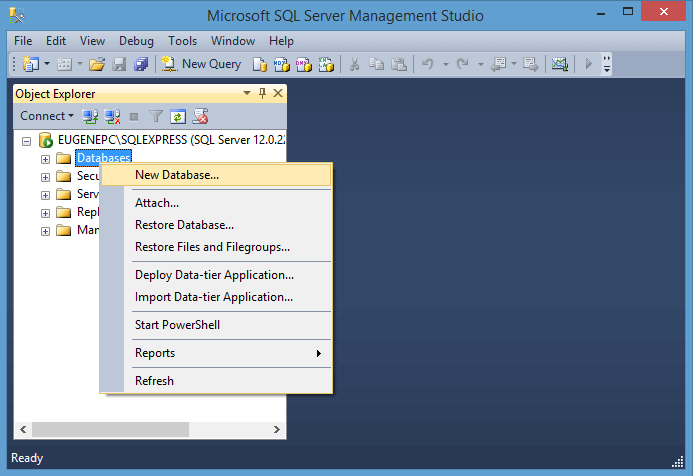
**C# и MS SQL Server**

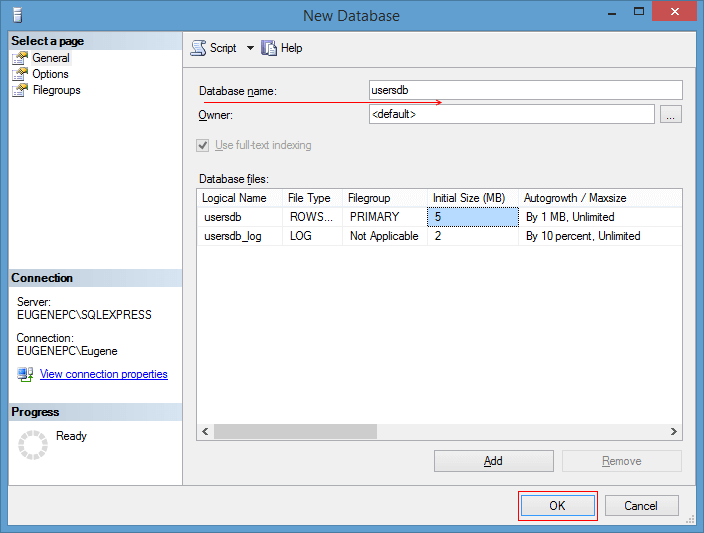
**Создание базы данных**

Для работы с базами данных нам естественно прежде всего надо иметь какую-нибудь базу данных. В данном случае мы будем рассматривать основные концепции ADO.NET преимущественно на примере MS SQL Serverа. Поэтому вначале нам надо установить SQL Server Express. Все необходимые материалы для установки можно найти по адресу <https://www.microsoft.com/en-US/download/details.aspx?id=42299>. Вместе с сервером также устанавливается специальная программа SQL Server Management Studio, которая используется для управления базами данных на сервере.

Для начала создадим простейшую базу данных на MS SQL Server. Для этого откроем SQL Server Management Studio и нажмем на узел Databases правой кнопкой мыши. После этого в открывшемся контекстном меню выберем **New Database**:

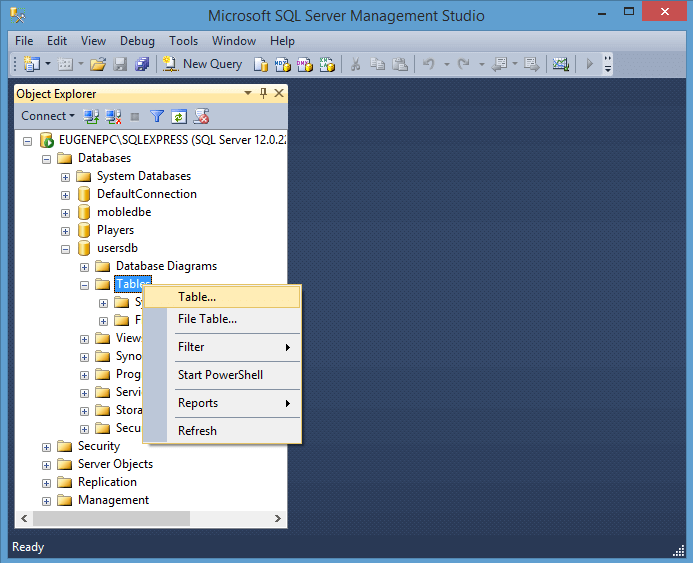


После этого нам открывается окно для создания базы данных:

