в метод GetDataAsync() (*т.к. здесь имеет место завершение работы async метода*).

И в этот момент в текстовое поле Result будет занесена прочитанная информация.

> *О спецификаторе await важно понимать, что он НЕ блокирует поток, в котором вызывается. Вместо этого, await указывает компилятору дописать текущий исполняемый код, идущий после вызова await, в очередь на выполнение сразу после завершения вызванной (ожидаемой) задачи. Затем управление сразу же возвращается потоку, вызвавшему async метод. Когда ожидаемая задача будет завершена, автоматически начнется выполнение дописанного компилятором кода, в котором содержится продолжение async метода, с того места, где он вызвал await.*

Итак, подведем итог — что такое async метод. Если вы пометили какой-либо метод, как async, это значит, что внутри этого метода вы можете использовать await вызовы. Для компилятора этот спецификатор является сигналом, что данный метод надо компилировать особым образом: так, чтобы его можно было приостанавливать, а затем возобновлять его выполнение. Точками приостановки/возобновления являются await вызовы в методе. На самом деле, async метод не является асинхронным по определению. Если в async методе нет await вызовов, то такой метод будет выполняться, как синхронный. Бытует мнение, что только async метод с типом возвращаемого значения Task, может выполняться, как синхронный, при отсутствии await вызовов. А async метод с типом возвращаемого значения Task<T>, даже при отсутствии await вызовов, будет

выполняться асинхронно. Это неверно. Водоразделом для синхронного или асинхронного выполнения async метода является отсутствие или наличие await вызовов. Но даже при наличии await вызовов внутри async метода, этот async метод может выполняться, как синхронный! Это будет происходить в том случае, если компилятор увидит, что данные, которые должны быть получены в результате await вызова, уже готовы и ожидать их не надо.

Надо понять такое простое утверждение: наличие в методе await вызова не означает, что данный метод будет приостановлен в месте этого вызова, до тех пор пока вызванная асинхронная операция будет завершена. Это превратило бы async метод в блокирующий! Наличие в методе await вызова означает, что если результаты вызванной асинхронной операции еще не готовы, то все строки метода, следующие после await вызова, дописываются компилятором на выполнение, ПОСЛЕ завершения ожидаемой асинхронной операции. А затем сразу же управление возвращается методу, вызвавшему текущий async метод. Когда асинхронная операция завершится, сразу же продолжится выполнение метода, вызвавшего ее.

Присутствие в методе await вызовов разбивает этот метод на части, которые могут выполняться по отдельности. Сначала выполниться часть метода до await вызова. Затем какой-либо другой код, затем следующая часть async метода, расположенная после await вызова.

Самый удивительный факт об async методах заключается в том, что они НЕ создают дополнительных потоков, и НЕ выполняются в дополнительных потоках.

Асинхронность не означает выполнения в отдельном потоке. Асинхронность означает НЕ синхронность, т.е. отсутствие блокировки. Выполнение в отдельном потоке — один из способов избежать блокирующих вызовов. Но не единственный способ.

Асинхронные методы выполняются в своем конексте выполнения (в том потоке, в котором они вызваны), создавая «чередующуюся многозадачность». Другими словами, async методы встраивают в свой рабочий поток передачу управления в те моменты, когда может возникнуть блокировка. А затем, при завершении потенциально блокирующей задачи, возвращают управление в строку, после вызова этой задачи. Аналогично тому, как поступают callback методы. Но, при этом, нам НЕ надо создавать делегаты, callback методы, передавать данные между потоками! К тому же, если вы вспомните о том, насколько ресурсоемким является процесс создания нового потока и его синхронизация с основным потоком, вы поймете, что async методы могут быть намного эффективнее, чем новые потоки. Еще раз полезно напомнить, что сделать async методом можно только метод с типом возвращаемого значения void, Task или Task<T>.

Немного о классе Task

Значит ли все вышесказанное, что async и await не работают с многопоточностью? Нет. Если вам нужна многопоточность, вы можете ее создать с помощью async методов. Для этого вам надо будет использовать класс Task. В этом классе есть метод Task.Run<T>(), который

ставит в очередь на выполнение в ThreadPool, переданные ему строки кода. Давайте рассмотрим, как можно это сделать. Предположим, что у нас есть синхронный метод, выполнение которого забирает много времени:

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

Мы можем обернуть этот метод каким-либо асинхронным методом, чтобы избежать возникающей блокировки. Например так:

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

Смотрите, наш метод SlowMethodAsync() асинхронный, мы отметили это в его имени. Но спецификатор async перед этим методом мы не указали. Ведь await вызовов внутри SlowMethodAsync() нет. Почему он асинхронный? Потому, что не вызывает блокировки. Почему не вызывает блокировки? Потому, что выполняет работу (вызов метода SlowMethod()) в дополнительном потоке ThreadPool.

