Урок №3

Содержание

1. Фабрика провайдеров ADO.NET

Фабричный конструктивный шаблон

В основе модели программирования для написания не зависящего от поставщиков кода лежит использование «фабричного» конструктивного шаблона, в котором используется один API-интерфейс для доступа к базам данных нескольких поставщиков. Этому шаблону присваивается соответствующее имя, поскольку он задает использование специализированного объекта, только чтобы создавать другие объекты, что очень напоминает настоящую фабрику.

В **ADO.NET 2.0** в пространстве имен **System.Data.Common** появились новые базовые классы. Эти базовые классы являются абстрактными. К ним относятся классы **DbConnection**, **DbCommand** и **DbDataAdapter**; они совместно используются такими поставщиками данных платформы **.NET Framework**, как **System.Data.SqlClient** и **System.Data.OleDb**. Добавление базовых классов упрощает создание новых функций для поставщиков данных **.NET Framework** без необходимости в создании новых

интерфейсов. В **ADO.NET 2.0** также появились абстрактные базовые классы, с помощью которых разработчик может создавать обобщенный код доступа к данным, не зависящий от конкретного поставщика данных.

Классы DbProviderFactories и DbProviderFactory

Начиная с **ADO.NET 2.0**, для создания экземпляра **DbProviderFactory** класс **DbProviderFactories** предоставляет методы **static**. После этого экземпляр возвращает правильный строго типизированный объект, основанный на данных поставщика и строке соединения, предоставленной во время выполнения.

Имя	**Описание**
GetFactory	Перегружен. Возвращает экземпляр класса DbProviderFactory.
GetFactoryClasses	Возвращает объект DataTable, содержащий сведения обо всех установленных поставщиках, реализующих объект DbProviderFactory.

Клас **DbProviderFactory** представляет набор методов для создания экземпляров классов поставщиков, реализующих источник данных

Тип **DbProviderFactory** предоставляет следующие члены:

Методы

Имя	**Описание**
CreateCommand	Возвращает новый экземпляра класса поставщика, реализующий класс DbCommand.
CreateCommandBuilder	Возвращает новый экземпляра класса поставщика, реализующий класс DbCommandBuilder.
CreateConnection	Возвращает новый экземпляра класса поставщика, реализующий класс DbConnection.
CreateConnectionStringBuilder	Возвращает новый экземпляра класса поставщика, реализующий класс DbConnectionStringBuilder.
CreateDataAdapter	Возвращает новый экземпляра класса поставщика, реализующий класс DbDataAdapter.
CreateDataSourceEnumerator	Возвращает новый экземпляра класса поставщика, реализующий класс DbDataSourceEnumerator.
CreateParameter	Возвращает новый экземпляра класса поставщика, реализующий класс DbParameter.

CreatePermission	Возвращает новый экземпляра класса поставщика, реализующий версию поставщика класса CodeAccessPermission.

Иерархия наследования

- System.Object
  - **System.Data.Common.DbProviderFactory**
    - System.Data.Odbc.OdbcFactory
    - System.Data.OleDb.OleDbFactory
    - System.Data.OracleClient.OracleClientFactory
    - System.Data.SqlClient.SqlClientFactory

Получение и использование фабрики

Процесс получения **DbProviderFactory** состоит из передачи сведений о поставщике данных классу **DbProviderFactories**. На основе этих сведений метод **GetFactory** создает строго типизированную фабрику поставщика. Например, чтобы создать фабрику **SqlClientFactory**, можно передать методу **GetFactory** строку с именем поставщика, указанным в формате «**System.Data.SqlClient**». Другая перегрузка метода **GetFactory** принимает **DataRow**. После создания фабрики поставщика можно использовать ее методы для создания дополнительных объектов. К методам фабрики **SqlClientFactory** относятся **CreateConnection**, **CreateCommand** и **CreateDataAdapter**. Классы **.NET Framework OracleClientFactory**, **OdbcFactory** и **OleDbFactory** также предоставляют похожие возможности.

Каждый поставщик данных **.NET Framework**, который поддерживает фабричный класс, регистрирует сведения о конфигурации в разделе **DbProviderFactories** файла **machine.config** на локальном компьютере (О файлах конфигурации будет рассказано в третьей части урока). В следующем фрагменте файла конфигурации показан синтаксис и формат для **System.Data.SqlClient**.

```
<system.data>
  <DbProviderFactories>
    <add name="SqlClient Data Provider"
     invariant="System.Data.SqlClient"
     description=".Net Framework Data Provider for SqlServer"
     type="System.Data.SqlClient.SqlClientFactory, System.Data,
     Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
    />
  </DbProviderFactories>
</system.data>
```

Атрибут **invariant** определяет базовый поставщик данных. Этот трехкомпонентный синтаксис имени также применяется при создании новой фабрики и для определения поставщика в файле конфигурации, чтобы имя поставщика вместе со связанной с ним строкой соединения можно было получать во время выполнения.

Сведения обо всех поставщиках данных, установленных на компьютере, можно получить с помощью метода **GetFactoryClasses**. Он возвращает таблицу **DataTable** с именем **DbProviderFactories**, в которой содержатся столбцы, описанные в следующей таблице.

№	Имя столбца	Пример результата	Описание
0	**Name**	Поставщик данных SqlClient	Понятное имя поставщика данных.
1	**Description**	Поставщик данных .NET Framework для SqlServer	Понятное описание поставщика данных.
2	**InvariantName**	System.Data.SqlClient	Имя, которое можно использовать программно, чтобы ссылаться на поставщик данных.
3	**AssemblyQualified Name**	System.Data.SqlClient. SqlClientFactory, System.Data, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a 5c561934e089	Полное имя фабричного класса, которое содержит достаточно данных для создания экземпляров объектов.

Эту таблицу **DataTable** можно применять, чтобы дать пользователям возможность выбирать **DataRow** во время выполнения. Выбранная строка **DataRow** затем может передаваться методу **GetFactory** для создания строго типизированной фабрики **DbProviderFactory**. Выбранная строка **DataRow** затем может передаваться методу **GetFactory** для создания нужного объекта **DbProviderFactory**.

Продемонстрируем, как использовать метод **GetFactoryClasses** для возвращения таблицы **DataTable** со сведениями об установленных поставщиках. В коде выполняется просмотр каждой строки в таблице **DataTable** с выводом сведений по каждому поставщику в консольном окне.

```
static DataTable GetProviderFactoryClasses()
        {
            DataTable table = DbProviderFactories.GetFactoryClasses();

            foreach (DataRow row in table.Rows)
```

```
            {
                foreach (DataColumn column in table.Columns)
                {
                    Console.WriteLine(row[column]);
                }
            }

            return table;
        }
```

В этом примере демонстрируется, как создать фабрику **DbProviderFactory** и объект **DbConnection** путем передачи имени поставщика в формате «**System.Data.ProviderName**» и строки соединения. В случае успеха возвращается объект **DbConnection**; в случае любой ошибки — **null**.

Этот код получает **DbProviderFactory** путем вызова **GetFactory**. Затем метод **CreateConnection** создает объект **DbConnection**, а свойству **ConnectionString** присваивается значение строки соединения.