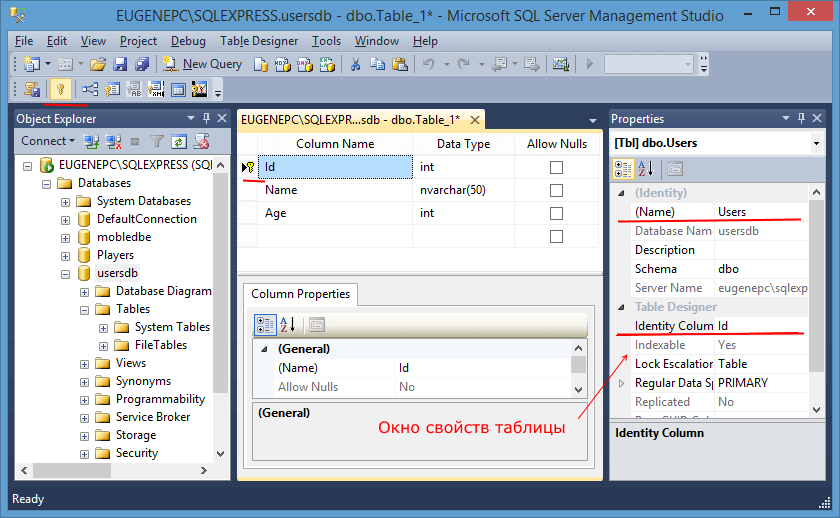


В нем в поле Database Name нам надо ввести название базы данных. Введем **usersdb**. Больше здесь вводить ничего не требуется, и поэтому нажмем на ОК.

После этого в узле Databases появляется новый элемент, который представляет только что созданную базу данных usersdb. Раскроем его и нажмем правой кнопкой мыши на его подузел Tables:



Затем нам открывается дизайнер таблицы:

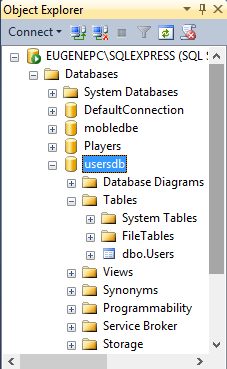


В нем надо указать три столбца: Id, Name и Age, которые будут представлять соответственно уникальный идентификатор пользователя, его имя и возраст. У первого и третьего столбца надо указать тип int (то есть целочисленный), а у столбца Name - тип nvarchar (строковый).

Кроме того, в окне свойств таблицы в поле Name надо ввести имя таблицы - Users, а в поле Identity ввести Id, то есть тем самым указывая, что столбец Id будет идентификатором.

И в конце нам надо установить курсор на столбец Id и в панели инструментов программы нажать на золотой ключик. После этого напротив поля Id также должен появиться золотой ключик. Этот ключик будет указывать, что столбец Id будет выполнять роль первичного ключа.

После этого нажмем на сохранение и затем на клавишу F5 (обновление), и в узле нашей базы данных появится новая таблица, которая будет называться dbo.Users:



Итак, мы создали базу данных и таблицу, которые будут выполнять функции по хранению данных. И теперь мы можем использовать инфраструктуру ADO.NET для подключения к ним.

**Строка подключения**

После определения источника данных мы можем к нему подключаться. Для этого создадим проект простого консольного приложения.

Первым делом нам надо определить строку подключения, предоставляющую информацию о базе данных и сервере, к которым предстоит установить подключение:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | class Program  {      static void Main(string[] args)      {          string connectionString = @"Data Source=.\SQLEXPRESS;Initial Catalog=usersdb;Integrated Security=True";      }  } |

При использовании различных систем управления базами данных, различных провайдеров данных .NET строка подключения может отличаться. Даже для подключения одной и той же базы данных строка подключения может меняться в зависимости от обстоятельств.

Строка подключения представляет набор параметров в виде пар ключ=значение. В данном случае для подключения к ранее созданной базе данных usersdb мы определяем строку подключения из трех параметров:

* Data Source: указывает на название сервера. По умолчанию это ".\SQLEXPRESS". Поскольку в строке используется слеш, то в начале строки ставится символ @. Если имя сервера базы данных отличается, то соответственно его и надо использовать.
* Initial Catalog: указывает на название базы данных на сервере
* Integrated Security: устанавливает проверку подлинности

Жесткое кодирование строки подключения (то есть ее определение в коде приложения), как правило, редко используется. Гораздо более гибкий путь представляет определение ее в специальных конфигурационных файлах приложения. В проектах десктопных приложений это файл **App.config**, а в веб-приложениях это в основном файл **Web.config**. Хотя приложение также может использовать другие способы определения конфигурации.

В нашем случае, так как мы создали проект консольного приложения, то у нас в проекте должен быть файл *App.config*, который на данный момент имеет следующее определение:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <?xml version="1.0" encoding="utf-8" ?>  <configuration>      <startup>          <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.6" />      </startup>  </configuration> |

Изменим его, добавив определение строки подключения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | <?xml version="1.0" encoding="utf-8" ?>  <configuration>      <startup>          <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.6" />      </startup>    <connectionStrings>     <add name="DefaultConnection" connectionString="Data Source=.\SQLEXPRESS;Initial Catalog=usersdb;Integrated Security=True"         providerName="System.Data.SqlClient"/>    </connectionStrings>  </configuration> |

Для определения всех подключений в программе в пределах узла <configuration> добавляется новый узел <connectionStrings>. В этом узле определяются строки подключения с помощью элемента <add>. Мы можем использовать в приложении множество строк подключения, и соответственно также в файле можно определить множество элементов <add>.

Каждая строка подключения имеет название, определяемое с помощью атрибута name. В данном случае строка подключения называется "DefaultConnection". Название может быть произвольное.

Атрибут connectionString собственно хранит строку подключения, то есть весь тот текст, который мы выше определяли в методе Main. И третий атрибут providerName задает пространство имен провайдера данных. Так как мы будем подключаться к базе данных MS SQL Server, то соответственно мы будем использовать провайдер для SQL Server, функциональность которого заключена в пространстве имен System.Data.SqlClient.