Но сам метод SlowMethodAsync() должен вызываться со спецификатором await!

```
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);
}
```

Этот пример показывает вам, что async и await могут применяться и с реальной многопоточностью. Выбор конкретной асинхронности — то ли без дополнительных потоков, то ли с дополнительными потоками, за вами. В каждой конкретной ситуации надо выбирать наиболее подходящий вариант.

Использование класса Task предоставляет в наше распоряжение еще несколько полезных возможностей. Раньше у нас не было никакой возможности прервать выполнение метода, запущенного в потоке ThreadPool. Сейчас такая возможность есть. Для того, чтобы остановить запущенную задачу, надо познакомится с классом CancellationTokenSource. Перед вызовом асинхронного метода, использующего поток ThreadPoll в него надо передать CancellationToken в котором уже запрограммирована отмена выполняющегося в другом потоке метода:

```
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);
    }
}
```

Вызов метода CancelAfter() приведет к тому, что запущенный в отдельном потоке SlowMethod1(), будет остановлен через указанный период времени. Соответствующим образом надо изменить и два других метода:

```
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 и await для ADO.NET

Теперь рассмотрим async и await применительно к работе с поставщиками данных. В .NET Framework 4.5 добавлен целый ряд асинхронных методов для использования в ADO.NET. Вот самые востребованные из этих методов:

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

Более полный перечень новых асинхронних методов можно посмотреть на странице: https://msdn.microsoft.com/en-us/library/hh211418(v=vs.110).aspx.

Давайте рассмотрим, как выглядит применение этих методов на практике. Запустите Visual Studio 2015 и создайте новый проект с именем LibraryTest4. Мы продолжим работу с нашей БД Library, но теперь будем обращаться к ней с использованием async и await. Давайте, к тому же, еще используем в этом проекте класс DbProviderFactory. Это не обязательно, но для повторения рассмотренного материала будет полезно. Этот проект будет выполнять уже знакомые вам действия. Он будет подключаться к указанной в строке подключения БД, и выполнять в ней запрос, введенный пользователем в главном окне приложения. Однако, сейчас наше приложение будет добавлять перед пользовательским запросом запрос «WAITFOR

DELAY ‘00:00:05’;», чтобы имитировать длительную обработку и дать нам возможность убедиться в отсутствии блокировки главного потока.

Добавляем в конфигурационный файл строку подключения к нашей БД. Добавляем в проекте ссылку на пространство имен System.Configuration и приводим код приложения к такому виду:

```
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());
        }
        //выводим результаты запроса
        dataGridView1.DataSource = null;
        dataGridView1.DataSource = table;
    }

    ///<summary>
    ///При загрузке окна выбираем фабрику для
    ///поставщика System.Data.SqlClient вызываем
    ///метод для получения строки подключения
    ///из конфигурациооного файла
```

```
        ///</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("В конфигурационном
                                 файле нет требуемой
                                 строки подключения");
             }
        }
        ///<summary>
        ///Этот метод по имени поставщика данных
        ///считывает из конфигурационного файла
        ///и возвращает строку подключения, если
        ///эта строка есть в конфигурационном файле :)
        ///</summary>
        ///<param name="providerName"></param>
        ///<returns></returns>
        static string GetConnectionStringByProvider(
                                    string providerName)
        {
              string returnValue = null;

        //читаем все строки подключения из App.config
              ConnectionStringSettingsCollection
              settings = ConfigurationManager.
              ConnectionStrings;

              //ищем и возвращаем строку подключения
              //для providerName
```

```
            if (settings != null)
            {
                foreach (ConnectionStringSettings cs
                          in settings)
            {
                if (cs.ProviderName == providerName)
                {
                   returnValue = cs.ConnectionString;
                   break;
                }
            }
        }
        return returnValue;
    }