Если объект **DbConnection** является допустимым, то открывается соединение, создается и выполняется команда **DbCommand**. **CommandText** задается инструкции **SQL UPDATE**, которая выполняет вставку в таблицу в базе данных. Предполагается, что база данных существует в источнике данных, а также что используемый в инструкции **UPDATE** синтаксис **SQL** является допустимым для указанного поставщика. Ошибки в источнике данных обрабатываются блоком кода **DbException**, а все остальные исключения — в блоке **Exception**.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data;
using System.Data.Common;

namespace CreateDbConnectionExample
{
    class Program
    {
        static void Main(string[] args)
        {
            DbConnection connection =
                CreateDbConnection("System.Data.SqlClient",
```

```
                                "Data Source=(local);" +
                                "Integrated Security=SSPI;" +
                                "Initial Catalog=AdventureWorks;"
                                );

            ExecuteDbCommand(connection);

            Console.ReadKey(true);
        }

        static void ExecuteDbCommand(DbConnection connection)
        {
            if (connection != null)
            {
                using (connection)
                {
                    try
                    {
                        connection.Open();

                        DbCommand command = connection.CreateCommand();
                        command.CommandText =
                            "UPDATE Production.Product " +
                            "SET ReorderPoint = ReorderPoint + 1 " +
                            "WHERE ReorderPoint Is Not Null;" +
                            "UPDATE Production.Product " +
                            "SET ReorderPoint = ReorderPoint - 1 " +
                            "WHERE ReorderPoint Is Not Null";

                        int rows = command.ExecuteNonQuery();

                        Console.WriteLine("Updated {0} rows.", rows);
                    }

                    catch (DbException exDb)
                    {
                        Console.WriteLine("DbException.GetType: {0}",
```

```
                                        exDb.GetType());
                    Console.WriteLine("DbException.Source: {0}",
                                        exDb.Source);
                    Console.WriteLine("DbException.ErrorCode: {0}",
                                        exDb.ErrorCode);
                    Console.WriteLine("DbException.Message: {0}",
                                        exDb.Message);
                }
                catch (Exception ex)
                {
                    Console.WriteLine("Exception.Message: {0}",
                                         ex.Message);
                }
            }
        }
        else
        {
            Console.WriteLine("Failed: DbConnection is null.");
        }
    }

    static DbConnection CreateDbConnection(
        string providerName, string connectionString)
    {
        DbConnection connection = null;

        if (connectionString != null)
        {
            try
            {
                DbProviderFactory factory =
                    DbProviderFactories.GetFactory(providerName);

                connection = factory.CreateConnection();
                connection.ConnectionString = connectionString;
            }
            catch (Exception ex)
```

```
                {
                    if (connection != null)
                    {
                        connection = null;
                    }
                    Console.WriteLine(ex.Message);
                }
            }

            return connection;
        }
    }
}
```

2. Асинхронные механизмы доступа к данным

Асинхронность. Необходимость использования асинхронного механизма

Для завершения некоторых операций базы данных, например для выполнения команд, требуется значительное время. В таком случае однопотоковые приложения должны блокировать другие операции и ждать окончания команды перед тем, как они смогут продолжить свои собственные операции. В противоположность этому, так как поток переднего плана может назначать долговременные операции фоновому потоку, он остается активным во время всей операции. Приложение Windows, например, делегирующее долговременную операцию фоновому потоку, позволяет потоку пользовательского интерфейса реагировать при выполнении операции.

Асинхронное программирование в .NET

Делегаты позволяют вызывать синхронные методы асинхронно. При синхронном вызове делегата метод **Invoke** вызывает метод назначения непосредственно в текущий поток. При вызове метода **BeginInvoke** среда **CLR** помещает запрос в очередь и сразу же возвращает управление вызывающему объекту. Метод назначения асинхронно вызывается в поток из пула потоков. Исходный поток, отправивший этот запрос, теперь может беспрепятственно продолжать работу параллельно с методом назначения. Если при вызове метода **BeginInvoke** был задан метод обратного вызова, он будет вызван при завершении выполнения метода назначения. В методе обратного вызова метод **EndInvoke** получает возвращаемое значение и любые входные и выходные параметры или только выходные параметры. Если метод обратного вызова не указан при вызове **BeginInvoke**, **EndInvoke** может быть вызван из потока, который вызвал **BeginInvoke**.

В **.NET Framework** можно асинхронно вызывать любой синхронный метод. Для этого необходимо определить делегат с той же подписью, что и у вызываемого метода. Среда **CLR** автоматически определяет для этого делегата методы **BeginInvoke** и **EndInvoke** с соответствующими подписями.

Асинхронный вызов инициируется с помощью метода **BeginInvoke**. Он имеет те же параметры, что и метод, который нужно выполнить асинхронно, а также два дополнительных параметра. Первый параметр является делегатом **AsyncCallback**, который ссылается на метод, вызываемый при завершении асинхронного вызова. Второй параметр — это определяемый пользователем объект, который передает информацию в метод обратного вызова. Метод **BeginInvoke** завершает работу моментально, не дожидаясь завершения асинхронного вызова. Метод **BeginInvoke** возвращает объект **IAsyncResult**, который можно использовать для отслеживания хода выполнения асинхронного вызова.

Метод **EndInvoke** извлекает результаты асинхронного вызова. Его можно вызвать в любое время после вызова метода **BeginInvoke**. Если асинхронный вызов не завершен, метод **EndInvoke** блокирует вызывающий поток до завершения вызова. Список параметров метода **EndInvoke** включает параметры **out** и **ref** метода, который требуется вызвать асинхронно, а также значение **IAsyncResult**, возвращаемое методом **BeginInvoke**.

Существуют четыре основных способа использования методов **BeginInvoke** и **EndInvoke** для выполнения асинхронных вызовов. После вызова метода **BeginInvoke** можно делать следующее:

1) Выполнить какие-либо операции, а затем вызвать метод **EndInvoke** для блокировки потока до тех пор, пока вызов не завершится.
2) Получить объект **WaitHandle** с помощью свойства IAsyncResult.**AsyncWaitHandle**, использовать метод **WaitOne** для блокирования выполнения до получения сигнала **WaitHandle**, а затем вызвать метод **EndInvoke**.
3) Периодически опрашивать интерфейс **IAsyncResult**, возвращаемый методом **BeginInvoke**, для определения момента завершения асинхронного вызова, и затем вызвать метод **EndInvoke**.
4) Передать в метод **BeginInvoke** делегат для метода обратного вызова. Этот метод выполняется для потока **ThreadPool** после выполнения асинхронного вызова. Метод обратного вызова вызывает метод **EndInvoke**.

Независимо от выбранного варианта необходимо всегда использовать для завершения асинхронного вызова метод **EndInvoke**.

Рассмотрим различные способы асинхронного вызова одного и того же долгосрочного метода **TestMethod**. Метод **TestMethod** выводит сообщение консоли для отображения начала обработки, ждет несколько секунд, после чего завершается. У метода **TestMethod** имеется параметр **out**, демонстрирующий способ добавления таких параметров в подписи методов **BeginInvoke** и **EndInvoke**.

В следующем примере показано определение **TestMethod** и делегата **AsyncMethodCaller**, который может использоваться для асинхронного вызова **TestMethod**.

