

top

КОМПЬЮТЕРНАЯ
АКАДЕМИЯ

Технология доступа к базам данных ADO.NET

Урок №1

Присоединенный режим работы

Содержание

Общее представление об ADO.NET	3
Присоединенный режим работы.....	7
Создание БД	7
Подключение к БД (DbConnection).....	14
Создание и выполнение запросов (DbCommand) ...	19
Получение и обработка результатов запросов (DbDataReader).....	25
Пакетная обработка запросов	31
Конфигурационный файл	35
Параметризованные запросы в DbCommand ..	38
Хранимые процедуры в DbCommand	42
Домашнее задание	48

Общее представление об ADO.NET

Вы уже знаете, что такое реляционные базы данных. Теперь к этим знаниям вам надо добавить умение писать программы, которые будут использовать для своих потребностей реляционные базы данных. В ходе изучения этого курса вы познакомитесь с фреймворком ADO.NET, предназначенным для работы с базами данных в .NET Framework. ADO.NET представляет собой набор классов, объединенных в пространстве имен System.Data и позволяющих осуществлять работу с такими поставщиками данных, как MS SQL Server, OLE DB, ODBC и Oracle. Другими словами, вы научитесь писать программы на C#, которые будут уметь работать с указанными реляционными базами данных.

Из каких этапов состоит работа с БД? Прежде всего, приложение должно подключиться к поставщику данных, который содержит необходимую вам БД. Поставщиком данных выступает программный продукт, называемый сервером баз данных или СУБД, который является контейнером для пользовательских БД. Для подключения к такому серверу вы должны знать адрес сервера, имя и пароль для подключения к серверу, имя нужной вам БД, иногда еще некоторые значения. Подключившись к серверу и открыв свою БД, вы должны иметь инструменты, позволяющие вам пересылать серверу свои SQL

запросы и получать от сервера результаты выполнения этих запросов. Значит, ваши программы должны будут уметь подключаться к серверам с БД и выполнять запросы к БД. Кроме этого, ваши программы должны будут уметь хранить данные, полученные из БД. Вот эти две основные задачи: доступ к базам данных и хранение информации из таблиц и решают классы, входящие в состав ADO.NET.

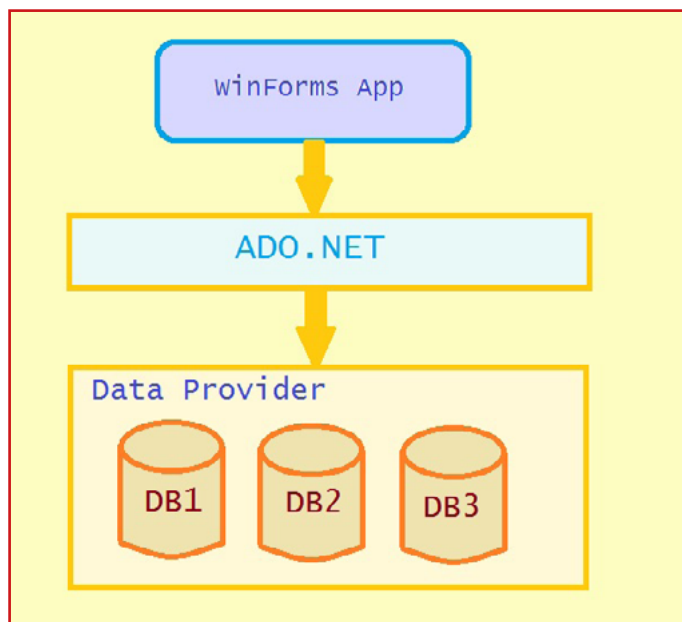


Рис. 1. Взаимодействие приложения с сервером БД

Для подключения к БД и выполнения SQL запросов используются классы:

- `DbConnection;`
- `DbCommand;`
- `DbDataReader;`
- `DbDataAdapter;`

Для хранения данных из БД применяются такие классы:

- DataTable;
- DataSet;
- Другие

С самого начала нашего курса вы должны запомнить, что существует два принципиально разных способа работы приложения с БД: присоединенный режим и отсоединенный режим. Далее мы более подробно поговорим об этих двух режимах.

При присоединенном режиме приложение подключается к БД и остается в подключенном состоянии продолжительное время. На протяжении этого времени приложение может обращаться к БД и выполнять какие-либо запросы. В то же самое время приложение может заниматься и другими своими делами, не обращаясь к БД, но удерживая при этом открытое соединение с БД.

При отсоединенном режиме приложение подключается к БД, выполняет необходимое действие, например, читает данные из одной или нескольких таблиц, и тут же отключается от БД, сохраняя прочитанные данные у себя в соответствующих классах для локальной работы с этими данными. При необходимости выполнить новый запрос, приложение снова подключается к БД, выполняет запрос и тут же отключается.

Другими словами, при присоединенном режиме приложение остается в подключенном к БД состоянии продолжительное время, а при отсоединенном режиме работы приложение подключается к БД в дискретном режиме только на время выполнения каждого запроса.

У каждого из этих способов работы есть свои преимущества и недостатки и вам надо научиться использовать в каждой конкретной ситуации оптимальный метод. Или грамотно комбинировать оба эти способа работы с БД.

На данном этапе давайте отметим основные характеристики этих режимов работы. Поскольку подключение к серверу процедура достаточно трудоемкая и затратная по времени, то при присоединенном режиме работы запросы выполняются быстрее, так как каждому последующему запросу не надо выполнять подключение к БД. Это подключение выполнено и остается в открытом состоянии. Но такой способ работы здорово грузит сервер. Ведь серверу приходится удерживать связь со многими клиентами, даже с теми, которые в данный момент с ним не работают, но удерживают открытым свое подключение. Второй режим работы намного меньше грузит сервер, так как каждый клиент подключается только на микроскопический период времени, пока выполняется запрос, а затем сразу отключается. Но в этом случае, каждому запросу приходится выполнять подключение «с нуля», что несколько замедляет работу приложения. Пока остановимся на этих отличиях. О других поговорим позже.

Перейдем к рассмотрению основных этапов использования ADO.NET.

Присоединенный режим работы

Создание БД

Давайте создадим простую БД с именем Library, а в ней создадим две связанные таблицы Books и Authors. База данных может размещаться, как на выделенном сервере, так и на локальном сервере. Принципы работы с БД не зависят от ее расположения.

Запускаем Visual Studio 2015 и создаем консольное приложение с произвольным именем, например AdoNetLibrary1. Прежде всего нам надо создать БД, с которой будет работать приложение. Это можно сделать прямо в Visual Studio 2015. Перейдите в меню View и активируйте опцию SQL Server Object Explorer. Выделите узел SQL Server, активируйте контекстное меню и выберите опцию Add SQL Server.

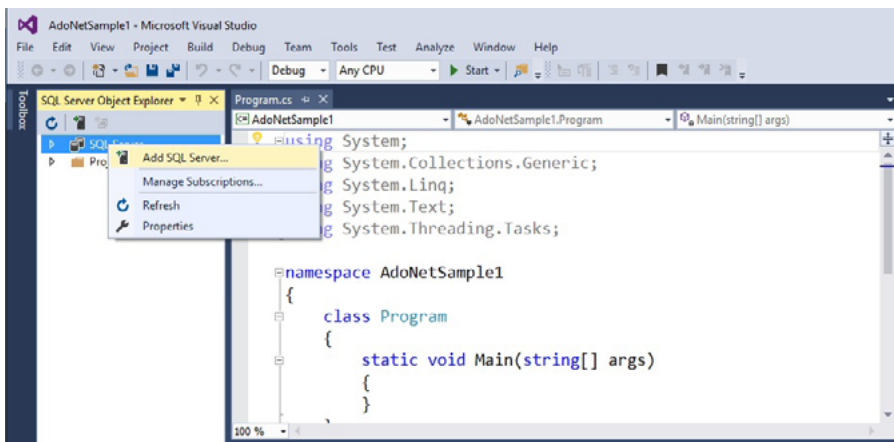


Рис. 2. Добавление нового SQL Server

В появившемся окне вы увидите атрибуты создаваемого сервера. По умолчанию Visual Studio 2015 создает сервер LocalDB. LocalDB — это облегченный вариант SQL Server Express Edition. Он поддерживает всю функциональность SQL Server Express Edition (за исключением FileStream), но устанавливается быстрее и выполняется в пользовательском режиме, а не как служба. LocalDB не предназначен для сценариев с удаленным подключением, но, поскольку сама БД хранится в файле (с расширением .mdf), позволяет легко переносить приложение с компьютера на компьютер. Другими словами, LocalDB — это идеальное средство для отладки (и не только), и научиться работать с ним будет полезно каждому. Более подробную информацию о LocalDB можете поискать в сети — например, [здесь](#).