    ///<summary>
    ///управление доступностью кнопки
    ///</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;
    }
  }
}
```

Остановимся на строках кода, выделенных желтым цветом. В нашем обработчике кнопки присутствуют четыре вызова со спецификатором await. Поэтому, мы указали возле обработчика спецификатор async. Типом возвращаемого значения обработчика является void, поэтому

мы можем применить к обработчику спецификатор async. Если вспомнить, что каждый await вызов является для компилятора сигналом разделить метод (наш обработчик) на части, которые могут выполняться отдельно одна от другой, то мы понимаем, что наш обработчик будет разбит на пять частей.

Первой из выделенных строк является строка подключения к серверу БД:

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

В этом месте выполнение обработчика может приостановиться и управление будет возвращено главному потоку, т.е. пользователь сможет выполнять другие действия с главным окном приложения. Мы позаботились о том, чтобы во время выполнения запроса кнопка была заблокирована, и пользователь не мог вводить новые запросы до завершения обработки текущего. Кстати, наша процедура подключения к серверу БД вполне может выполниться и в синхронном режиме.

Дальше мы добавляем в объект DbCommand запрос, вызывающий остановку сервера на 5 секунд. Затем мы в режиме await вызываем метод ExecuteReaderAsync(), который должен выполнить все запросы, добавленные в свойство CommandText, создать объект DbDataReader, занести в этот объект результаты выполненных запросов и вернуть заполненный объект DbDataReader в вызывающий метод. В нашем проекте эта операция является самой ресурсозатратной и временезатратной. Особенно, если учесть наши 5 секунд ☺. Но, как вы можете убедиться,

запустив приложение, блокировки главного потока не происходит. Т.е., мы видим, что это сложное действие выполняется асинхронно.

Дальше у нас идет еще два await вызова, но они намного более легкие, чем выполнение ExecuteReaderAsync(). На приведенном ниже рисунке приведено окно нашего приложения после выполнения двух запросов, введенных пользователем и скрытого запроса задержки, добавляемого приложением. Пока эти запросы выполнялись, окно приложения отвечало на все действия — перемещения, изменения размера самого окна и колонок DataGridView и другие. Т.е. окно не было заблокированным.

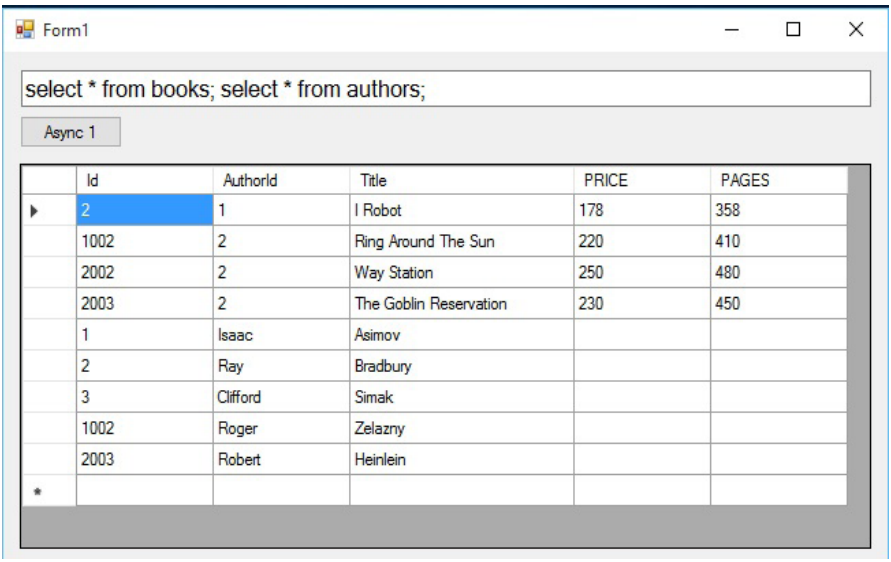

Рис. 1. Выполнение запросов с использованием async и await

Итак, приложение создано и выполняет то, что мы от него ожидаем. Но мы все это уже делали и видели. Чем же

это приложение отличается от предыдущего асинхронного приложения?

Это приложение отличается от предыдущего тем, что работает с поставщиками данных асинхронно, но не создает при этом дополнительных потоков! Следовательно, это приложение намного более эффективнее предыдущего многопоточного, уже просто потому, что создание дополнительных потоков очень усложняет приложение с точки зрения написания кода и с точки зрения затрат ресурсов и времени на выполнение.

Шифрование конфигурационного файла

Вы уже привыкли использовать в наших приложениях конфигурационный файл. Согласитесь, что это удобный способ для хранения строк подключения. В некоторых ситуациях, это единственный способ сделать так, чтобы сохранить возможность изменять реквизиты подключения к источникам данных без необходимости пересобирать приложение. Однако, вы, наверное, отметили для себя и недостаток такого подхода. Этот недостаток заключается в том, что реквизиты подключения к БД хранятся в текстовом формате в открытом виде, и, следовательно, могут быть доступны для просмотра.

Вопросы безопасности — это отдельная большая тема разговора. Мы сейчас затронем ее только частично. Прежде всего, вы, конечно же, должны держать в уме другие способы хранения и использования реквизитов подключения, отличные от конфигурационного файла. Если вы опасаетесь хранить реквизиты для подключения к серверу

в конфигурационном файле — не храните их там. Не храните эти реквизиты нигде. Просто создайте в приложении диалоговое окно, в которое пользователь должен будет заносить требуемые данные, при каждом подключении. Правда, в этом случае, пользователи должны будут знать эти реквизиты подключения. Я хочу сейчас предложить вам рассмотреть некий промежуточный способ соблюдения безопасности. Этот способ сводится к шифрованию строки подключения, хранящейся в конфигурационном файле. Т.е., пользователь знать реквизиты подключения не должен, и сами эти реквизиты подсмотреть в конфигурационном файле нельзя. В некоторых случаях такое решение будет оптимальным.

Чтобы реализовать такой подход надо воспользоваться классом RSAProtectedConfigurationProvider или DPAPIProtectedConfigurationProvider. Каждый из этих классов использует свой собственный способ для шифрования указанного раздела конфигурационного файла. В остальном они работают одинаково. Считывают раздел конфигурационного файла ConnectionStringsSection, в котором хранится строка подключения, шифруют его содержимое, используя свой алгоритм, и записывают обратно в файл, в зашифрованном виде. Всякий раз при чтении этого раздела он автоматически расшифровывается и приложение получает доступ к указанному в строке подключения поставщику данных. Давайте проверим, как этот подход работает на практике. Создадим новое Windows Forms приложение. В окне приложения расположим два текстовых поля, в которых будем отображать строку

подключения из конфигурационного файла на разных этапах выполнения приложения. Затем, для того чтобы убедиться, что шифрование раздела в конфигурационном файле не нарушает доступ к БД, расположим в окне кнопку и DataGridView, с помощью которых выполним тестовое подключение и отобразим результаты выполнения тестового запроса. Добавим в конфигурационный файл приложения строку подключения к нашей БД:

```
<?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>
```

Обратите внимание, что имя строки подключения — MyLibrary. Мы будем обращаться к этому имени в коде приложения.

Приведите код созданного проекта к такому виду:

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

```
//не шифрованная строка подключения
       string connStringNotEncrypted = "";
//шифрованная строка подключения
      string connStringWithEncryption = "";
      public Form1()
      {
          InitializeComponent();
      //читаем строку подключения до шифрования
          connStringNotEncrypted =
          ConfigurationManager.
          ConnectionStrings["MyLibrary"].
          ConnectionString;

      //отображаем строку подключения до шифрования
          textBox1.Text = connStringNotEncrypted;

      //выполняем шифрование
          EncryptConnSettings("connectionStrings");

      //читаем строку подключения после шифрования
          connStringWithEncryption =
          ConfigurationManager.ConnectionStrings
          ["MyLibrary"].ConnectionString;
          //отображаем строку подключения после шифрования
          textBox2.Text = connStringWithEncryption;
      }

      private static void
                  EncryptConnSettings(string section)
      {
      //создаем объект нашего конфигурационного файла
      //AppConfigDecrypt.exe – это имя выполняемого
      //файла вашего приложения если у вас оно
      //другое, то учтите этот момент
         Configuration objConfig = ConfigurationManager.
         OpenExeConfiguration(GetAppPath() +
         "AppConfigDecrypt.exe");
```

```
            //получаем доступ к разделу ConnectionStrings
            //нашего конфигурационного файла
            ConnectionStringsSection conSringSection
            = (ConnectionStringsSection)objConfig.
            GetSection(section);
            //если раздел не зашифрован – шифруем его
            if (!conSringSection.SectionInformation.
            IsProtected)
            {
               conSringSection.SectionInformation.
                   ProtectSection(
                   "RsaProtectedConfigurationProvider");
               conSringSection.SectionInformation.
                   ForceSave = true;
               objConfig.Save(ConfigurationSaveMode.
                   Modified);
            }
         }