```
public class AsyncDemo
    {
        public string TestMethod(int callDuration, out int threadId)
        {
            Console.WriteLine("Test method begins.");
            Thread.Sleep(callDuration);
            threadId = Thread.CurrentThread.ManagedThreadId;
            return String.Format("My call time was {0}.",
                                 callDuration.ToString());
        }
    }

public delegate string AsyncMethodCaller(int callDuration, out int threadId);
```

Самым простым способом асинхронного вызова метода является запуск метода посредством вызова метода **BeginInvoke** делегата, выполнения каких-либо действий с

основным потоком и последующего вызова метода **EndInvoke** делегата. Метод **EndInvoke** может блокировать вызывающий поток, поскольку он не возвращается до завершения асинхронного вызова. Этот подход хорошо использовать с файловыми и сетевыми операциями.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;

namespace AsynchronousOperations
{
    public class AsyncMain
    {
        public static void Main()
        {
            int threadId;

            AsyncDemo ad = new AsyncDemo();

            AsyncMethodCaller caller = new AsyncMethodCaller(ad.TestMethod);

            IAsyncResult result = caller.BeginInvoke(20000,
                                                     out threadId,
                                                     null, null);

            Thread.Sleep(0);
            Console.WriteLine("Main thread {0} does some work.",
                              Thread.CurrentThread.ManagedThreadId);

            string returnValue = caller.EndInvoke(out threadId, result);

            Console.WriteLine("The call executed on thread {0}, with return
                              value \"{1}\".",
                              threadId, returnValue);

            Console.ReadKey(true);
```

```
        }
    }
…
}
```

Рассмотрим асинхронный вызов с использованием дескриптора ожидания **WaitHandle**. Объект **WaitHandle** можно получить с помощью свойства **AsyncWaitHandle** объекта **IAsyncResult**, возвращаемого методом **BeginInvoke**. Объект **WaitHandle** получает сигнал при завершении асинхронного вызова; его можно дождаться путем вызова метода **WaitOne**. При использовании объекта **WaitHandle** можно выполнять дополнительные операции до или после завершения асинхронного вызова, но до вызова метода **EndInvoke** для получения результатов. Отметим, что при вызове метода **EndInvoke** дескриптор ожидания не закрывается автоматически. Чтобы освободить системные ресурсы сразу после завершения использования дескриптора ожидания, необходимо удалить его с помощью метода **WaitHandle.Close**.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;

namespace AsynchronousOperations
{
    public class AsyncMain
    {

        static void Main()
        {
            int threadId;

            AsyncDemo ad = new AsyncDemo();
            AsyncMethodCaller caller = new AsyncMethodCaller(ad.TestMethod);

            IAsyncResult result = caller.BeginInvoke(3000, out threadId,
                                                     null, null);

            Thread.Sleep(0);
            Console.WriteLine("Main thread {0} does some work.",
                Thread.CurrentThread.ManagedThreadId);
```

```
            result.AsyncWaitHandle.WaitOne();

            string returnValue = caller.EndInvoke(out threadId, result);

            result.AsyncWaitHandle.Close();

            Console.WriteLine("The call executed on thread {0},
                               with return value \"{1}\".",
                               threadId, returnValue);
        }
    }
…
}
```

Свойство **IsCompleted** объекта **IAsyncResult**, возвращаемого методом **BeginInvoke**, также можно использовать для отслеживания завершения асинхронного метода. Это можно делать, когда асинхронный вызов произведен из потока, обслуживающего пользовательский интерфейс. Опрос завершения позволяет вызывающему потоку продолжить выполнение при асинхронном вызове потока **ThreadPool**.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;

namespace AsynchronousOperations
{
    public class AsyncMain
    {
        static void Main()
        {
            int threadId;

            AsyncDemo ad = new AsyncDemo();

            AsyncMethodCaller caller = new AsyncMethodCaller(ad.TestMethod);
```

```
            IAsyncResult result = caller.BeginInvoke(3000,
                out threadId, null, null);

            while (result.IsCompleted == false)
            {
                Thread.Sleep(250);
                Console.Write(".");
            }

            string returnValue = caller.EndInvoke(out threadId, result);

            Console.WriteLine("\nThe call executed on thread {0},
                               with return value \"{1}\".",
                               threadId, returnValue);

            Console.ReadKey(true);
        }
    }
…
}
```

Если поток, инициировавший асинхронный вызов, не обязательно должен быть потоком, обрабатывающим результаты вызова, после завершения асинхронного вызова можно выполнить метод обратного вызова. Метод обратного вызова выполняется для потока **ThreadPool**.

Чтобы использовать метод обратного вызова, необходимо передать в метод **BeginInvoke** делегат **AsyncCallback**, который ссылается на метод обратного вызова. Кроме того, можно передать объект, содержащий данные, которые будут использоваться методом обратного вызова. В методе обратного вызова параметр **IAsyncResult**, который является единственным параметром метода обратного вызова, можно привести к типу объекта **AsyncResult**. После этого свойство **AsyncResult**.**AsyncDelegate** можно будет использовать для получения делегата, с помощью которого инициирован вызов, чтобы можно было вызвать метод **EndInvoke**.

Параметр **threadId** метода **TestMethod** является параметром **out**, поэтому входные значения никогда не используются методом **TestMethod**. При вызове метода **BeginInvoke** ему передается фиктивный параметр. Если параметр **threadId** является параметром **ref**, переменная должна быть полем уровня класса, чтобы ее можно было передавать методам **BeginInvoke** и **EndInvoke**.

Сведения о состоянии передаются методу **BeginInvoke** в виде строки форматирования, используемую методом обратного вызова для форматирования выходного сообщения.

Поскольку сведения о состоянии передаются в виде типа **Object**, перед использованием их необходимо привести к соответствующему типу.

Обратный вызов осуществляется в потоке **ThreadPool**. Потоки **ThreadPool** — это фоновые потоки, которые не поддерживают работу приложения, если завершается основной поток, поэтому основной поток в примере должен дождаться завершения обратного вызова.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;

namespace AsynchronousOperations
{
    public class AsyncMain
    {
        static void Main()
        {
            AsyncDemo ad = new AsyncDemo();

            AsyncMethodCaller caller = new AsyncMethodCaller(ad.TestMethod);

            int dummy = 0;

            IAsyncResult result = caller.BeginInvoke(3000, out dummy,
                                    new AsyncCallback(CallbackMethod),
                                    "The call executed on thread {0},
                                     with return value \"{1}\".");

            Console.WriteLine("The main thread {0} continues to execute...",
                              Thread.CurrentThread.ManagedThreadId);

            Thread.Sleep(4000);

            Console.WriteLine("The main thread ends.");

            Console.ReadKey(true);
```

```
        }

        static void CallbackMethod(IAsyncResult ar)
        {
            AsyncResult result = (AsyncResult)ar;
            AsyncMethodCaller caller=(AsyncMethodCaller)result.AsyncDelegate;

            string formatString = (string)ar.AsyncState;

            int threadId = 0;
            string returnValue = caller.EndInvoke(out threadId, ar);