Теперь получим эту строку подключения в приложении:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | using System;  using System.Configuration;    namespace AdoNetConsoleApp  {      class Program      {          static void Main(string[] args)          {              //string connectionString = @"Data Source=.\SQLEXPRESS;Initial Catalog=usersdb;Integrated Security=True";              // получаем строку подключения              string connectionString = ConfigurationManager.ConnectionStrings["DefaultConnection"].ConnectionString;              Console.WriteLine(connectionString);                Console.Read();          }      }  } |

Прежде всего чтобы работать с конфигурацией приложения, нам надо добавить в проект библиотеку **System.Configuration.dll**.

С помощью объекта ConfigurationManager.ConnectionStrings["название\_строки\_подключения"] мы можем получить строку подключения и использовать ее в приложении.

### Параметры строки подключения

* **Application Name**: название приложения. Может принимать в качестве значения любую строку. Значение по умолчанию: ".Net SqlClient Data Provide"
* **AttachDBFileName**: хранит полный путь к прикрепляемой базе данных
* **Connect Timeout**: временной период в секундах, через который ожидается установка подключения. Принимает одно из значений из интервала 0–32767. По умолчанию равно 15.

В качестве альтернативного названия параметра может использоваться **Connection Timeout**

* **Data Source**: название экземпляра SQL Servera, с которым будет идти взаимодействие. Это может быть название локального сервера, например, "EUGENEPC/SQLEXPRESS", либо сетевой адрес.

В качестве альтернативного названия параметра можно использовать **Server**, **Address**, **Addr** и **NetworkAddress**

* **Encrypt**: устанавливает шифрование SSL при подключении. Может принимать значения true, false, yes и no. По умолчанию значениеfalse
* **Initial Catalog**: хранит имя базы данных

В качестве альтернативного названия параметра можно использовать **Database**

* **Integrated Security**: задает режим аутентификации. Может принимать значения true, false, yes, no и sspi. По умолчанию значениеfalse

В качестве альтернативного названия параметра может использоваться **Trusted\_Connection**

* **Packet Size**: размер сетевого пакета в байтах. Может принимать значение, которое кратно 512. По умолчанию равно 8192
* **Persist Security Info**: указывает, должна ли конфиденциальная информация передаваться обратно при подключении. Может принимать значения true, false, yes и no. По умолчанию значение false
* **Workstation ID**: указывает на рабочую станцию - имя локального компьютера, на котором запущен SQL Server
* **Password**: пароль пользователя
* **User ID**: логин пользователя

Например,если для подключения необходим логин и пароль, то мы можем их передать в строку подключения через параметры user id иpassword:

|  |  |
| --- | --- |
| 1 | string connectionString = @"Data Source=.\SQLEXPRESS;Initial Catalog=usersdb;User Id = sa; Password = 1234567fd";"; |

**Создание подключения**

Чтобы подключаться к базе данных, нам необходимо создать и использовать объект SqlConnection:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | using System;  using System.Data.SqlClient;    namespace AdoNetConsoleApp  {      class Program      {          static void Main(string[] args)          {              string connectionString = @"Data Source=.\SQLEXPRESS;Initial Catalog=usersdb;Integrated Security=True";                // Создание подключения              SqlConnection connection = new SqlConnection(connectionString);              try              {                  // Открываем подключение                  connection.Open();                  Console.WriteLine("Подключение открыто");              }              catch (SqlException ex)              {                  Console.WriteLine(ex.Message);              }              finally              {                  // закрываем подключение                  connection.Close();                  Console.WriteLine("Подключение закрыто...");              }                Console.Read();          }      }  } |

В конструктор объекту SqlConnection передается строка подключения, которая инициализирует объект. Чтобы использовать этот объект и подключаться к базе данных, мы должны выполнить его метод Open(), а после завершения работы с базой данных нам надо вызвать методClose() для закрытия подключения. В случае возникновения ошибок закрытие подключения происходит в блоке finally.

В качестве альтернативного метода можно использовать конструкцию using, которая автоматически закрывает подключение:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | static void Main(string[] args)  {      string connectionString = @"Data Source=.\SQLEXPRESS;Initial Catalog=usersdb;Integrated Security=True";        using (SqlConnection connection = new SqlConnection(connectionString))      {          connection.Open();          Console.WriteLine("Подключение открыто");      }      Console.WriteLine("Подключение закрыто...");        Console.Read();  } |

### Асинхронное подключение

Для создания асинхронного подключения необходимо использовать асинхронный метод **OpenAsync()**:

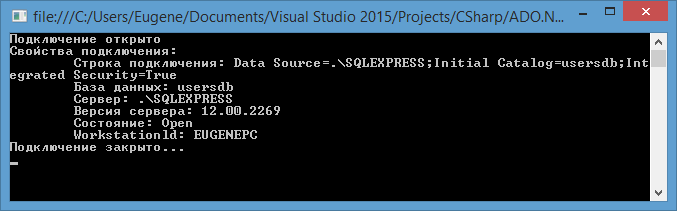
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | class Program  {      static void Main(string[] args)      {              ConnectWithDB().GetAwaiter();      }        private static async Task ConnectWithDB()      {          string connectionString = @"Data Source=.\SQLEXPRESS;Initial Catalog=usersdb;Integrated Security=True";            using (SqlConnection connection = new SqlConnection(connectionString))          {              await connection.OpenAsync();              Console.WriteLine("Подключение открыто");          }          Console.WriteLine("Подключение закрыто...");      }  } |