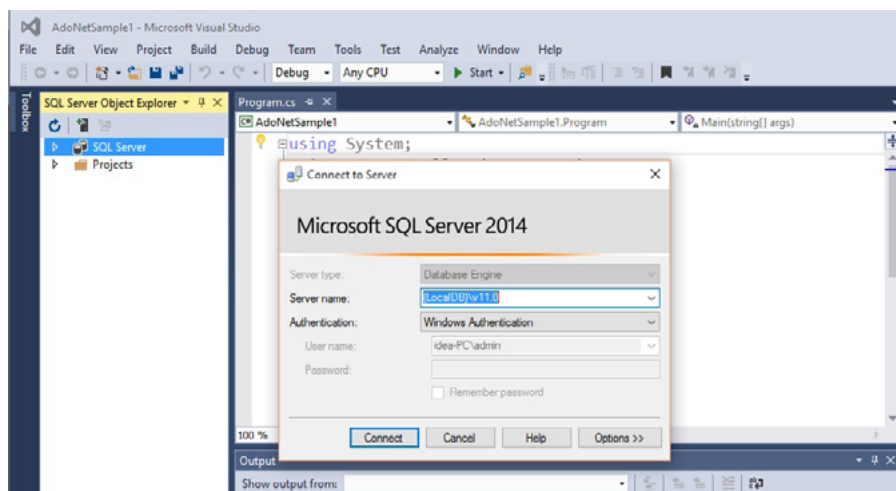


Рис. 3. Создание нового SQL Server

Оставим эту опцию без изменений. Также оставим без изменения способ аутентификации для доступа к серверу.

Нажимаем Connect. После выполнения этих действий мы имеем поставщика данных, который может хранить наши пользовательские БД и управлять ими.

Теперь надо создать новую БД. Наша база данных будет состоять из двух связанных таблиц Books и Authors. Такой простой БД нам будет достаточно на этом этапе, чтобы познакомиться с базовыми возможностями фреймворка ADO.NET. Для создания новой БД снова переходим в окно SQL Server Object Explorer. Раскрываем узел созданного сервера. Выделяем узел Databases и активируем для него контекстное меню. В контекстном меню активируем опцию Add New Database.

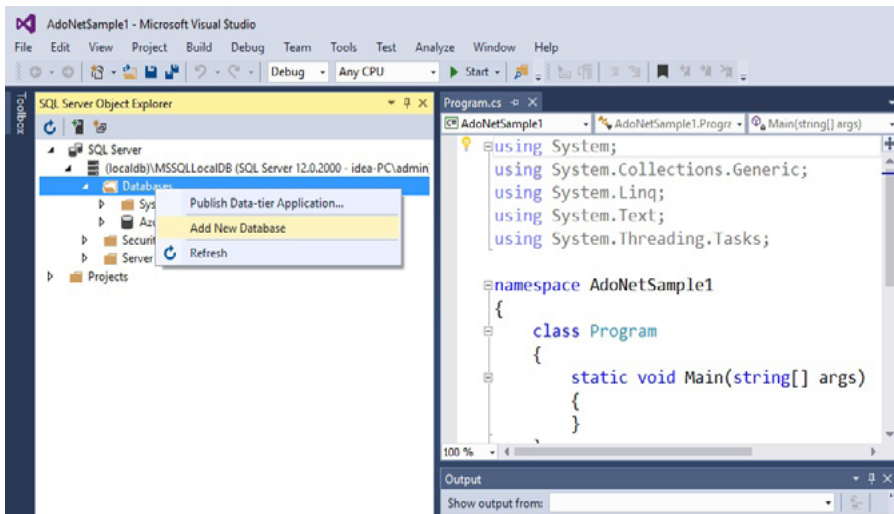


Рис. 4. Добавление новой базы данных

В появившемся окне можно задать имя создаваемой БД и ее расположение. Введем для нашей БД имя Library, а расположение оставим без изменения.

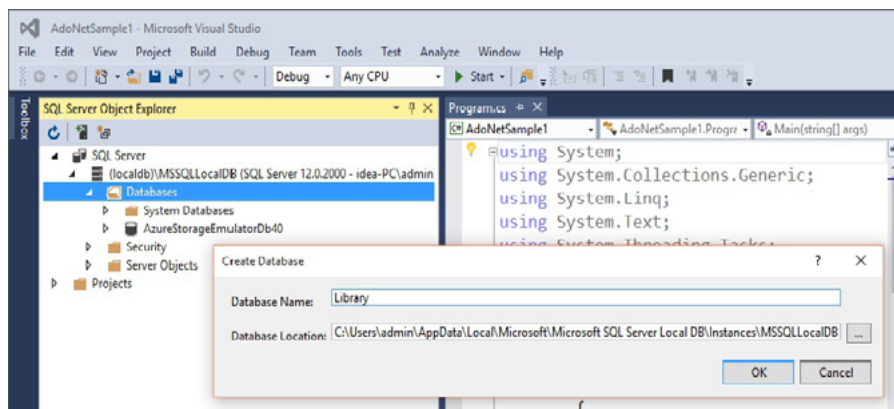


Рис. 5. Создание новой базы данных Library

Теперь мы можем перейти к созданию таблиц в нашей БД. Для этого надо перейти к узлу Tables созданной БД и из контекстного меню активировать опцию Add New Table...

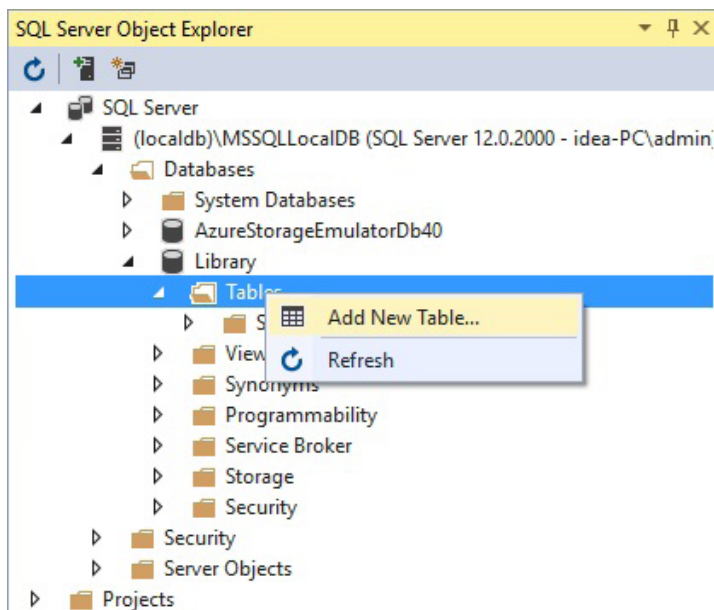


Рис. 6. Добавление новой таблицы в базу данных Library

После выполнения этого действия Visual Studio 2015 переведет вас в режим изменения структуры БД. В этом режиме можно создавать таблицы. Для создания таблиц можно использовать возможности встроенного графического редактора. Мы же создадим наши таблицы с помощью DDL SQL запросов. Ниже приведены запросы для создания наших двух таблиц и связи между ними. Сначала надо создать таблицу Authors, а затем — Books.

```
CREATE TABLE [dbo].[Authors]
(
    Id INT NOT NULL IDENTITY(1,1) PRIMARY KEY,
    FirstName VARCHAR(100) NOT NULL,
    LastName VARCHAR(100) NOT NULL
)

CREATE TABLE [dbo].[Books]
(
    Id INT NOT NULL IDENTITY(1,1) PRIMARY KEY,
    AuthorId INT NOT NULL,
    FOREIGN KEY (AuthorId) REFERENCES AUTHORS (Id),
    Title VARCHAR(100) NOT NULL,
    PRICE INT,
    PAGES INT
)
```

Скопируйте запрос для создания таблицы Authors в окно T-SQL в нижней части экрана, проверьте, что запрос скопировался без ошибок и нажмите кнопку Update на верхней панели.