            Console.WriteLine(formatString, threadId, returnValue);
        }
    }
…
}
```

Асинхронные механизмы доступа к данным

.NET Framework предоставляет несколько асинхронных шаблонов конструирования, которые разработчики могут использовать для извлечения преимуществ из фоновых потоков и освобождает пользовательский интерфейс высокоприоритетных потоков от выполнения других операций. **ADO.NET** поддерживает те же шаблоны конструирования в своем классе **SqlCommand**, а именно, методы **BeginExecuteNonQuery**, **BeginExecuteReader** и **BeginExecuteXmlReader** в сочетании с методами **EndExecuteNonQuery**, **EndExecuteReader** и **EndExecuteXmlReader** предоставляют асинхронную поддержку.

Асинхронное программирование является основной функцией **.NET Framework**, а **ADO.NET** пользуется всеми преимуществами стандартных шаблонов конструирования. Хотя использование асинхронной техники при помощи функций **ADO.NET** не требует какого-либо дополнительного специального рассмотрения, большинство разработчиков использует асинхронные функции в **ADO.NET**, а не в других областях **.NET Framework**. Важно знать о преимуществах и проблемах создания многопотоковых приложений. Приводимые примеры указывают на несколько важных проблем, которые должны учитывать разработчики при построении приложений, включающих функциональные возможности многопотоковой техники.

Приведем пример Windows-приложения, использующего ответные вызовы. В большинстве сценариев асинхронной обработки необходимо запустить операцию в базе данных и продолжить выполнение других процессов, не дожидаясь завершения данной операции. Однако во многих случаях требуется выполнение каких-либо действий после завершения операции в базе данных. Например, в Windows-приложении может

потребоваться передать длительную операцию в фоновый поток, чтобы иметь возможность работать с пользовательским интерфейсом. Однако по завершении операции в базе данных необходимо будет использовать результаты для заполнения формы. В таких случаях лучше всего использовать ответный вызов.

Ответный вызов определяется путем задания делегата **AsyncCallback** в методе **BeginExecuteNonQuery**, **BeginExecuteReader** или **BeginExecuteXmlReader**. Этот делегат вызывается по завершении операции. Делегату можно передать ссылку на саму команду **SqlCommand**, чтобы облегчить доступ к объекту **SqlCommand** и вызов соответствующего метода **End** без необходимости использовать глобальную переменную.

Продемонстрируем использование метода **BeginExecuteNonQuery**, выполняющего инструкцию **Transact-SQL**, которая содержит задержку в несколько секунд, имитируя длительную команду. В примере показаны несколько важных приемов, в том числе вызов метода, который взаимодействует с формой из отдельного потока. Кроме того, в этом примере показано, как блокировать попытки пользователей одновременно выполнять команду несколько раз и как гарантировать, что форма не закроется до вызова процедуры ответного вызова. Для моделирования длительного процесса в примере в текст команды вставлена инструкция **WAITFOR**. Обычно замедление выполнения команд не производится, однако в данном случае это упрощает демонстрацию асинхронного поведения.

Создадим новое Windows-приложение. Поместим в форму элемент управления **Button** и два элемента управления **Label**. Подключим соответствующие обработчики событий, и добавим в класс формы следующий код:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

using System.Data.SqlClient;

namespace BeginExecuteNonQueryExample
{
    public partial class Form1: Form
    {
        public Form1()
        {
```

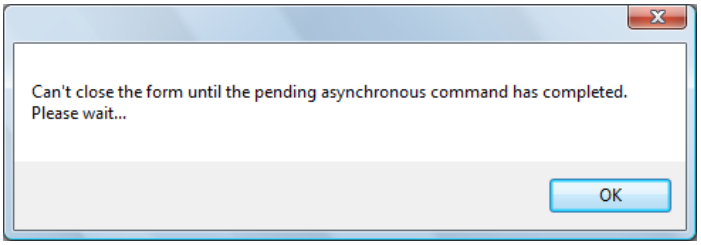
```
            InitializeComponent();
        }

        private delegate void DisplayInfoDelegate(string Text);

        private bool isExecuting;

        private SqlConnection connection;

        private static string GetConnectionString()
        {
            return "Data Source=(local);Integrated Security=SSPI;" +
                   "Initial Catalog=Works;
                    Asynchronous Processing=true";
        }

        private void DisplayStatus(string Text)
        {
            this.label1.Text = Text;
        }

        private void DisplayResults(string Text)
        {
            this.label2.Text = Text;
            DisplayStatus("Ready");
        }

        private void Form1_FormClosing(object sender,
System.Windows.Forms.FormClosingEventArgs e)
        {
            if (isExecuting)
            {
                MessageBox.Show(this, "Can't close the form until " +
                "the pending asynchronous command has completed. Please " +
                "wait...");
                e.Cancel = true;
            }
```

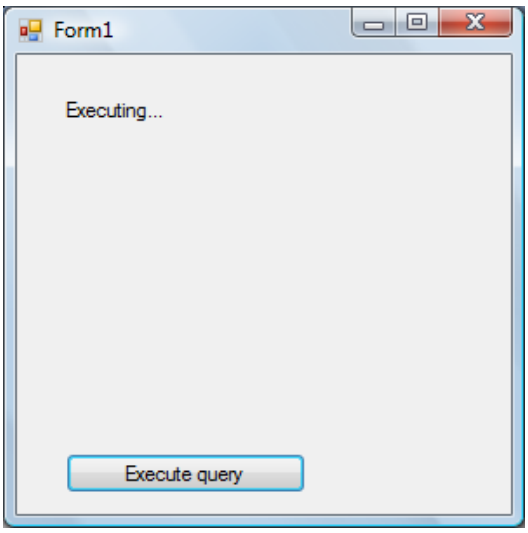
```
        }

        private void button1_Click(object sender, System.EventArgs e)
        {
            if (isExecuting)
            {
              MessageBox.Show(this, "Already executing. Please wait until " +
               "the current query has completed.");
            }
            else
            {
                SqlCommand command = null;
                try
                {
                    DisplayResults("");
                    DisplayStatus("Connecting...");
                    connection = new SqlConnection(GetConnectionString());

                    string commandText =
                        "WAITFOR DELAY '0:0:05';" +
                        "UPDATE Production.Product " +
                        "SET ReorderPoint = ReorderPoint + 1 " +
                        "WHERE ReorderPoint Is Not Null;" +
                        "UPDATE Production.Product " +
                        "SET ReorderPoint = ReorderPoint - 1 " +
                        "WHERE ReorderPoint Is Not Null";

                    command = new SqlCommand(commandText, connection);
                    connection.Open();

                    DisplayStatus("Executing...");
                    isExecuting = true;

                    AsyncCallback callback=new AsyncCallback(HandleCallback);

                    command.BeginExecuteNonQuery(callback, command);
                }
```

```
            catch (Exception ex)
            {
                isExecuting = false;
                DisplayStatus(string.Format("Ready (last error: {0})",
                              ex.Message));

                if (connection != null)
                {
                    connection.Close();
                }
            }
        }
    }

    private void HandleCallback(IAsyncResult result)
    {
        try
        {
            SqlCommand command = (SqlCommand)result.AsyncState;
            int rowCount = command.EndExecuteNonQuery(result);
            string rowText = " rows affected.";
            if (rowCount == 1)
            {
                rowText = " row affected.";
            }
            rowText = rowCount + rowText;

            DisplayInfoDelegate del =
             new DisplayInfoDelegate(DisplayResults);
            this.Invoke(del, rowText);
        }
        catch (Exception ex)
        {
            this.Invoke(new DisplayInfoDelegate(DisplayStatus),
            String.Format("Ready(last error: {0}", ex.Message));
        }
```