### Получение информации о подключении

Объект SqlConnection обладает рядом свойств, которые позволяют получить информацию о подключении:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | string connectionString = @"Data Source=.\SQLEXPRESS;Initial Catalog=usersdb;Integrated Security=True";  using (SqlConnection connection = new SqlConnection(connectionString))  {      connection.Open();      Console.WriteLine("Подключение открыто");        // Вывод информации о подключении      Console.WriteLine("Свойства подключения:");      Console.WriteLine("\tСтрока подключения: {0}", connection.ConnectionString);      Console.WriteLine("\tБаза данных: {0}", connection.Database);      Console.WriteLine("\tСервер: {0}", connection.DataSource);      Console.WriteLine("\tВерсия сервера: {0}", connection.ServerVersion);      Console.WriteLine("\tСостояние: {0}", connection.State);      Console.WriteLine("\tWorkstationld: {0}", connection.WorkstationId);  }    Console.WriteLine("Подключение закрыто..."); |

Консольный вывод в данном случае:



### Возможные ошибки и их решение

При подключении к базе данных может столкнуться с рядом ошибок, одной из наиболее распространенных является следующая:

Unhandled Exception: System.ArgumentException: Keyword not supported...

Данная ошибка вызвана из-за неправильного указания параметров строки подключения, причем неважно, как мы определяем строку подключения - в коде или в файле конфигурации. В сообщении об ошибке, как привило, говорится, какой параметр указан неправильно, и в этом случае его надо исправить.

Еще одна распространенная ошибка говорит о том, что данной базы нет или по крайней мере нет подобной базы данных для текущего пользователя:

Cannot open database "название базы данных" requested by the login. The login failed.

Login failed for user 'название\_пользователя'

В этом случае надо убедиться, что на сервере есть база данных с таким названием, а если есть, то проверить, есть ли доступ для данного пользователя к этой бд.

И еще одна часто встречающаяся ошибка:

A network-related or instance-specific error occurred while establishing a connection to SQL Server.

The server was not found or was not accessible...

В этом случае, как правило, SQL Server не запущен. И его надо запустить или перезапустить, через панель служб.

Также может быть, что в строке подключения указано неправильное имя сервера, либо MS SQL Server вообще не установлен.

**Пул подключений**

Как правило, в программе используется одна или несколько одних и тех же конфигураций подключений. И чтобы разработчику не приходилось создавать по нескольку раз в коде программы фактически одно и тоже подключение, в ADO.NET используется механизм **пула подключений**. К тому же сама по себе операция создания нового объекта подключений является довольно затратной, и использование пула позволяет оптимизировать производительность приложения.

Пул подключений позволяет использовать ранее созданные подключения. Когда менеджер подключений, который управляет пулом, получает запрос на открытие нового подключения с помощью метода Open(), то он проверяет все подключения пула.

Если менеджер подключений находит в пуле доступное подключение, которое в текущий момент не используется, то оно возвращается для использования. Если же доступного подключения нет, и максимальный размер пула еще не превышен (по умолчанию размер равен 100), то создается новое подключение. Если доступного подключения нет, но при этом превышен максимальный размер пула, то новое подключение добавляется в очередь и ожидает, пока в пуле не освободится место, и тогда оно станет доступным.

После закрытия подключения с помощью метода Close() закрытое подключение возвращается в пул подключений, где оно оно готово к повторному использованию при следующем вызове метода Open().

Например, несмотря на закрытия подключения программа в обоих случаях будет использовать одно и то же подключение:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | SqlConnection connection;  connection = new SqlConnection(connectionString);    connection.Open();  Console.WriteLine(connection.ClientConnectionId);  connection.Close();    connection.Open();  Console.WriteLine(connection.ClientConnectionId);  connection.Close(); |

В пул помещаются подключения только с одинаковой конфигурацией. ADO.NET поддерживает несколько пулов одновременно, и для каждой конфигурации строки подключения создается свой собственный пул.

Все подключения в пуле различаются по нескольким признакам:

* строка подключения
* учетные записи, используемые при подключении
* процесс приложения

В следующем примере в первых двух блоках using будет использоваться одно и то же подключение из пула, поскольку строка подключения будет совпадать:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | static string connectionString = @"Data Source=.\SQLEXPRESS;Initial Catalog=usersdb;Integrated Security=True";  static string connectionString2 = @"Data Source=.\SQLEXPRESS;Initial Catalog=players;Integrated Security=True";  using (SqlConnection connection = new SqlConnection(connectionString))  {      connection.Open(); // создается первый пул      Console.WriteLine(connection.ClientConnectionId);  }  using (SqlConnection connection = new SqlConnection(connectionString))  {      connection.Open(); // подключение извлекается из первого пула      Console.WriteLine(connection.ClientConnectionId);  }  using (SqlConnection connection = new SqlConnection(connectionString2))  {      connection.Open(); // создается второй пул, т.к. строка подключения отличается      Console.WriteLine(connection.ClientConnectionId);  } |

Если параметр Min Pool Size не указан в строке подключения или его значение равно 0, то подключения в пуле будут закрыты после периода отсутствия активности (4-8 минут), либо если разорвана связь с сервером базы данных. Но если значение параметра Min Pool Size больше 0, пул подключений не удаляется, пока не будет выгружен домен приложения AppDomain и не завершится процесс.