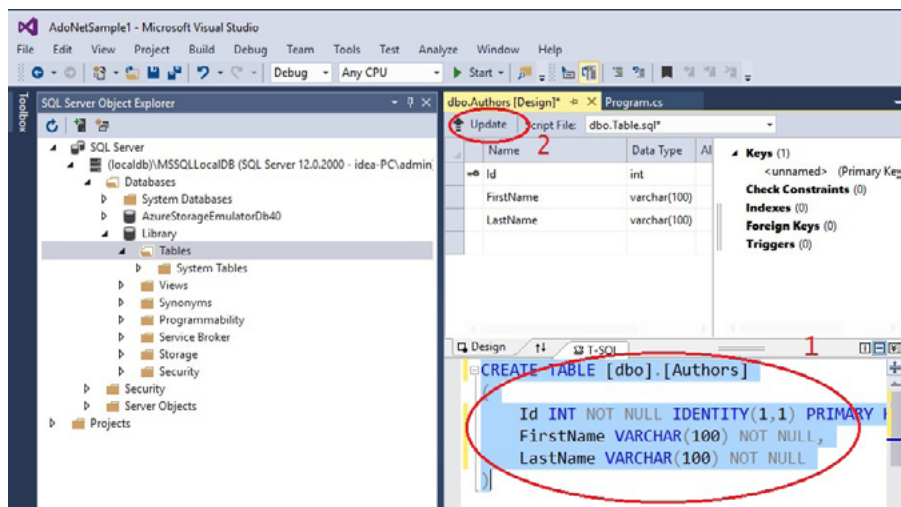


Рис. 7. Создание таблицы Author в базе данных Library

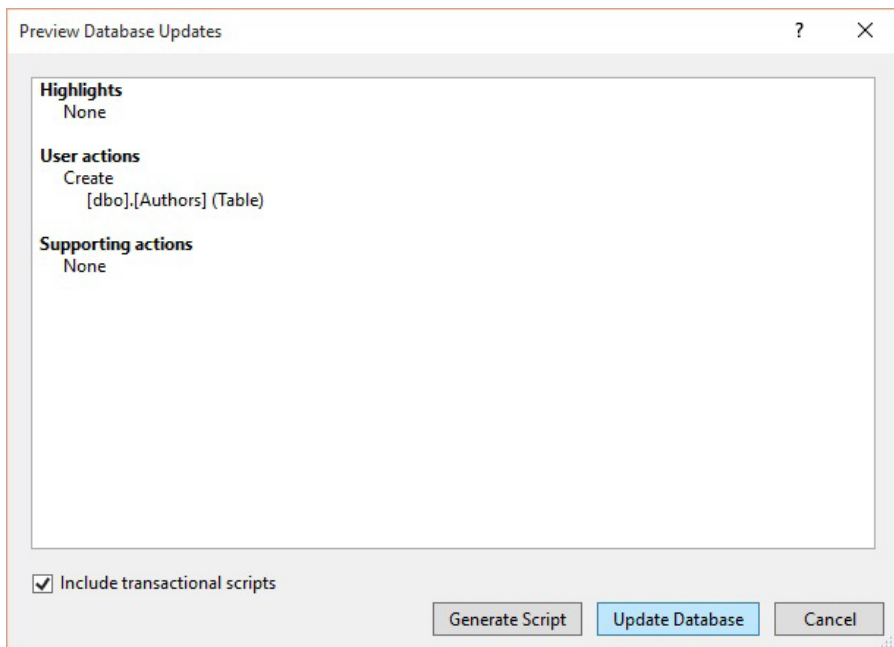


Рис. 8. Подтверждение выполнения запроса

После нажатия на кнопку Update Visual Studio 2015 предложит вам подтвердить действие, приводящее к изменению структуры БД (Рис. 8).

В появившемся окне надо нажать кнопку Update Database и новая таблица будет создана. Если говорить более точно, то после нажатия на эту кнопку будет выполнен запрос, размещенный в окне T-SQL. В нашем случае этот запрос создает новую таблицу.

Аналогичным образом скопируйте запрос для создания таблицы Books и выполните его. Теперь в нашей БД созданы все необходимые таблицы и связи между ними. Разверните узел Tables в нашей БД и вы увидите там созданные таблицы.

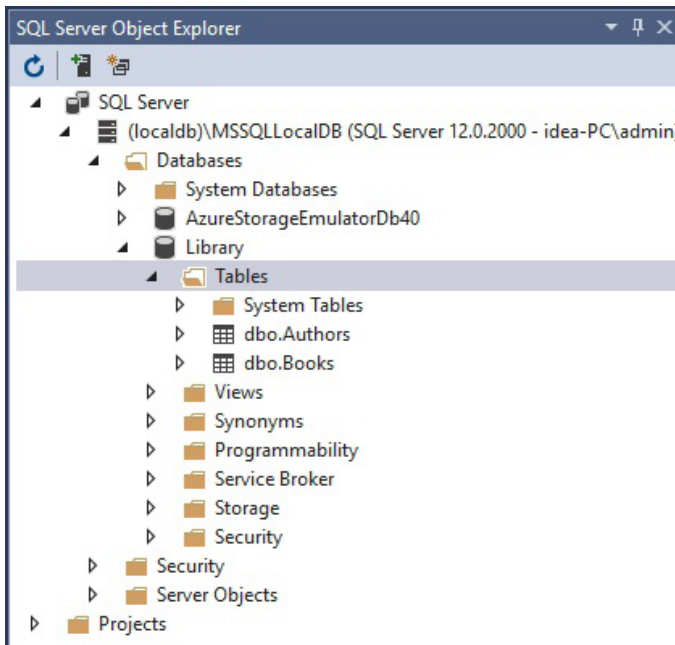


Рис. 9. Созданные таблицы

Подключение к БД (DbConnection)

Сейчас мы начинаем рассматривать конкретные способы применения классов фреймворка ADO.NET. Отметьте себе, что перечисленные выше классы, имена которых начинаются с Db, определены в пространстве имен System.Data.Common и являются абстрактными классами. Каждый из них является базовым типом для ряда производных классов. Рассмотрим иерархии этих классов.

Базовый класс	Производные классы
DbConnection	SqlConnection OleDbConnection OdbcConnection OracleConnection
DbCommand	SqlCommand OleDbCommand OdbcCommand OracleCommand
DbDataReader	SqlDataReader OleDbDataReader OdbcDataReader OracleDataReader
DbDataAdapter	SqlDataAdapter OleDbDataAdapter OdbcDataAdapter OracleDataAdapter

Как вы понимаете, каждый из классов потомков предназначен для работы с соответствующей конкретной БД. Существование этих иерархий делает возможным создание полиморфного кода, позволяющего приложению работать с разными БД.

Надо отметить, что какое-то время тому назад Microsoft перестала поддерживать классы ADO.NET, предназначенные для работы с Oracle. Подробнее поговорим об этом в следующих разделах нашего урока.

Будем рассматривать в качестве примера подключение к базе данных MS SQL Server. При этом очередность выполняемых действий будет одинаковой для случая подключения к любой из других допустимых БД. Поменяются только используемые классы.

Мы уже говорили о том, что для подключения к серверу надо знать некоторые константные значения. При работе с базами данных набор таких значений указывается в так называемой строке подключения. Строки подключения объединяют в себе в специальном формате значения, позволяющие подключаться к конкретному серверу. Поэтому, для подключения к серверу надо знать его строку подключения. Как узнать строку подключения для каждого сервера? Можно спросить у Google.

Мы с вами собираемся подключаться к серверу MS SQL Server, и очень важную роль в таком подключении играет строка подключения. Сначала почитайте о строке подключения в этом уроке. Потом, когда у вас будет начальное понимание об этом объекте, вам будет полезно почитать о строке подключения на MSDN.

Существует обязательный набор значений, который надо знать для подключения к серверу. Кроме этих значений есть еще ряд других для более тонкой настройки.

К обязательным относятся:

- Имя сервера
- Имя базы данных
- Способ аутентификации на сервере
- Имя и пароль для входа на сервер для случая SQL Authentication

Давайте рассмотрим, какими должны быть все эти значения для нашего случая.

Имя сервера у нас — (localdb)\v11.0. Это зарегистрированное имя сервера LocalDB. Его надо указать с точностью до каждого символа, как указано здесь. Регистр символов в значении localdb не важен. Если бы мы работали с удаленным выделенным сервером, то его имя могло бы выглядеть как-нибудь так [\\10.3.0.10\MyServer](#) или даже просто так [\\10.3.0.20](#). Если вы не знаете имени сервера, спросите об этом у системного администратора.