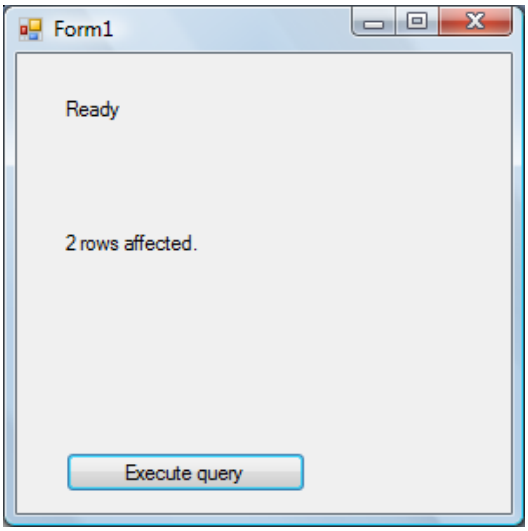
```
            finally
            {
                isExecuting = false;
                if (connection != null)
                {
                    connection.Close();
                }
            }
        }

        private void Form1_Load(object sender, System.EventArgs e)
        {
           this.button1.Click += new System.EventHandler(this.button1_Click);
           this.FormClosing += new System.Windows.Forms.
                               FormClosingEventHandler(this.Form1_FormClosing);
        }
    }
}
```

В примере производится подключение к базе **Works**. Производиться запрос на изменение записей в таблице **Product**. Результаты выводятся на форму. Блокируются попытки одновременно выполнять команду несколько раз. Форма не закрывается до вызова процедуры ответного вызова:

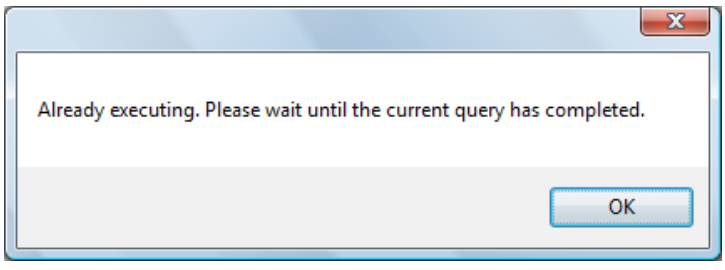




Выполнение запроса



Запрос выполнен – выводиться число измененных записей



Блокировка попытки одновременного выполнения нескольких команд на изменение

Перед запуском данного примера не забудьте убедиться в наличии необходимой базы и соответствующих таблиц на вашем локальном сервере, а также проверьте настройки соединения.

Асинхронные операции в **ADO.NET** позволяют запускать трудоемкие операции базы данных в одном потоке, а остальные задачи выполнять в другом потоке. Однако в

большинстве сценариев в конечном итоге достигается положение, в котором не следует продолжать работу приложения, пока не завершится операция базы данных. В таких случаях полезно опросить асинхронную операцию, чтобы определить, завершена ли эта операция. Для выяснения того, завершена ли операция, используем свойство **IsCompleted**.

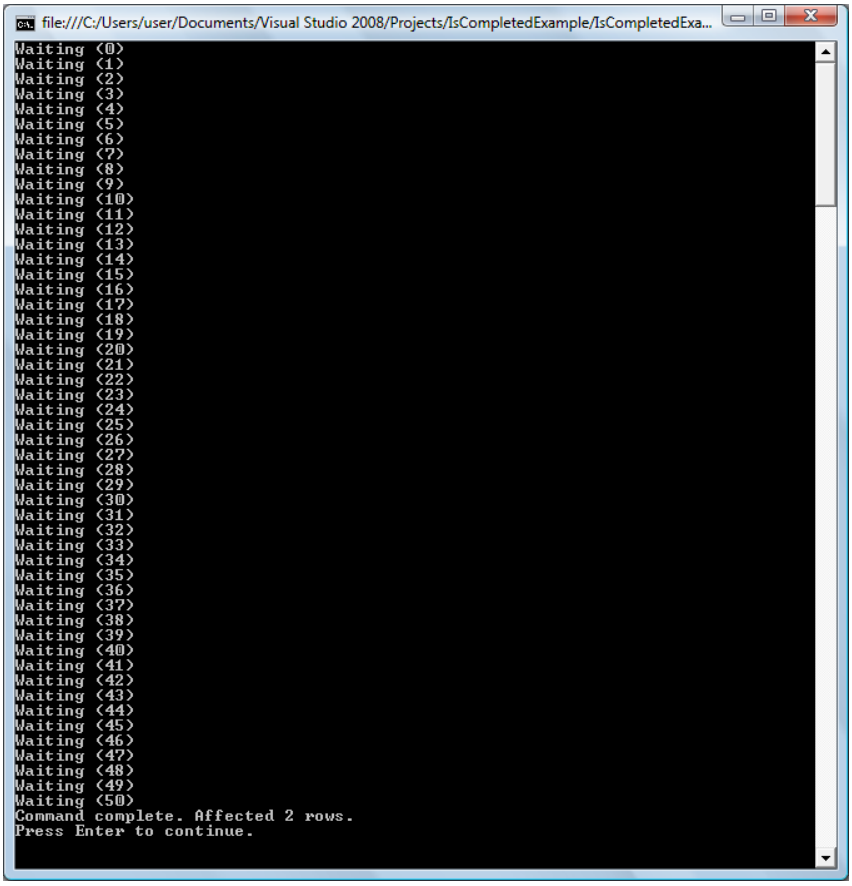
В следующем приложении командной строки происходит асинхронное обновление данных в образце базы данных **Works**. Для моделирования длительного процесса в примере в текст команды вставлена инструкция **WAITFOR**:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data;
using System.Data.SqlClient;


namespace IsCompletedExample
{
    class ADOAsyncDemo
    {
        [STAThread]
        static void Main()
        {
            string commandText =
              "UPDATE Production.Product SET ReorderPoint = " +
              "ReorderPoint + 1 " +
              "WHERE ReorderPoint Is Not Null;" +
              "WAITFOR DELAY '0:0:3';" +
              "UPDATE Production.Product SET ReorderPoint = " +
              "ReorderPoint - 1 " +
              "WHERE ReorderPoint Is Not Null";

            RunCommandAsynchronously(
                commandText, GetConnectionString());

            Console.WriteLine("Press Enter to continue.");
            Console.ReadLine();
        }
```

```
private static void RunCommandAsynchronously(
  string commandText, string connectionString)
{
    using(
      SqlConnection connection = new SqlConnection(connectionString)
        )
    {
        try
        {
            int count = 0;
            SqlCommand command =
                new SqlCommand(commandText, connection);
            connection.Open();

            IAsyncResult result =
                command.BeginExecuteNonQuery();
            while (!result.IsCompleted)
            {
                Console.WriteLine(
                                "Waiting ({0})", count++);

                System.Threading.Thread.Sleep(100);
            }
            Console.WriteLine(
                "Command complete. Affected {0} rows.",
            command.EndExecuteNonQuery(result));
        }
        catch (SqlException ex)
        {
            Console.WriteLine("Error ({0}): {1}",
                ex.Number, ex.Message);
        }
        catch (InvalidOperationException ex)
        {
            Console.WriteLine("Error: {0}", ex.Message);
        }
```