**Выполнение команд и SqlCommand**

После установки подключения мы можем выполнить к базе данных какие-либо команды, например, добавить в базу данных объект, удалить, изменить его или просто извлечь. Команды представлены объектом интерфейса System.Data.IDbCommand. Провайдер для MS SQL предоставляет его реализацию в виде класса **SqlCommand**. Этот класс инкапсулирует sql-выражение, которое должно быть выполнено.

Для выполнения команды нам потребуется sql-выражение и объект подключения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | string connectionString = @"Data Source=.\SQLEXPRESS;Initial Catalog=usersdb;Integrated Security=True";  using (SqlConnection connection = new SqlConnection(connectionString))  {      connection.Open();      SqlCommand command = new SqlCommand();      command.CommandText = "SELECT \* FROM Users";      command.Connection = connection;  } |

С помощью свойства CommandText устанавливается SQL-выражение, которое будет выполняться. В данном случае это запрос на получение всех объектов из таблицы Users. А с помощью свойства Connection можно установить объект подключения SqlConnection.

В качестве альтернативы можно было бы использовать одну из версий конструктора класса:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | string connectionString = @"Data Source=.\SQLEXPRESS;Initial Catalog=usersdb;Integrated Security=True";  string sqlExpression = "SELECT \* FROM Users";  using (SqlConnection connection = new SqlConnection(connectionString))  {      connection.Open();      SqlCommand command = new SqlCommand(sqlExpression, connection);  } |

Чтобы выполнить команду, необходимо применить один из методов SqlCommand:

* **ExecuteNonOuery**: просто выполняет sql-выражение и возвращает количество измененных записей. Подходит для sql-выражений INSERT, UPDATE, DELETE.
* **ExecuteReader**: выполняет sql-выражение и возвращает строки из таблицы. Подходит для sql-выражения SELECT.
* **ExecuteScalar**: выполняет sql-выражение и возвращает одно скалярное значение, например, число. Подходит для sql-выражения SELECT в паре с одной из встроенных функций SQL, как например, Min, Max, Sum, Count.

### Добавление объектов

Выполним команду по добавлению одного объекта в таблицу Users базы данных usersdb, которая ранее была создана:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | class Program  {      static void Main(string[] args)      {          string connectionString = @"Data Source=.\SQLEXPRESS;Initial Catalog=usersdb;Integrated Security=True";          string sqlExpression = "INSERT INTO Users (Name, Age) VALUES ('Tom', 18)";            using (SqlConnection connection = new SqlConnection(connectionString))          {              connection.Open();              SqlCommand command = new SqlCommand(sqlExpression, connection);              int number = command.ExecuteNonQuery();              Console.WriteLine("Добавлено объектов: {0}", number);          }          Console.Read();      }  } |

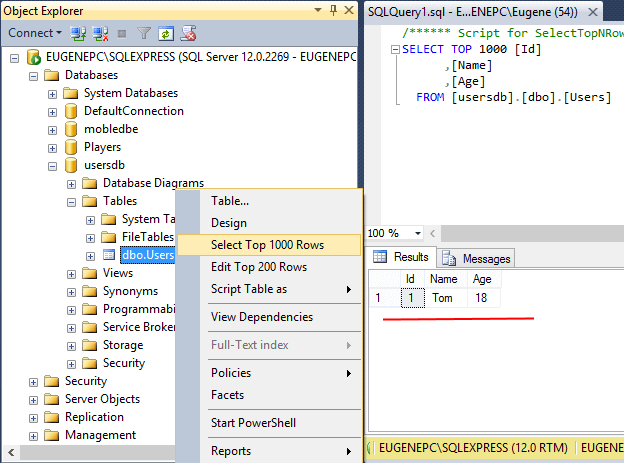
Для вставки объекта используется sql-выражение INSERT, которое имеет следующий синтаксис:

|  |  |
| --- | --- |
| 1 | INSERT INTO название\_таблицы (столбец1, столбец2, столбецN) VALUES ( значение1, значение2, значениеN) |

В данном случае мы знаем, что в базе данных у нас есть таблица Users, в которой есть три столбца - Id и Age, хранящие целое число, и Name, хранящий строку. Поэтому соответственно мы добавляем для столбца Name значение 'Tom', а для столбца Age число 18.

Здесь метод ExecuteNonOuery() возвращает число затронутых строк (в данном случае добавленных в таблицу объектов). Хотя нам необязательно возвращать результат метода, но данный результат может использоваться в качестве проверки, что операция, в частности, добавление, прошла успешно.