Имя сервера в строке подключение указывается с помощью атрибута Data Source:

```
Data Source=(localdb)\v11.0;
```

Обратите внимание, что значение атрибута не заключается в кавычки и в конце значения указывается символ «;».

Имя базы данных в строке подключение указывается с помощью атрибута Initial Catalog:

```
Initial Catalog=Library;
```

Обратите внимание, что значение атрибута не заключается в кавычки и в конце значения указывается символ «;».

От способа аутентификации на сервере зависит внешний вид самой строки подключения. Если сервер настроен на аутентификацию Windows, то в строке подключения не надо указывать имя и пароль пользователя для подключения к серверу. Если же аутентификация Windows отключена, то в строке подключения надо указывать имя и пароль пользователя для подключения к серверу.

Способ аутентификации указывается в строке подключения с помощью атрибута `Integrated Security`. Правда, значения этого атрибута отличаются для разных серверов. Для MS SQL Server указать в строке подключения аутентификацию Windows можно двумя способами:

```
Integrated Security=true;
```

или

```
Integrated Security=SSPI;
```

Снова обратите внимание, что значение атрибута не заключается в кавычки и в конце значения указывается символ «;». Значения этого атрибута для других серверов рассмотрим позже.

Если же в строке подключения отсутствует атрибут `Integrated Security`, то это значит, что вход на сервер выполняется в режиме аутентификации SQL Server и в этом случае, в строке подключения надо указывать имя пользователя, зарегистрированного на сервере и пароль этого пользователя. Эти значения указываются в строке подключения с атрибутами `User ID` и `Password`:

```
User ID=имя_пользователя; Password=пароль;
```

Обратите внимание, что значение атрибутов не заключается в кавычки и в конце значения указывается символ «;». Также обратите внимание, что имя пользователя и пароль указываются в строке подключения в открытом виде. О возникающих в связи с этим проблемах и о способах решения этих проблем поговорим позже.

Итак, мы перечислили основные значения для строки подключения и в нашем случае строка подключения должна выглядеть так:

```
Data Source=(localdb)\v11.0; Initial Catalog=Library;  
Integrated Security=SSPI;
```

Если бы на наш сервер надо было входить с именем “manager” и паролем “123456”, то строка подключения была бы такой:

```
Data Source=(localdb)\v11.0; Initial Catalog=Library;  
User ID=manager; Password=123456;
```

Переходим к созданию подключения к нашей БД. Подключение выполняется с помощью класса `DbConnection`, точнее — с помощью какого-либо класса, производного от `DbConnection`. Мы будем использовать для подключения к нашему серверу класс `SqlConnection`. Потому, что нашим поставщиком данных является MS SQL Server.

Существует несколько конструкторов для создания объектов класса `SqlConnection`. Но в любом случае вы должны знать строку подключения к своей БД. Мы строку подключения знаем. Поэтому можем создать объект класса `SqlConnection` одним из следующих способов:

```

SqlConnection conn = null;
conn = new SqlConnection();
conn.ConnectionString = @"Data Source=(localdb)\v11.0;
Initial Catalog=Library; Integrated Security=SSPI;";
//или

SqlConnection conn = null;
conn = new SqlConnection(@"Data Source=(localdb)\v11.0;
Initial Catalog=Library; Integrated Security=SSPI;");

```

Как видите, разница заключается в способе инициализации свойства `ConnectionString` экземпляра `SqlConnection` значением строки подключения.

Необходимо учитывать, что создание объекта `SqlConnection` не приводит к автоматическому выполнению подключения к БД. Чтобы выполнить подключение надо вызвать метод `conn.Open()`, а чтобы выполнить отсоединение от сервера, надо вызвать метод `conn.Close()`. Теперь мы готовы продолжать. Давайте посмотрим, каким образом можно передавать в БД и выполнять SQL запросы. Для этого надо использовать класс `SqlCommand`.

Создание и выполнение запросов (SqlCommand)

Вы уже понимаете, что мы будем использовать в нашем приложении класс `SqlCommand`, производный от класса `SqlCommand`. В этом классе есть два свойства, которые обязательно должны быть заполнены. Это свойства `Connection` и `CommandText`. Свойство `Connection` имеет тип `DbConnection` и должно инициализироваться объектом этого типа. У нас такой объект уже есть — это наш `conn`. Свойство `CommandText` имеет тип `String` и должно

инициализироваться каким-либо SQL запросом. Эти два свойства можно инициализировать при создании объекта `SqlCommand`, используя конструктор с двумя параметрами, а можно создать объект `SqlCommand` конструктором без параметров, а затем инициализировать эти свойства отдельно:

```
string insertString = @"insert into Authors
                        (FirstName, LastName)
                        values ('Roger', 'Zelazny')";
SqlCommand cmd = new SqlCommand(insertString, conn);
или
string insertString = @"insert into Authors
                        (FirstName, LastName)
                        values ('Roger', 'Zelazny')";
SqlCommand cmd = new SqlCommand();
cmd.Connection = conn;
cmd.CommandText = insertString;
```

Точно так же, как создание объекта `SqlConnection` не приводит автоматически к установлению соединения с БД, создание объекта `SqlCommand` не приводит к немедленному выполнению запроса, занесенного в свойство `CommandText`. Для выполнения подготовленного запроса надо вызвать один из специальных методов. Таких методов несколько. Они делятся на синхронные и асинхронные. Сейчас мы воспользуемся синхронным выполнением запроса к БД. Но и методов для выполнения синхронных запросов существует несколько. Рассмотрим эти методы, предназначенные для выполнения запросов разных видов.

`ExecuteScalar()` предназначен для выполнения запросов, которые возвращают какое-либо значение, но при

этом этот метод возвращает только первое поле первой строки результата. Т.е. только одно значение. Этот метод можно использовать, например, для выполнения запросов с агрегационными функциями.

`ExecuteNonQuery()` предназначен для выполнения запросов `insert`, `update` и `delete`. Этот метод возвращает количество обработанных запросом строк в таблице, хотя этим возвращаемым значением чаще всего пренебрегают.

`ExecuteReader()` предназначен для выполнения запросов `select`. Возвращает результат выполнения запроса и размещает его в объекте типа `DbDataReader`.

Исходя из вышесказанного, для выполнения нашего запроса `insert`, мы должны использовать метод `ExecuteNonQuery()`. Например, таким образом:

```
cmd.ExecuteNonQuery();
```

Создадим в нашем приложении метод `InsertQuery()` в котором реализуем выполнение запроса `insert`. Теперь полный код нашего приложения может выглядеть так:

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

namespace AdoNetSample1
{
    class Program
    {
        SqlConnection conn=null;
```

```
public Program()
{
    conn = new SqlConnection();
    conn.ConnectionString = @"Data
        Source=(localdb)\MSSQLLocalDB;
        Initial Catalog=Library;
        Integrated Security=SSPI;";
}

static void Main(string[] args)
{
    Program pr = new Program();
    pr.InsertQuery();
}

public void InsertQuery()
{
    try
    {
        //открыть соединение
        conn.Open();
        //подготовить запрос insert
        //в переменной типа string
        string insertString = @"insert into
            Authors (FirstName, LastName)
            values ('Roger', 'Zelazny')";
        //создать объект command,
        //инициализировав оба свойства
        SqlCommand cmd =
            new SqlCommand (insertString, conn);

        //выполнить запрос, занесенный
        //в объект command
        cmd.ExecuteNonQuery();
    }
    finally
    {

```