```
                catch (Exception ex)
                {
                    Console.WriteLine("Error: {0}", ex.Message);
                }
            }
        }

        private static string GetConnectionString()
        {
            return "Data Source=(local);Integrated Security=SSPI;" +
            "Initial Catalog=Works; " +
            "Asynchronous Processing=true";
        }
    }
}
```

В примере производится подключение к базе **Works**. Производиться запрос на изменение записей в таблице **Product**. Количество модифицированных записей выводятся на консоль. Перед запуском данного примера не забудьте убедиться в наличии необходимой базы и соответствующих таблиц на вашем локальном сервере, а также проверьте настройки соединения.


file:///C:/Users/user/Documents/Visual Studio 2008/Projects/IsCompletedExample/IsCompletedExa...
Waiting (0)
Waiting (1)
Waiting (2)
Waiting (3)
Waiting (4)
Waiting (5)
Waiting (6)
Waiting (7)
Waiting (8)
Waiting (9)
Waiting (10)
Waiting (11)
Waiting (12)
Waiting (13)
Waiting (14)
Waiting (15)
Waiting (16)
Waiting (17)
Waiting (18)
Waiting (19)
Waiting (20)
Waiting (21)
Waiting (22)
Waiting (23)
Waiting (24)
Waiting (25)
Waiting (26)
Waiting (27)
Waiting (28)
Waiting (29)
Waiting (30)
Waiting (31)
Waiting (32)
Waiting (33)
Waiting (34)
Waiting (35)
Waiting (36)
Waiting (37)
Waiting (38)
Waiting (39)
Waiting (40)
Waiting (41)
Waiting (42)
Waiting (43)
Waiting (44)
Waiting (45)
Waiting (46)
Waiting (47)
Waiting (48)
Waiting (49)
Waiting (50)
Command complete. Affected 2 rows.
Press Enter to continue.

3. Использование конфигурационных файлов

Цели и задачи конфигурационных файлов

Файл конфигурации или конфигурационный файл используется для хранения настроек компьютерных программ, в том числе и операционных систем. Как правило, конфигурационные файлы имеют текстовый формат и могут быть прочитаны и отредактированы пользователем программы.

Файлы конфигурации имеют формат **XML**, и при необходимости их можно изменять. Файлы конфигурации позволяют настраивать параметры приложения без перекомпиляции. Кроме того, используя файлы конфигурации, администраторы могут задавать политики, влияющие на выполнение приложений на определенных компьютерах. С помощью классов из пространства имен **System.Configuration** управляемый код может считывать установки из конфигурационных файлов, но не записывать их в эти файлы.

Типы конфигурационных файлов

Файлы конфигурации состоят из элементов, которые являются логическими структурами данных, задающими сведения о конфигурации. Начало и конец каждого элемента в файле конфигурации отмечены специальными тегами. Например, элемент **<runtime>** имеет структуру: **<runtime>***дочерние элементы***</runtime>**. Пустой элемент состоит из открывающего тега, закрывающий тег отсутствует. Параметры конфигурации задаются с помощью предварительно определенных атрибутов (пар имя-значение) в открывающем теге элемента. В следующем примере заданы два атрибута (**version** и **href**) элемента **<codeBase>**, определяющие для среды выполнения расположение сборок. В файлах конфигурации учитывается регистр.

```
<codeBase version="2.0.0.0"
          href="http://www.litwareinc.com/myAssembly.dll"/>
```

Существует три типа таких конфигурационных файлов: конфигурации компьютера, приложения и безопасности.

В файле конфигурации компьютера, **Machine.config**, задаются параметры, влияющие на работу компьютера в целом. Этот файл расположен в папке *%папка установки среды выполнения%\Config*. В файле **Machine.config** задаются параметры конфигурации для привязки сборок компьютера, встроенных каналов удаленного взаимодействия и **ASP.NET**. Поиск элемента Элемент **appSettings** (схема общих параметров) и других разделов конфигурации, определяемых разработчиками, производится системой конфигурации в первую очередь в файле конфигурации компьютера. Затем поиск выполняется в файле конфигурации приложения. Для облегчения управления файлом конфигурации компьютера рекомендуется переместить эти параметры в файл конфигурации приложения. Однако размещение параметров в файле конфигурации компьютера упрощает управление системой. Например, при наличии компонентов сторонних производителей, используемых сервером и клиентами, рекомендуется разместить параметры этих компонентов в одном файле. В этом случае следует задавать параметры в файле конфигурации компьютера, чтобы не дублировать параметры в разных файлах.

В файлах конфигурации приложений задаются параметры приложений. Параметры считываются из этих файлов средой **CLR** (политика привязки сборок, удаленные объекты и т. д.) и приложением. Имя и расположение файла конфигурации приложения зависят от хоста приложения, которым может быть один из следующих компонентов:

1) Исполняемый файл в роли хоста.

Файл конфигурации приложения, для которого в роли хоста выступает исполняемый файл, должен находиться в той же папке, где хранится приложение. Имя файла конфигурации — это имя приложения с расширением **CONFIG**. Например, файл конфигурации приложения **myApp.exe** должен называться **myApp.exe.config**.

2) ASP.NET в роли хоста.

3) Internet Explorer в роли хоста.

Если для приложения, хостом которого служит Internet Explorer, определен файл конфигурации, его расположение задается в теге **<link>** следующим образом.

```
<link rel="ИмяФайлаКонфигурации" href="location">
```

В параметре **location** указывается **URL**-адрес файла конфигурации. Таким образом задается базовая папка приложения. Файл конфигурации должен размещаться на том же веб-узле, что и приложение.

В файлах конфигурации безопасности содержатся сведения об иерархии групп кода и наборах разрешений, связанных с уровнем политики. Для изменения политики управления доступом для кода настоятельно рекомендуется использовать средство настройки платформы **.NET Framework (Mscorcfg.msc)** или средство настройки управления доступом для кода **(Caspol.exe)**, что гарантирует целостность файлов конфигурации безопасности.

В следующей таблице приведено расположение файлов конфигурации безопасности для Windows NT

Файл конфигурации	**Расположение файлов**
Файл конфигурации политики предприятия	%папка установки среды выполнения%\Config\Enterprisesec.config
Файл конфигурации политики компьютера	%папка установки среды выполнения%\Config\Security.config
Файл конфигурации политики	%USERPROFILE%\Application data\Microsoft\CLR security config\vxx.xx\Security.config

пользователя	

Схема файлов конфигурации для .NET Framework

Элемент **<configuration>** является корневым элементом в любом файле конфигурации, используемом средой **CLR** и приложениями **.NET Framework**.

```
<configuration>
    <!-- configuration settings -->
  </configuration>
```

Элемент **<assemblyBinding>** для элемента **<configuration>** определяет политику привязки сборок на уровне конфигурации.

```
<assemblyBinding
   xmlns="urn:schemas-microsoft-com:asm.v1">
</assemblyBinding>
```

Элемент **<linkedConfiguration>** указывает файл конфигурации, который следует включить.