         //получаем путь к папке, в которой лежит
         //конфигурационный файл
         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 = "";
           //повторно читаем строку подключения
           //из зашифрованной секции

           connString = ConfigurationManager.
               ConnectionStrings["MyLibrary"].
               ConnectionString;
           //повторно выводим строку подключения
           //в окно приложения
           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);
           }
       }
    }
}
```

Запустите приложение и нажмите кнопку Connect. Вы должны будете увидеть окно, подобное приведенному ниже. Давайте поговорим о том, что происходит в этом приложении.

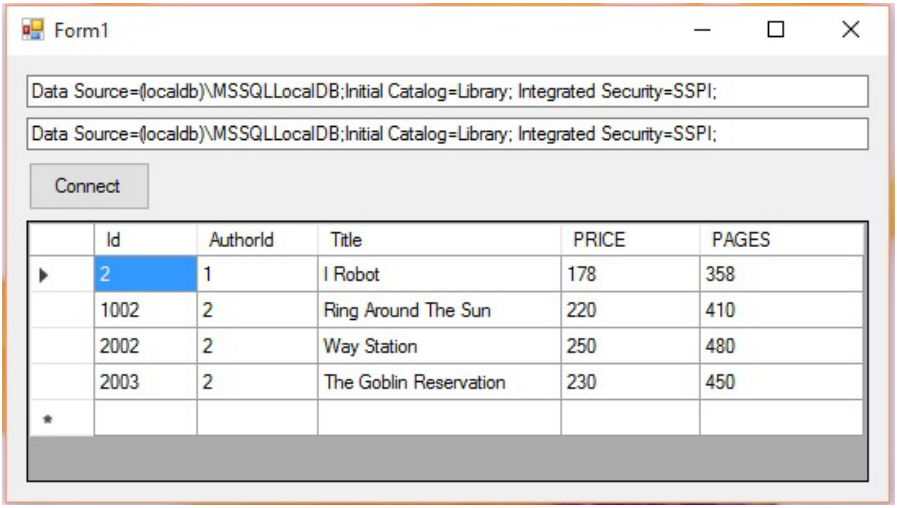

Рис. 2. Шифрование конфигурационного файла

Сразу после запуска приложения, еще до нажатия на кнопку Connect, вы видите в обоих текстовых полях нормальную строку подключения к нашей БД. Со строкой в первом текстовом поле все понятно. Мы выводим ее туда еще до шифрования, поэтому видим в нормальном, незашифрованном виде. Но во второе текстовое поле, мы выводим строку подключения, прочитанную из конфигурационного файла, после вызова метода EncryptConnString(), в котором выполняется шифрование. Почему же в этом случае мы видим строку подключения снова в открытом виде? Может приложение не зашифровало строку подключения. Перейдите в папку, где расположен .exe файл вашего приложения, конфигурационный файл с расширением .exe.config. Откройте блокнотом этот конфигурационный файл. У меня он выглядит так:

```
<?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>
```

Как видите, строка подключения зашифрована и никакие конфиденциальные данные не доступны для просмотра. Просто, при чтении зашифрованной строки подключения в приложении, она автоматически расшифровывается. Чтобы дальше можно было использовать ее по назначению.

То, что вы видите в окне результат выполнения тестового запроса, подтверждает, что строка подключения полностью функциональна. Но, зашифрована.

Необходимо учитывать, что такой способ шифрования конфигурационного файла машинозависим. Другими словами, если вы попробуете файл конфигурации, зашифрованный на одном компьютере, использовать на каком-нибудь другом компьютере, то у вас ничего не получится. Шифрование должно выполняться на том компьютере, где предполагается использование конфигурационного файла.

Домашнее задание

В качестве домашнего задания вам надо доработать приложение LibraryTest4 из этого урока. На главное окно приложения надо добавить комбобокс и метку для отображения небольших числовых значений. Логику работы приложения надо изменить таким образом, чтобы при выполнении запроса «select * from books» результаты выполнения этого запроса, как и раньше, отображались бы в DataGridView. При этом одновременно должен заполняться комбобокс, куда должны заноситься имена авторов, книги которых есть в таблице Books. При выборе в комбобоксе какого-либо автора, в добавленной метке должно отображаться количество книг выбранного автора в таблице Books.

Комбобокс должен заполняться только в том случае, когда пользователь ввел в окно единственный запрос «select * from books». Без уточнения полей, без каких-либо условий, без других запросов. При вводе любых других запросов, комбобокс должен оставаться пустым.

Заполнение комбобокса именами авторов и заполнение метки количеством книг должны выполняться асинхронно с использованием async и await.