```
// закрыть соединение
if (conn != null)
{
    conn.Close();
}
}
}
}
```

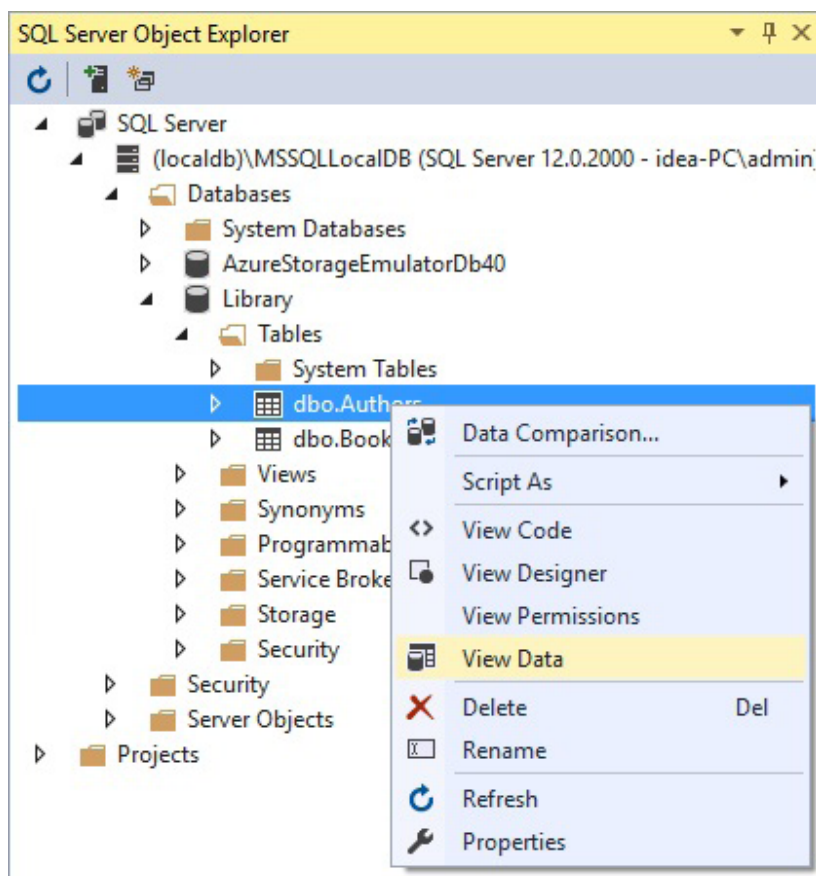


Рис 10. Просмотр данных таблицы

Выполните это приложение. Теперь давайте убедимся в том, что приложение выполнено успешно и в таблицу Authors добавлена информация о новом писателе. Для этого надо перейти в окно SQL Server Object Explorer. Затем последовательно развернуть узел нашего сервера, базы данных и узел Tables. Выделить таблицу Authors и из контекстного меню активировать опцию View Data (рис. 10).

Если все прошло без ошибок, то вы увидите в таблице занесенную информацию. Занесите таким образом в таблицу Authors еще несколько записей, чтобы нам было что смотреть в запросе select в следующей части нашего урока.

Подведем итог нашему знакомству с классом SqlCommand. Этот класс позволяет хранить запросы к БД, отправлять эти запросы серверу и принимать результаты выполнения этих запросов. Для хранения запросов используется свойство CommandText. Для выполнения запросов класс SqlCommand содержит три метода:

- *ExecuteReader()* предназначен для обработки запросов select и возвращает результат выполнения этих запросов в объект SqlDataReader;
- *ExecuteNonQuery()* предназначен для выполнения запросов insert, update и delete;
- *ExecuteScalar()* предназначен для выполнения запросов, возвращающих одно единственное агрегированное значение;

Получение и обработка результатов запросов (DbDataReader)

Теперь, когда у нас есть заполненная таблица, мы можем перейти к рассмотрению того, как выполнять запрос `select`. Вы конечно же понимаете специфику этого запроса, которая заключается в том, что он возвращает в качестве результата своего выполнения прочитанные строки. Нам надо научиться принимать и обрабатывать результат выполнения запроса `select`. Для этого мы будем использовать класс `DbDataReader`. Точнее — производный от него `SqlDataReader`.

В этот раз мы создадим объект `SqlCommand`, передав в него запрос “`select * from Authors`”. А для выполнения этого запроса вызовем метод `ExecuteReader()`, принимая возвращаемое им значение в объект `SqlDataReader`.

```
SqlDataReader rdr = null;
SqlCommand cmd = new SqlCommand("select *
                                from Authors", conn);
rdr = cmd.ExecuteReader();
```

Объект класса `SqlDataReader` представляет собой курсор, содержащий строки, полученные в результате выполнения запроса `select`. Работа с этим объектом выглядит таким образом. Чаще всего мы не знаем, сколько строк вернул выполненный запрос `select`. Но мы должны знать, что в объекте `SqlDataReader` есть внутренний указатель. Изначально он указывает на первую строку в этом объекте. Главную роль в обработке полученных данных выполняет метод `Read()`. Он извлекает строку, на которую указывает

указатель и превращает объект `SqlDataReader` в массив, содержащий извлеченную строку. После этого метод `Read()` перемещает указатель на следующую строку. Таким образом мы можем вызывать метод `Read()` в цикле и на каждой итерации объект будет `SqlDataReader` представлять собой массив, элементами которого будут поля текущей строки нашей таблицы. В случае нашей таблицы `Authors` и при использовании запроса `select`, который возвращает все поля таблицы, в полученном массиве будет три элемента — в первом будет храниться `id` писателя, во втором и третьем — имя и фамилия. Когда строки в курсоре закончатся, метод `Read()` вернет `false`. Обработка информации, занесенной в объект `SqlDataReader` может выполняться в таком цикле:

```
while (rdr.Read())
{
    Console.WriteLine(rdr[1] + " " + rdr[2]);
}
```

В классе `SqlDataReader` есть ряд других методов, позволяющих более тонко с учетом типа обрабатывать полученные данные. Эти методы мы рассмотрим в нашем уроке чуть позже. После обработки полученных результатов объект `SqlDataReader` надо закрыть, для освобождения занимаемых им ресурсов. Полный код метода `ReadData()` может выглядеть так:

```
public void ReadData()
{
    SqlDataReader rdr = null;
    try
    {
```

```

        //открыть соединение
        conn.Open();
        //создать новый объект command с запросом select
        SqlCommand cmd = new SqlCommand("select *
                                         from Authors", conn);
        //выполнить запрос select, сохранив
        //возвращенный результат
        rdr = cmd.ExecuteReader();

        //извлечь полученные строки
        while (rdr.Read())
        {
            Console.WriteLine(rdr[1] + " " + rdr[2]);
        }
    }
    finally
    {
        //закрыть reader
        if (rdr != null)
        {
            rdr.Close();
        }
        //закрыть соединение
        if (conn != null)
        {
            conn.Close();
        }
    }
}

```

Выполните наше приложение, вызвав метод `ReadData()`. При успешном выполнении вы увидите в консольном окне информацию из своей таблицы `Authors`. У меня это выглядело так:

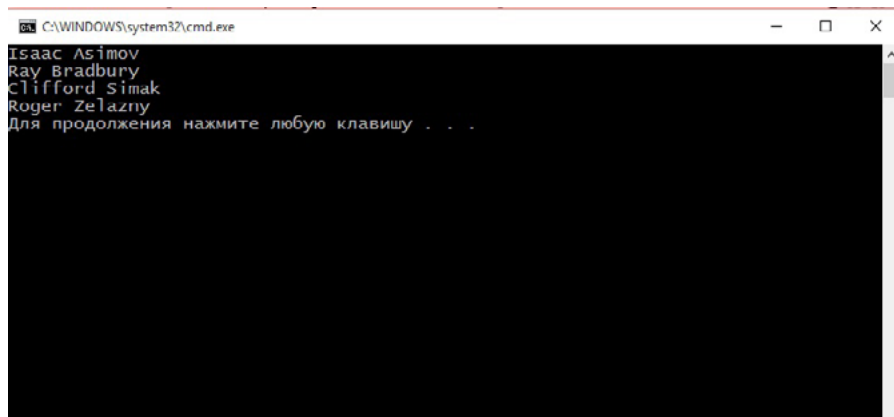


Рис. 11. Выполнение запроса select

Посмотрим на объект [SqlDataReader](#) повнимательнее. В нашем примере мы использовали только метод `Read()`. Вы уже поняли, что этот метод считывает каждую возвращенную запросом `select` строку, превращает ее в массив и заносит в объект [SqlDataReader](#). На каждой итерации этот массив перезаписывается данными из следующей строки. Необходимо помнить, что в объекте [SqlDataReader](#) мы в каждый момент времени имеем информацию только об одной строке из таблицы. Поэтому мы должны обработать каждую прочитанную строку в текущей итерации.

Предоставляет ли нам [SqlDataReader](#) какую-либо информацию о массиве, сформированном из текущей строки? Да. Например, из свойства `FieldCount` мы можем узнать количество полей, возвращенных запросом `select`. Если мы вызовем метод `GetName(index)` и передадим ему индекс в диапазоне от 0 до `FieldCount-1`, то мы получим имя поля таблицы, соответствующего указанному индексу. Другими словами, в объекте [SqlDataReader](#) хранится

информация об именах полей таблиц, прочитанных запросом `select` и эта информация доступна нам.

В нашем примере мы обращались к объекту как к индексированному массиву. Однако, это не единственная возможность. Если вам известны имена полей прочитанной таблицы, вы можете извлекать данные по именам этих полей. Другими словами, в строке

```
Console.WriteLine(rdr[1] + " " + rdr[2]);
```

можно было написать так:

```
Console.WriteLine(rdr["FirstName"] + " " +  
    + rdr["LastName"]);
```

Давайте изменим код нашего метода `ReadData()` с учетом этой новой информации и сделаем так, чтобы этот метод выводил имена прочитанных полей таблицы.