```
<linkedConfiguration
   href="URL of linked configuration file"/
```

Атрибуты

Атрибут	**Описание**
href	URL-адрес файла конфигурации, который необходимо включить. Для атрибута **href** поддерживается только один формат — "file://". Поддерживаются локальные файлы и файлы UNC.

Элемент **<linkedConfiguration>** упрощает обслуживание для сборок компонентов. Если одно или несколько приложений используют сборку, файл конфигурации которой находится в общеизвестном местоположении, файлы конфигурации приложений, использующих сборку, могут включать файл конфигурации сборки с помощью элемента **<linkedConfiguration>** и не использовать непосредственное включение информации о конфигурации. При обслуживании сборки компонентов обновление общего файла конфигурации позволяет обновить информацию о конфигурации для всех приложений, использующих эту сборку. Для связанных файлов конфигурации действуют следующие правила:

1) Настройки включенного файла конфигурации затрагивают только политику привязки загрузчика и используются только загрузчиком. Включенные файлы конфигурации

могут содержать настройки, не имеющие отношения к политикам привязки, но эти настройки не используются.

2) Для атрибута **href** поддерживается только один формат — **"file**.
3) Ограничение числа связанных конфигураций для файла конфигурации отсутствует.
4) Все связанные файлы конфигурации объединяются в один файл; схожим образом работает директива **#include** в **C/C++**.
5) Элемент **<linkedConfiguration>** разрешен только в файлах конфигурации приложений; в файле **Machine.config** этот элемент игнорируется.
6) Обнаруживаются и удаляются циклические ссылки. Если элементы **<linkedConfiguration>** ряда файлов конфигурации образуют цикл, этот цикл обнаруживается и обрывается.

В следующем примере показано, как можно включить файл конфигурации с локального жесткого диска.

```
<configuration>
  <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
    <linkedConfiguration
                  href="file://c:\Program Files\Contoso\sharedConfig.xml"/>
  </assemblyBinding>
</configuration>
```

Использование конфигурационных файлов при доступе к источникам данных

Внедрение строк соединения в код приложения может привести к появлению уязвимых мест в системе безопасности и проблем с обслуживанием. Незашифрованные строки соединения, скомпилированные в исходный код приложения, можно просматривать с помощью средства «*Дизассемблер **MSIL***». Кроме того, после изменения строки соединения необходимо перекомпилировать приложение. По этим причинам рекомендуется хранить строки соединения в файле конфигурации приложения.

Напомним, файлы конфигурации приложения содержат настройки, относящиеся к отдельному приложению. Приложение **ASP.NET** может иметь один или несколько файлов **web.config**, а приложение **Windows** может иметь дополнительный файл **app.config**. В файлах конфигурации присутствуют общие элементы, хотя имя и расположение любого файла конфигурации в значительной степени зависят от того, где размещается приложение.

Строки соединения могут храниться в виде пар «*ключ/значение*» в разделе **connectionStrings** элемента **configuration** файла конфигурации приложения. Дочерние элементы включают **add**, **clear** и **remove**.

Следующий фрагмент файла конфигурации демонстрирует схему и синтаксис хранения строки соединения. Атрибут **name** является именем, которое задано для уникальной идентификации строки соединения, чтобы ее можно было получить во время выполнения.

Атрибут **providerName** является неизменяемым именем поставщика данных **.NET Framework**, которое регистрируется в файле **machine.config**

Чтобы добавить файл конфигурации приложения в проект в меню «Проект» выбираем команду «Добавить новый элемент», открывается диалоговое окно «Добавление нового элемента». Выбираем шаблон «Файл конфигурации приложения» и нажимаем кнопку «Добавить». В проект будет добавлен файл с именем **app.config**.

```
<?xml version='1.0' encoding='utf-8'?>
<configuration>
  <connectionStrings>
    <clear />
    <add name="Name"
     providerName="System.Data.ProviderName"
     connectionString="Valid Connection String;" />
  </connectionStrings>
</configuration>
```

При построении проекта среда разработки автоматически создает копию файла **app.config**, изменяет имя этого файла, чтобы оно совпадало с именем исполняемого файла, и перемещает новый файл с расширением **CONFIG** в каталог **bin**.

Внешние файлы конфигурации представляют собой отдельные файлы, каждый из которых содержит фрагмент файла конфигурации, состоящий из одного раздела. В таком случае основной файл конфигурации ссылается на внешний файл конфигурации. Хранение раздела **connectionStrings** в физически отдельном файле становится удобным в ситуациях, когда может потребоваться внесение изменений в строки соединения после развертывания приложения. Кроме того, для ограничения доступа к внешним файлам конфигурации могут использоваться средства обеспечения безопасности доступа к файлам и разрешения. Работа с внешними файлами конфигурации во время выполнения осуществляется в прозрачном режиме и не требует разработки специального кода.

Для хранения строк соединения во внешнем файле конфигурации создайте отдельный файл, содержащий единственный раздел **connectionStrings**. Не следует включать какие-либо дополнительные элементы, разделы или атрибуты. В данном примере показан синтаксис внешнего файла конфигурации:

```
<connectionStrings>
  <add name="Name"
   providerName="System.Data.ProviderName"
   connectionString="Valid Connection String;" />
</connectionStrings>
```

В основном файле конфигурации приложения для указания полного имени и расположения внешнего файла используется атрибут **configSource**. В следующем примере применяется ссылка на внешний файл конфигурации с именем **connections.config**.

```
<?xml version='1.0' encoding='utf-8'?>
<configuration>
  <connectionStrings configSource="connections.config"/>
</configuration>
```

В **.NET Framework 2.0** в пространстве имен **System.Configuration** появились новые классы, упрощающие извлечение строк соединения из файлов конфигурации во время выполнения. Предусмотрена возможность получить строку соединения программным путем по ее имени или по имени поставщика.

Файл **machine.config** также содержит раздел **connectionStrings**, включающий в себя строку соединения, используемую **Visual Studio**. При получении строк соединения из файла **app.config** приложения Windows с помощью имени поставщика в первую очередь загружаются строки соединения из файла **machine.config**, затем записи из файла **app.config**. Добавление ключевого слова **clear** сразу после элемента **connectionStrings** приводит к удалению из памяти всех ссылок, унаследованных от структуры данных, поэтому учитываются только строки соединения, определенные в локальном файле **app.config**.

Для работы с файлами конфигурации на локальном компьютере используется класс **ConfigurationManager**, заменивший устаревший класс **ConfigurationSettings**. **WebConfigurationManager**, использовавшийся для работы с файлами конфигурации **ASP.NET**. Он создан для работы с файлами конфигурации веб-сервера и предоставляет программный доступ к разделам файла конфигураций.

Для получения строк соединения из файлов конфигурации приложения используется **ConnectionStringSettingsCollection**. Этот объект содержит коллекцию объектов **ConnectionStringSettings**, каждый из которых представляет одну запись в разделе **connectionStrings**. Его свойства сопоставляются с атрибутами строк соединения, что позволяет получить строку соединения, указав имя строки или имя поставщика.

Свойство	**Описание**
Name	Имя строки соединения. Сопоставляется с атрибутом **name**.
ProviderName	Полное имя поставщика. Сопоставляется с атрибутом **providerName**.
ConnectionString	Строка соединения. Сопоставляется с атрибутом **connectionString**.

Приведем пример отображения списка всех строк соединения. В данном примере выполняется итерация в коллекции **ConnectionStringSettings** и отображение свойств **Name**, **ProviderName** и **ConnectionString** в окне консоли. Файл **System.Configuration.dll** не включается в проекты всех типов, поэтому для использования классов конфигурации может потребоваться сформировать на него ссылку.