Чтобы убедиться, что данные добавлены, мы можем перейти к таблице Users в SQL Server Management Studio и с помощью опции Select Top 1000 Rows вывести добавленную строку:



### Обновление объектов

Обновление будет происходить аналогично, только теперь будет использоваться sql-выражение UPDATE, которое имеет следующий синтаксис:

|  |  |
| --- | --- |
| 1  2  3 | UPDATE название\_таблицы  SET столбец1=значение1, столбец2=значение2, столбецN=значениеN  WHERE некоторый\_столбец=некоторое\_значение |

Применим это выражение:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | string connectionString = @"Data Source=.\SQLEXPRESS;Initial Catalog=usersdb;Integrated Security=True";    string sqlExpression = "UPDATE Users SET Age=20 WHERE Name='Tom'";  using (SqlConnection connection = new SqlConnection(connectionString))  {      connection.Open();      SqlCommand command = new SqlCommand(sqlExpression, connection);      int number = command.ExecuteNonQuery();      Console.WriteLine("Обновлено объектов: {0}", number);  } |

Здесь обновляется строка, в которой Name=Tom, то есть выше добавленный объект. Если в таблице будет несколько строк, у которых Name=Tom, то обновятся все эти строки.

### Удаление

Удаление производится с помощью sql-выражения DELETE, которое имеет следующий синтаксис:

|  |  |
| --- | --- |
| 1  2 | DELETE FROM таблица  WHERE столбец = значение |

Удалим, например, всех пользователей, у которых имя Tom:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | string connectionString = @"Data Source=.\SQLEXPRESS;Initial Catalog=usersdb;Integrated Security=True";  string sqlExpression = "DELETE Users WHERE Name='Tom'";  using (SqlConnection connection = new SqlConnection(connectionString))  {      connection.Open();      SqlCommand command = new SqlCommand(sqlExpression, connection);      int number = command.ExecuteNonQuery();      Console.WriteLine("Удалено объектов: {0}", number);  } |

Во всех трех случаях фактически меняется только sql-выражение, а остальная логика остается неизменной. И мы также можем выполнять сразу несколько операций:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | static void Main(string[] args)  {      string connectionString = @"Data Source=.\SQLEXPRESS;Initial Catalog=usersdb;Integrated Security=True";        Console.WriteLine("Введите имя:");      string name = Console.ReadLine();        Console.WriteLine("Введите возраст:");      int age = Int32.Parse(Console.ReadLine());        string sqlExpression = String.Format("INSERT INTO Users (Name, Age) VALUES ('{0}', {1})", name, age);      using (SqlConnection connection = new SqlConnection(connectionString))      {          connection.Open();          // добавление          SqlCommand command = new SqlCommand(sqlExpression, connection);          int number = command.ExecuteNonQuery();          Console.WriteLine("Добавлено объектов: {0}", number);            // обновление ранее добавленного объекта          Console.WriteLine("Введите новое имя:");          name = Console.ReadLine();          sqlExpression = String.Format("UPDATE Users SET Name='{0}' WHERE Age={1}", name, age);          command.CommandText = sqlExpression;          number = command.ExecuteNonQuery();          Console.WriteLine("Обновлено объектов: {0}", number);      }      Console.Read();  } |

Консольный вывод:

Введите имя:

Tom

Введите возраст:

41

Добавлено объектов: 1

Введите новое имя:

Alex

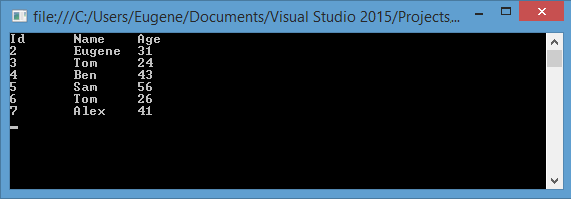
Обновлено объектов: 1

**Чтение результатов запроса и SqlDataReader**

В прошлой теме мы рассмотрели, как выполнять команды с помощью метода ExecuteNonOuery(), однако если мы хотим считывать данные, которые хранятся в таблице, то нам потребуется другой метод - ExecuteReader(). Этот метод возвращает объект **SqlDataReader**, который используется для чтения данных. Так, получим все данные из таблицы Users и выведем их на консоль:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | static void Main(string[] args)  {      string connectionString = @"Data Source=.\SQLEXPRESS;Initial Catalog=usersdb;Integrated Security=True";        string sqlExpression = "SELECT \* FROM Users";      using (SqlConnection connection = new SqlConnection(connectionString))      {          connection.Open();          SqlCommand command = new SqlCommand(sqlExpression, connection);          SqlDataReader reader = command.ExecuteReader();            if(reader.HasRows) // если есть данные          {              // выводим названия столбцов              Console.WriteLine("{0}\t{1}\t{2}", reader.GetName(0), reader.GetName(1), reader.GetName(2));                while (reader.Read()) // построчно считываем данные              {                  object id = reader.GetValue(0);                  object name = reader.GetValue(1);                  object age = reader.GetValue(2);                    Console.WriteLine("{0} \t{1} \t{2}", id, name, age);              }          }            reader.Close();      }        Console.Read();  } |

Консольный вывод:



Для выборки данных из БД используется sql-выражение SELECT. В данном случае мы выбираем все столбцы всех строк таблицы. Получив при выполнении запроса объект SqlDataReader, мы можем считать все полученные данные.

Но вначале мы проверяем, а есть ли вообще данные с помощью свойства HasRows. Если данные есть, то выводим заголовки таблицы с помощью методов reader.GetName(). Причем мы получаем столбцы в выборке именно в том порядке, в котором они определены в таблицы. То есть если вторым в таблицы идет столбец "Name", то чтобы получить его столбец применяется метод GetName(1) (так как нумерация столбцов идет с нуля).