```
public void ReadData()  
{  
    SqlDataReader rdr = null;  
    try  
    {  
        //открыть соединение  
        conn.Open();  
  
        //создать новый объект command с запросом select  
        SqlCommand cmd = new SqlCommand("select *  
            from Authors", conn);  
        //выполнить запрос select, сохранив  
        //возвращенный результат  
        rdr = cmd.ExecuteReader();  
        int line = 0; //счетчик строк
```

```

//извлечь полученные строки
while (rdr.Read())
{
    //формируем шапку таблицы перед выводом
    //первой строки
    if (line == 0)
    {
        //цикл по числу прочитанных полей
        for (int i = 0; i < reader.FieldCount; i++)
        {
            //вывести в консольное окно имена полей
            Console.Write(rdr.GetName(i).
                ToString()+" ");
        }
        Console.WriteLine();
        line++;
        Console.WriteLine(rdr[1] + " " + rdr[2]);
    }
    Console.WriteLine("Обработано записей: " +
        line.ToString());
}

finally
{
    //закрыть reader
    if (rdr != null)
    {
        rdr.Close();
    }
    //закрыть соединение
    if (conn != null)
    {
        conn.Close();
    }
}
}

```

C:\WINDOWS\system32\cmd.exe

```

Id      FirstName    LastName
1       Isaac      Asimov
2       Ray        Bradbury
3       Clifford   Simak
1002    Roger       Zelazny
Total records processed: 4
Для продолжения нажмите любую клавишу . . .

```

Рис. 12. Выполнение модифицированного метода `ReadData()`

Подведем итог нашему знакомству с классом `SqlDataReader`. Этот класс позволяет хранить результаты выполнения запросов `select` и извлекать их последовательно один за другим в направлении от первого до последнего. Мы не можем возвращаться назад к извлеченным строкам. По окончании обработки объекта `SqlDataReader` его надо закрывать вызовом метода `Close()`.

Пакетная обработка запросов

Давайте поговорим о том, что произойдет, если в свойство `CommandText` объекта `SqlCommand` будет занесен не один запрос, а несколько. Оказывается, `SqlDataReader` совершенно спокойно умеет выполнять и возвращать результаты нескольких запросов. Единственное условие, которое надо при этом соблюдать — это отделять тексты запросов, заносимых в свойство `CommandText`, один от другого символами «;».

Если вы внимательно смотрели в студии на свойства и методы объекта [SqlDataReader](#), то могли заметить там метод `NextResult()`. Этот метод и предназначен для обработки результатов множества запросов, занесенных в свойство `CommandText` объекта [SqlCommand](#). Посмотрим на наш код, в котором обрабатывается результат запроса `select`. Обработка строк происходит в цикле `while`, до тех пор, пока метод `Read()` не вернет `false`. Но таким образом обрабатываются результаты только первого запроса. Если запросов несколько, наш код просто не узнает об этом. Чтобы уметь обрабатывать результаты нескольких запросов, надо наш цикл `while` вставить в другой цикл `do-while`, управляемый методом `NextResult()`:

```
int line = 0; // счетчик строк
// извлечь полученные строки
do {
    while (rdr.Read())
    {
        if (line == 0) //формируем шапку таблицы
        {
            //выводом первой строки
            //цикл по числу прочитанных полей
            for (int i = 0; i < rdr.FieldCount; i++)
            {
                //вывести в консольное окно имена полей
                Console.WriteLine(rdr.GetName(i) .
                    ToString() + " ");
            }
        }
        Console.WriteLine();
        line++;
        Console.WriteLine(rdr[1] + " " + rdr[2]);
    }
}
```

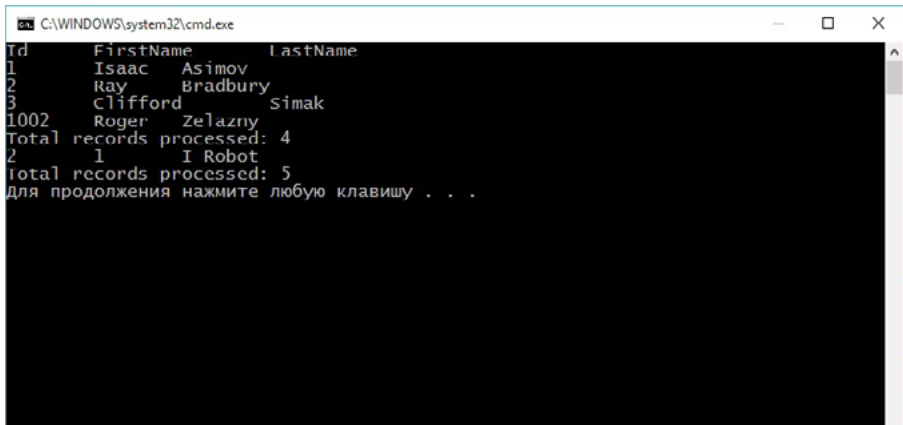


```

    }
    Console.WriteLine("Обработано записей:
        " + line.ToString());
} while (rdr.NextResult());

```

Теперь вы можете добавить к нашему запросу select еще один или несколько запросов, не забыв разделить их «;» и запустить приложение. В консольном окне вы увидите результаты, возвращенные всеми запросами.



```

C:\WINDOWS\system32\cmd.exe
Id      FirstName  LastName
1       Isaac     Asimov
2       Ray       Bradbury
3       Clifford  Simak
1002    Roger     Zelazny
Total records processed: 4
2       1         I Robot
Total records processed: 5
для продолжения нажмите любую клавишу . . .

```

Рис. 13. Пример пакетного выполнения запросов

Правда, при таком подходе есть одно ограничение. Если все запросы select, занесенные в пакет, возвращают одинаковое количество полей, тогда проблем не будет. Но если количество возвращаемых полей разное, то сначала надо выполнять запрос, возвращающий максимальное количество полей. И затем надо каким-либо образом решать вопросы с обращением к конкретным элементам полученного массива. Мы еще вернемся к этому вопросу во втором уроке в нашем следующем приложении.

Ниже приведен полный код метода, выполняющего сразу несколько запросов select.

```
public void ReadData2()
{
    SqlDataReader rdr = null;
    try
    {
        //Open the connection
        conn.Open();
        SqlCommand cmd = new SqlCommand("select *
            from Authors; select *
            from Books", conn);
        rdr = cmd.ExecuteReader();
        int line = 0;
        //извлечь полученные строки
        do
        {
            while (rdr.Read())
            {
                if (line == 0) //формируем шапку
                //таблицы перед выводом первой строки
                {
                    //цикл по числу прочитанных полей
                    for (int i = 0; i < rdr.
                        FieldCount; i++)
                    {
                        //Вывести в консольное окно
                        //имена полей
                        Console.Write(rdr.GetName(i).
                            ToString() + "\t");
                    }
                    Console.WriteLine();
                }
                line++;
                Console.WriteLine(rdr[0] + "\t" +
                    rdr[1] + "\t" + rdr[2]);
            }
        }
    }
}
```

```

    }
    Console.WriteLine("Total records
                      processed: " + line.ToString());
    } while (rdr.NextResult());
}
finally
{
    //close the reader
    if (rdr != null)
    {
        rdr.Close();
    }
    // Close the connection
    if (conn != null)
    {
        conn.Close();
    }
}
}

```

Конфигурационный файл

В нашем примере есть несколько недостатков, которые надо устранить. Одним из недостатков является хранение в коде данных для доступа к серверу. Вы понимаете, что мы говорим о строке подключения, которую мы храним в строковой переменной. Это недопустимо как с точки зрения безопасности, так и с той точки зрения, что при изменении адреса сервера, имени БД или другого какого-либо значения из строки подключения, нам придется вносить изменения в код и пересобирать наше приложение.

Каким может быть решение этой проблемы? Можно, например, предлагать пользователю вносить данные для

подключения в диалоговое окно. Такое решение является очень хорошим во многих случаях. Однако, стандартным способом решения этой проблемы считается хранение строки подключения в файле конфигурации приложения. Файл конфигурации приложения — это xml файл, автоматически добавляемый Visual Studio 2015 в состав приложения. В нашем распоряжении есть набор классов, позволяющих работать с файлом конфигурации приложения. Давайте познакомимся с тем, как хранить строку подключения в этом файле и извлекать из нее необходимую информацию.

По умолчанию файл конфигурации приложения выглядит таким образом:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <startup>
    <supportedRuntime version="v4.0"
      sku=".NETFramework,Version=v4.5.2" />
  </startup>
</configuration>
```