```
using System.Configuration;
```

```
class Program
{
    static void Main()
    {
        GetConnectionStrings();
        Console.ReadLine();
    }

    static void GetConnectionStrings()
    {
        ConnectionStringSettingsCollection settings =
            ConfigurationManager.ConnectionStrings;

        if (settings != null)
        {
            foreach(ConnectionStringSettings cs in settings)
            {
                Console.WriteLine(cs.Name);
                Console.WriteLine(cs.ProviderName);
                Console.WriteLine(cs.ConnectionString);
            }
        }
    }
}
```

Продемонстрируем способ получения строки соединения из файла конфигурации путем указания ее имени. Код создает объект **ConnectionStringSettings**, сопоставляя указанный входной параметр с именем **ConnectionStrings**. Если совпадающее имя не найдено, функция возвращает значение **null**.

```
static string GetConnectionStringByName(string name)
{
    string returnValue = null;

    ConnectionStringSettings settings =
        ConfigurationManager.ConnectionStrings[name];

    if (settings != null)
```

```
        returnValue = settings.ConnectionString;


    return returnValue;
}
```

А теперь продемонстрируем способ получения строки соединения путем указания неизменяемого имени поставщика в формате **System.Data.ProviderName**. В коде выполняется итерация по **ConnectionStringSettingsCollection** и происходит возврат строки соединения для первого найденного **ProviderName**. Если имя поставщика не найдено, функция возвращает значение **null**.

```
static string GetConnectionStringByProvider(string providerName)
{
    string returnValue = null;

    ConnectionStringSettingsCollection settings =
        ConfigurationManager.ConnectionStrings;

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

Шифрование разделов файлов конфигурации с использованием защищенной конфигурации

В **ASP.NET 2.0** появилась новая возможность, защищенная конфигурация, с помощью которой можно шифровать в файле конфигурации конфиденциальную информацию. Защищенная конфигурация разрабатывалась в первую очередь для **ASP.NET**, но ее можно также использовать для шифрования разделов файлов конфигурации в приложениях Windows.

В приведенном ниже фрагменте файла конфигурации показан раздел **connectionStrings** после шифрования. В разделе **configProtectionProvider** задается поставщик защищенной конфигурации, который используется для шифрования и дешифрования строк соединения. Раздел **EncryptedData** содержит зашифрованный текст.

```
<connectionStrings
             configProtectionProvider="DataProtectionConfigurationProvider">
  <EncryptedData>
    <CipherData>
      <CipherValue>AQAAANCMnd8BFdERjHoAwE/Cl+sBAAAAH2... </CipherValue>
    </CipherData>
  </EncryptedData>
</connectionStrings>
```

Когда зашифрованная строка соединения извлекается во время выполнения, платформа **.NET Framework** с помощью указанного поставщика дешифрует значение **CipherValue** и передает его приложению. Нет необходимости создавать дополнительный код для управления процессом расшифровки.

Поставщики защищенной конфигурации регистрируются в разделе **configProtectedData** файла **machine.config** на локальном компьютере. Два поставщика защищенной конфигурации поставляются с платформой **.NET Framework**.

Поставщики защищенной конфигурации

Поставщик	**Описание**
RSAProtectedConfigurationProvider	Использует алгоритм RSA для шифрования и расшифровки данных. Алгоритм RSA можно использовать для шифрования с открытым ключом и цифровых подписей. Он также известен как алгоритм с «открытым ключом» или алгоритм асимметричного шифрования, поскольку в нем используется два разных ключа.
DPAPIProtectedConfigurationProvider	Использует API-интерфейс защиты данных (DPAPI) для шифрования разделов конфигурации. Он использует встроенные службы шифрования Windows и может быть настроен для защиты отдельных компьютеров или отдельных учетных записей пользователей.

Оба поставщика обеспечивают надежное шифрование данных. Однако, если планируется использовать один файл конфигурации на нескольких серверах, то только **RsaProtectedConfigurationProvider** позволяет экспортировать ключи шифрования, применяемые для шифрования данных, и импортировать их в другой сервер.

Пространство имен **System.Configuration** предоставляет классы для программной обработки параметров конфигурации. Класс **ConfigurationManager** обеспечивает доступ к компьютеру, приложению и пользовательским файлам конфигурации.

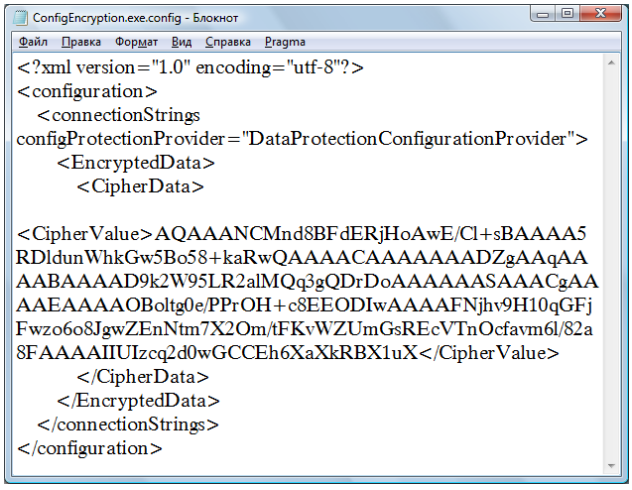
Продемонстрируем переключение шифрования раздела **connectionStrings** в файле **app.config** для приложения Windows. В этом примере процедура принимает имя приложения в качестве аргумента, например «**MyApplication.exe**». Затем файл **app.config** зашифровывается и копируется в папку, которая содержит исполняемый файл с именем «**MyApplication.exe.config**». Строку соединения можно расшифровать *только* на компьютере, где она была зашифрована.

В примере используется метод **OpenExeConfiguration**, чтобы открыть файл **app.config** для изменения, а метод **GetSection** возвращает раздел **connectionStrings**. Затем код проверяет свойство **IsProtected**, вызывая метод **ProtectSection** для шифрования раздела, если он не зашифрован. Метод **UnProtectSection()** вызывается для расшифровки раздела. Метод **Save** завершает операцию и сохраняет изменения. Для запуска кода необходимо задать ссылку на **System.Configuration.dll** в проекте.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Configuration;

namespace ConfigEncryption
{
    class Program
    {
        static void Main(string[] args)
        {
            ToggleConfigEncryption(@"ConfigEncryption.exe");
            Console.ReadKey(true);
        }

        static void ToggleConfigEncryption(string exeConfigName)
        {
            try
            {
                Configuration config = ConfigurationManager.
                    OpenExeConfiguration(exeConfigName);

                ConnectionStringsSection section =
                    config.GetSection("connectionStrings")
                    as ConnectionStringsSection;
```

```
                if (section.SectionInformation.IsProtected)
                {
                    section.SectionInformation.UnprotectSection();
                }
                else
                {
                    section.SectionInformation.ProtectSection(
                        "DataProtectionConfigurationProvider");
                }

                config.Save();

                Console.WriteLine("Protected={0}",
                    section.SectionInformation.IsProtected);
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
        }
    }
}
```

После шифрования файл конфигурации принимает следующий вид:



4. Домашнее задание

Разработать приложение ADO.NET, реализующее работу с базой данных учебного заведения. Приложение должно обеспечивать следующую функциональность:

Добавление, удаление, просмотр и модификация информации о

- Студентах заведения
- Преподавателях заведения
- Учебных аудиториях
- Читаемых предметах
- Составах групп
- Расписании групп
- Расписании преподавателей
- Учебных планах
- Расписании звонков

Работу приложения основать на фабричном конструктивном шаблоне. Использовать асинхронные механизма доступа к данным, для оптимизации производительности работы приложения. Использовать конфигурационные файлы для соединения с базой данных.