Далее считываем сами данные. С помощью метода reader.Read() ридер переходит к следующей строке и возвращает булевое значение, которое указывает, есть ли данные для считывания.

В цикле while (reader.Read()) в порядке следования столбов получаем данные с помощью метода GetValue(), который возвращает данные в виде объекта типа object. Например, столбец Id идет первым и представляет целое число, поэтому для его получения применяется методreader.GetValue(0). А столбец Name идет вторым, поэтому его значения получаем с помощью reader.GetValue(1).

После завершения работы с SqlDataReader надо его закрыть методом Close(). И пока один SqlDataReader не закрыт, другой объект SqlDataReader для одного и того же подключения мы использовать не сможем.

В качестве альтернативы мы могли бы обращаться к данным через название параметра:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | while (reader.Read())  {      object id = reader["id"];      object name = reader["name"];      object age = reader["age"];      Console.WriteLine("{0} \t{1} \t{2}", id, name, age);  } |

В этом случае результат будет аналогичным.

### Асинхронное чтение

Для асинхронного чтения, во-первых, применяется метод ExecuteReaderAsync() класса SqlCommand, и во-вторых, метод ReadAsync() класса SqlDataReader:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | static void Main(string[] args)  {      ReadDataAsync().GetAwaiter();        Console.Read();  }    private static async Task ReadDataAsync()  {      string connectionString = @"Data Source=.\SQLEXPRESS;Initial Catalog=usersdb;Integrated Security=True";        string sqlExpression = "SELECT \* FROM Users";      using (SqlConnection connection = new SqlConnection(connectionString))      {          await connection.OpenAsync();          SqlCommand command = new SqlCommand(sqlExpression, connection);          SqlDataReader reader = await command.ExecuteReaderAsync();            if (reader.HasRows)          {              // выводим названия столбцов              Console.WriteLine("{0}\t{1}\t{2}", reader.GetName(0), reader.GetName(1), reader.GetName(2));                while (await reader.ReadAsync())              {                  object id = reader.GetValue(0);                  object name = reader.GetValue(1);                  object age = reader.GetValue(2);                  Console.WriteLine("{0} \t{1} \t{2}", id, name, age);              }          }          reader.Close();      }  } |

**Типизация результатов SqlDataReader**

В прошлой теме для получения результатов SqlDataReader использовался метод GetValue, который возвращал значение определенного столбца в текущей ячейки в виде объекта типа object. Однако в ряде случаев такой способ не является оптимальным. Например, мы знаем, что в третьем столбце хранится возраст пользователя, который представляет целое число, и в программе мы хотели бы его использовать как целое число. Если мы получаем через метод GetValue значение, то нам его еще надо будет привести к типу object. Однако мы моем выбрать другой путь - использовать типизированные методы. Итак, изменим код программы следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | static void Main(string[] args)  {      string connectionString = @"Data Source=.\SQLEXPRESS;Initial Catalog=usersdb;Integrated Security=True";        string sqlExpression = "SELECT \* FROM Users";      using (SqlConnection connection = new SqlConnection(connectionString))      {          connection.Open();          SqlCommand command = new SqlCommand(sqlExpression, connection);          SqlDataReader reader = command.ExecuteReader();            if(reader.HasRows) // если есть данные          {              // выводим названия столбцов              Console.WriteLine("{0}\t{1}\t{2}", reader.GetName(0), reader.GetName(1), reader.GetName(2));                while (reader.Read()) // построчно считываем данные              {                  int id = reader.GetInt32(0);                  string name = reader.GetString(1);                  int age = reader.GetInt32(2);                    Console.WriteLine("{0} \t{1} \t{2}", id, name, age);              }          }            reader.Close();      }        Console.Read();  } |

Для получения данных здесь теперь используются методы GetInt32() и GetString(), которые возвращают объекты типа int и string соответственно. Причем поскольку мы знаем, что в столбце id хранится число, то мы можем получить его именно с помощью метода GetInt32, но никак не GetString. И также в этот метод передается номер столбца в таблице (нумерация опять же начинается с нуля).

Для получения данных каждого примитивного типа есть свой метод:

|  |  |  |
| --- | --- | --- |
| **Тип sql** | **Тип .NET** | **Метод** |
| bigint | Int64 | GetInt64 |
| binary | Byte[] | GetBytes |
| bit | Boolean | GetBoolean |
| char | String и Char[] | GetString и GetChars |
| datetime | DateTime | GetDateTime |
| decimal | Decimal | GetDecimal |
| float | Double | GetDouble |
| image и long varbinary | Byte[] | Bytes |
| int | Int32 | GetInt32 |
| money | Decimal | GetDecimal |
| nchar | String и Char[] | GetString и GetChars |
| ntext | String и Char[] | GetString и GetChars |
| numeric | Decimal | GetDecimal |
| nvarchar | String и Char[] | GetString и GetChars |
| real | Single (float) | GetFloat |
| smalldatetime | DateTime | GetDateTime |
| smallint | Intl6 | GetIntl6 |
| smallmoney | Decimal | GetDecimal |
| sql variant | Decimal | GetValue |
| long varchar | String и Char[] | GetString и GetChars |
| timestamp | Byte[] | GetBytes |
| tinyint | Byte | GetByte |
| uniqueidentifier | Guid | GetGuid |
| varbinary | Byte[] | GetBytes |
| varchar | String и Char[] | GetString и GetChars |