Строка подключения к БД в этом файле должна располагаться в элементе `connectionStrings` внутри элемента `configuration`. Измените файл конфигурации приложения, чтобы он выглядел таким образом:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <startup>
    <supportedRuntime version="v4.0"
      sku=".NETFramework,Version=v4.5.2" />
  </startup>
```

```
<connectionStrings>
  <add name="MyConnString"
        connectionString="Data Source=(localdb)\v11.0;
        Initial Catalog=Library;
        Integrated Security=SSPI;" />
</connectionStrings>
</configuration>
```

Для доступа к файлу конфигурации приложения из нашего кода мы будем пользоваться классом `ConfigurationManager`, определенным в пространстве имен `System.Configuration`. По умолчанию это пространство имен не добавляется в состав проекта. Поэтому надо выполнить команду `Add Link...` С помощью этого класса мы можем извлечь строку подключения в строковую переменную таким образом:

```
string connectionString = ConfigurationManager.
    ConnectionStrings["MyConnString"].ConnectionString;
```

Обратите внимание, что в качестве индекса для свойства `ConnectionStrings` мы указываем имя элемента `add`, в котором добавлена наша строка подключения. Поскольку одно приложение может работать с несколькими БД одновременно, в одном проекте может быть несколько строк подключения и каждая должна иметь уникальное имя в атрибуте `name`.

Давайте изменим конструктор нашего класса, в котором мы инициализировали объект `SqlConnection` и заносили в его свойство `ConnectionString` строку подключения:

```

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

```

Перестройте приложение и запустите его. Вы увидите в консольном окне результат работы метода `ReadData()`. Значит наше приложение работает, но при этом строка подключения уже находится вне кода приложения в конфигурационном файле.

Параметризованные запросы в `DbCommand`

Давайте вернемся к классу `DbCommand` и познакомимся с другими его очень полезными возможностями. Посмотрим на строки нашего метода `InsertQuery()`, в которых мы выполняем запрос `insert`.

```

//подготовить запрос insert в переменной типа string
string insertString = @"insert into Authors
(FirstName, LastName) values ('Roger', 'Zelazny')";

//создать объект command, инициализировав оба свойства
SqlCommand cmd = new SqlCommand(insertString, conn);

//выполнить запрос, занесенный в объект command
cmd.ExecuteNonQuery();

```

Вы понимаете, что передача данных для записи в таблицу в запросе `insert` в виде литералов — это плохое

решение. Надо реализовать какой-нибудь механизм, позволяющий пользователю передавать данные для ввода в БД динамически.

Пожалуйста, никогда не делайте чего-нибудь подобного этому:

```
string userName = // прочитать из консоли
                  //или из TextBox;
string sql = @"select * from users where name =
              '"+ userName +"'";
```

Такой стиль программирования — это огромная дыра в безопасности вашего приложения. Проверить данные, которые пользователь будет вносить в консоль или в элемент TextBox для записи в БД таким образом невозможно. Что произойдет, если пользователь занесет в TextBox вместо ожидаемого имени для userName такую строку:

```
"user5'; drop table users;'"?
```

Подставьте эту строку путем конкатенации в указанный запрос select. Вы получите такой запрос:

```
select * from users where name = 'user5';
drop table users;
```

Как вы понимаете, этот запрос, точнее уже эти два запроса удалят из вашей БД таблицу users. И таких хитрых запросов можно придумать много.

Нужно использовать другой механизм передачи данных в запрос, чтобы избежать таких неприятностей. И такой механизм есть. Этот механизм предоставлен нам классом DbCommand. В этом классе есть свойство

с именем `Parameters`. По имени свойства во множественном числе понятно, что это свойство является коллекцией. Элементы этой коллекции являются объектами типа `DbParameter`, точнее — типа производного от него, например, `SqlParameter`.

Класс `DbParameter` позволяет нам создавать параметризованные запросы к БД, обеспечивая безопасность таких запросов. Давайте перепишем наш запрос `select` в виде параметризованного запроса.

```
//подготовить запрос select в переменной типа string
string sql = @"select * from users where name = @p1";
```

В этом коде `@p1` представляет собой параметр нашего запроса. Признаком параметра является символ “@”. Имена параметров и их количество есть произвольными. Теперь мы должны для каждого указанного в запросе параметра создать объект типа `SqlParameter` и занести этот объект в свойство `Parameters` объекта `SqlCommand`. Есть несколько способов создания объекта `SqlParameter`. Рассмотрим эти способы.

```
SqlParameter param1 = new SqlParameter();
param1.ParameterName = "@p1"; //сопоставление с параметром
                               //в запросе
param.SqlDbType = System.Data.SqlDbType.NVarChar;
                               //тип параметра
param1.Value = firstName; //значение параметра
```

Созданный таким образом объект `param1` теперь можно добавить в коллекцию `Parameters` объекта `SqlCommand`.

```
cmd.Parameters.Add(param1);
```


Аналогично надо создавать и заносить в коллекцию `Parameters` объект для каждого параметра, указанного в запросе. Обратите внимание на большой выбор типов данных в перечислении `SqlDbType`, это необходимо для максимально точного соответствия между типами данных .NET Framework и типами данных MS SQL Server. Некоторые другие свойства объекта `SqlParameter` мы рассмотрим в следующем разделе урока, где будем рассматривать работу с хранимыми процедурами.

Приведенный выше пример определения параметров для запросов может показаться громоздким. Это действительно так. Если кто-то предпочитает более краткие формы кодирования, в данном случае я могу вас порадовать. Дело в том, что метод `Add()`, определенный в коллекции `Parameters`, обладает одной интересной особенностью. Он возвращает добавленный в коллекцию элемент. Если воспользоваться этой особенностью метода `Add()` и использовать его перегруженные варианты, то можно написать более короткий вариант кода:

```
cmd.Parameters.Add("@p1", SqlDbType.NVarChar).  
Value = firstName;
```

Эта одна строка кода заменяет четыре развернутые строки, которые мы рассматривали перед этим. В этой строке, благодаря использованию перегруженного метода `Add()`, «на лету» создается новый объект `SqlParameter`. При создании в этом объекте инициализируются свойства `ParameterName` и `DbType`. А затем инициализируется

и свойство Value. Таким образом, в одной строке инициализируются все необходимые свойства.

Существуют еще другие виды методов Add(), например AddWithValue(), который сразу инициализирует свойство Value:

```
cmd.Parameters.AddWithValue("@p1", firstName);
```

Помните, мы говорили о том, что динамически создавать запрос, конкатенируя строку с параметрами — это плохая идея? Так вот, при использовании SqlParameter в значениях параметров выполняется автоматическая проверка на наличие опасного содержимого. Если какой-либо хакер решит передать в ваше приложение вредный скрипт, этот скрипт не активируется. И хакер будет разочарован. Что же произойдет с «хитрым» запросом приведенным выше, если параметр для него передавать с помощью SqlParameter? Он превратится в такой запрос:

```
select * from users where name = 'user5';  
drop table users;
```

В этом запросе, все, что расположено после «=» интерпретируется, как строковая константа и никакого вреда вашей БД этот запрос не причинит.

Хранимые процедуры в SqlCommand

Мы можем использовать в наших приложениях не только запросы, а и хранимые процедуры. Рассмотрим применение хранимых процедур в технологии ADO.NET. Прежде всего, нам надо создать какую-либо хранимую процедуру. Желательно с параметрами. А еще лучше — с входными

и выходными параметрами! Вы же не забыли, что параметры могут быть выходными? Давайте создадим хранимую процедуру с двумя параметрами. В первом параметре процедура будет принимать id автора, а через второй параметр — возвращать количество книг этого автора в таблице Books. Для создания хранимой процедуры в нашей БД перейдите снова в окно SQL Server Object Explorer, разверните узел нашей БД, дальше разверните узел Programmability, дальше выделите узел Stored Procedures. Из контекстного меню активируйте опцию Add New Stored Procedure...

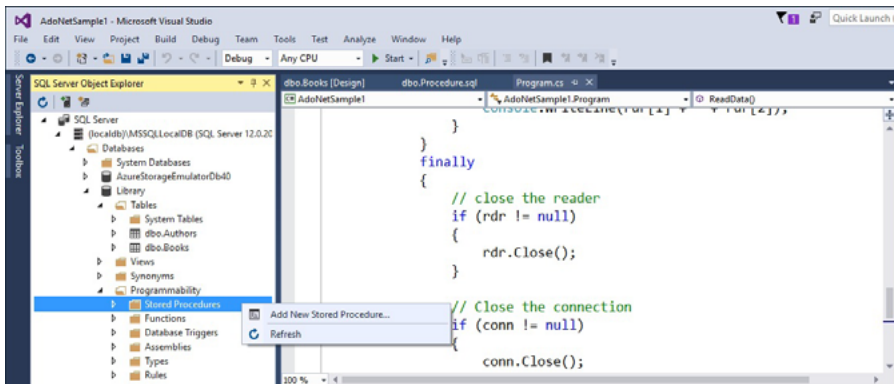


Рис. 14. Добавление новой хранимой процедуры

В правом окне появится заготовка текста хранимой процедуры. Удалите ее и вместо нее вставьте код нашей хранимой процедуры:

```
CREATE PROCEDURE getBooksNumber
@AuthorId int,
@BookCount int OUTPUT
AS
BEGIN
```

```

SET NOCOUNT ON;
SELECT @BookCount = count(b.id)
FROM Books b, Authors a
WHERE b.Authorid = a.id AND
a.id = @ AuthorId;

END;

```

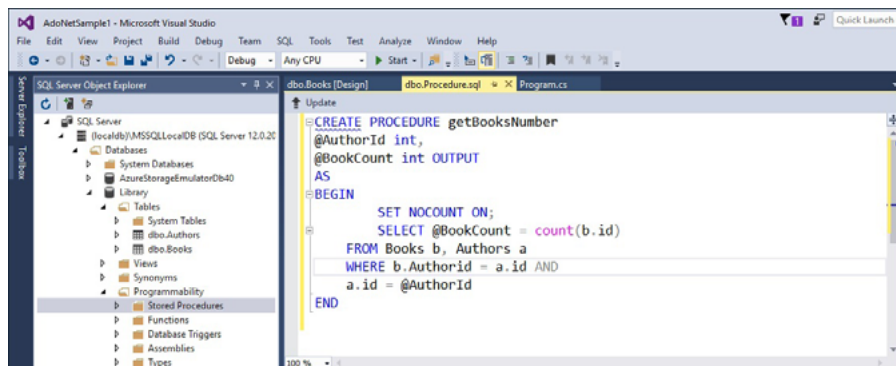


Рис. 15 Создание новой хранимой процедуры

Мы создаем хранимую процедуру с именем `getBooksNumber`. Через первый параметр `@AuthorId` типа `int` мы будем при вызове передавать в процедуру `id` автора. Второй параметр `@BookCount` типа `int` описан как `output`. При вызове процедуры мы будем подставлять в него неинициализированный параметр. После того, как процедура будет вызвана и выполнена мы сможем прочитать из второго параметра занесенное туда значение.

После вставки кода создаваемой хранимой процедуры нажмите кнопку `Update`, расположенную на верхней панели. В появившемся после этого окне нажмите кнопку `Update Database`. Теперь новая хранимая процедура добавлена в нашу БД. Давайте посмотрим, как мы можем вызывать хранимые процедуры из кода `C#`.

Если раньше в свойство `CommandText` объекта `SqlCommand` мы заносили текст запроса, то в случае, когда мы хотим выполнить хранимую процедуру, в свойство `CommandText` надо занести только имя хранимой процедуры:

```
SqlCommand cmd = new SqlCommand("getBooksNumber", conn);
```

Однако, в этом случае нам надо задействовать еще одно свойство, которое мы не использовали раньше. Мы должны указать, что вызываем именно хранимую процедуру. Для этого надо инициализировать свойство `CommandType` объекта `SqlCommand`. По умолчанию это свойство содержит ссылку на запрос, а нам надо занести в него значение `StoredProcedure`:

```
cmd.CommandType = CommandType.StoredProcedure;
```

Поскольку у нашей хранимой процедуры есть параметры, их надо правильно описать и передать в объект `SqlCommand`. С первым параметром все традиционно. Указываем имя, тип, значение и заносим в коллекцию `Parameters`:

```
cmd.Parameters.Add("@AuthorId", System.Data.  
    SqlDbType.Int).Value = 1;
```

Со вторым параметром, который у нас выходной, ситуация обстоит иначе. Его надо создать в отдельной переменной типа `SqlParameter`. Указать для него имя и тип. Значение (свойство `Value`) указывать не надо. Но для этого параметра необходимо заполнить свойство `Direction`,

в котором надо указать, что этот параметр выходной. Затем созданный параметр надо занести в коллекцию `Parameters`.

```
SqlParameter outputParam = new SqlParameter("@  
    BookCount", System.Data.SqlDbType.Int);  
outputParam.Direction = ParameterDirection.Output;  
//outputParam.Value = 0; //заполнять Value не надо!  
cmd.Parameters.Add(outputParam);
```

Соберем все это вместе и получим код метода `ExecStoredProcedure()`, предназначенного для вызова нашей хранимой процедуры. Обратите внимание, как в этом коде извлекается значение из выходного параметра после выполнения процедуры.

```
public void ExecStoredProcedure()  
{  
    conn.Open();  
    SqlCommand cmd = new SqlCommand("getBooksNumber",  
        conn);  
    cmd.CommandType = CommandType.StoredProcedure;  
    cmd.Parameters.Add("@AuthorId", System.Data.  
        SqlDbType.Int).Value = 1;  
  
    SqlParameter outputParam = new SqlParameter("@  
        BookCount", System.Data.SqlDbType.Int);  
    outputParam.Direction = ParameterDirection.  
        Output;  
    //outputParam.Value = 0; //заполнять Value не надо!  
    cmd.Parameters.Add(outputParam);  
  
    cmd.ExecuteNonQuery();  
    Console.WriteLine(cmd.Parameters["@BookCount"].  
        Value.ToString());  
}
```

Подведем итог нашему знакомству с ADO.NET. Вы создали приложение, в котором рассмотрели применение основных классов, входящих в этот фреймворк:

- *DbConnection* — для подключения к БД;
- *DbCommand* — для выполнения в БД запросов, хранимых процедур и для получения результатов из БД;
- *DbDataReader* — для извлечения результатов выполнения запросов *select*;
- *DbParameter* — для создания параметризованных обращений к БД;

Кроме этого, вы познакомились с использованием конфигурационного файла. Все, что мы делали в этом уроке, базировалось на применении присоединенного режима работы с сервером.

В следующем уроке мы рассмотрим еще другие полезные классы и перейдем к изучению отсоединенного режима работы с сервером.

Домашнее задание

Для нашей БД надо написать консольное приложение, которое должно вычислить сумму цен всех книг и сумму страниц всех книг в таблице Books. Но, сначала надо узнать количество книг в этой таблице, для чего предлагается выполнить такой запрос:

```
Select count(id) from Books;
```

Подумайте, каким из методов класса DbCommand (ExecuteNonQuery(), ExecuteScalar() или ExecuteReader()) удобнее всего выполнить этот запрос. Сохраните значение, возвращенное этим запросом в какой-либо int num переменной.

Затем выполните запрос

```
Select * from Books;
```

Для разнообразия, не повторяйте код, написанный нами в этом уроке, и результаты этого запроса извлекайте в цикле for, используя переменную num, инициализированную первым запросом, как ограничитель цикла. На каждой итерации этого цикла for, извлекайте значения из полей Price и Pages, как int значения (посмотрите, какие методы для этого есть у класса DbDataReader) и суммируйте их. Значения остальных полей извлекайте согласно их типу в БД. Еще раз внимательно посмотрите, какие методы в классе DbDataReader, позволяют получить

прочитанное из таблицы значение и преобразовать его к типу `string`, или `int`, или `double`. Выводите в консоль значения всех полей на каждой итерации. После окончания цикла выведите суммарные значения цен для всех книг и количества страниц для всех книг.

Цель домашнего задания:

- повторить рассмотренные в уроке действия;
- научиться извлекать из полей таблицы данные разных типов;
- использовать не только `ExecuteQuery()` метод класса `DbCommand`.