**Получение скалярных значений**

При отправке запросов мы можем использовать специальные встроенные функции SQL, например, Min, Max, Sum, Count и т.д., которые не выполняют операции с объектами и не извлекают объекты, а возвращают какое-то определенное значение. Например, функция Count подсчитывает количество объектов. И для работы с такими функциями в SqlCommand определен специальный метод ExecuteScalar. Например, найдем число всех объектов в таблице и получим минимальный возраст пользователей:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | string connectionString = @"Data Source=.\SQLEXPRESS;Initial Catalog=usersdb;Integrated Security=True";    string sqlExpression = "SELECT COUNT(\*) FROM Users";  using (SqlConnection connection = new SqlConnection(connectionString))  {      connection.Open();      SqlCommand command = new SqlCommand(sqlExpression, connection);      object count = command.ExecuteScalar();        command.CommandText = "SELECT MIN(Age) FROM Users";      object minAge = command.ExecuteScalar();        Console.WriteLine("В таблице {0} объектов", count);      Console.WriteLine("Минимальный возраст: {0}", minAge);  } |

Выражение "SELECT COUNT(\*) FROM Users" количество объектов в таблице Users, а выражение "SELECT MIN(Age) FROM Users" находит минимальное значение столбца Age. В качестве результата метод ExecuteScalar() возвращает объект типа object.

И если мы захотим получать данные асинхронным образом, то мы можем использовать асинхронную версию этого метода -**ExecuteScalarAsync()**.

**Параметризация запросов**

Ранее для отправки запросов мы напрямую добавляли данные в выражение sql. Например, для отправки запроса на добавление данных в БД применялось следующее выражение:

|  |  |
| --- | --- |
| 1 | string sqlExpression = String.Format("INSERT INTO Users (Name, Age) VALUES ('{0}', {1})", name, age); |

В данном случае предполагается, что значения для переменных name и age вводит пользователь.

Что если переменная name получит следующее значение:

|  |  |
| --- | --- |
| 1 | name = "Tom',10);INSERT INTO Users (Name, Age) VALUES('Hack"; |

В этом случае sql-выражение в итоге будет выглядеть следующим образом:

|  |  |
| --- | --- |
| 1 | INSERT INTO Users (Name, Age) VALUES ('Tom',10);INSERT INTO Users (Name, Age) VALUES('Hack,{1})"; |

В итоге в базу данных будет добавлено два объекта. Это относительно безобидный вид подмены sql-выражения, но реальные возможности встраивания зловредных скриптов таковы, что можно вообще потерять данные в БД, если предоставить пользователям подобным образом добавлять данные.

Чтобы выйти из этой ситуации, в sql-командах используются параметры:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | string connectionString = @"Data Source=.\SQLEXPRESS;Initial Catalog=usersdb;Integrated Security=True";  int age = 23;  string name = "T',10);INSERT INTO Users (Name, Age) VALUES('H";  string sqlExpression = "INSERT INTO Users (Name, Age) VALUES (@name, @age)";  using (SqlConnection connection = new SqlConnection(connectionString))  {      connection.Open();      SqlCommand command = new SqlCommand(sqlExpression, connection);      // создаем параметр для имени      SqlParameter nameParam = new SqlParameter("@name", name);      // добавляем параметр к команде      command.Parameters.Add(nameParam);      // создаем параметр для возраста      SqlParameter ageParam = new SqlParameter("@age", age);      // добавляем параметр к команде      command.Parameters.Add(ageParam);        int number = command.ExecuteNonQuery();      Console.WriteLine("Добавлено объектов: {0}", number); // 1  } |

Для определения параметров используется объект **SqlParameter**. Этот объект имеет ряд конструкторов, но в данном случае передается название параметра и его значение. Причем название параметров начинается со знака @ и должно совпадать с тем названием, которое используется в sql-выражении (то есть в "INSERT INTO Users (Name, Age) VALUES (@name, @age)"). После определения параметра он добавляется в коллекцию параметров команды.

При выполнении команды на место параметров в sql-выражении подставляются их значения. При этом не важно, что параметр @name в значении определяет еще одну команду INSERT - все его значение будет добавлено в столбец name в таблице Users.

**Выходные параметры запросов**

Параметры, которые используются в командах, могут быть нескольких типов. Тип параметра задается с помощью свойства Direction объкта SqlParameter. Данное свойство принимает одно из значений перечисления **ParameterDirection**:

* Input: параметр является входным, то есть предназначен для передачи значений в sql-выражение запроса. Это значение по умолчанию для всех параметров
* InputOutput: параметр может быть как входным, так и выходным.
* Output: параметр является выходным, то есть используется для возвращения запросом каких-либо значений
* ReturnValue: параметр представляет результат выполнения выражения или хранимой процедуры

Используем выходные параметры для возвращения id строки, которая была добавлена:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | string connectionString = @"Data Source=.\SQLEXPRESS;Initial Catalog=usersdb;Integrated Security=True";  int age = 23;  string name = "Kenny";  string sqlExpression = "INSERT INTO Users (Name, Age) VALUES (@name, @age);SET @id=SCOPE\_IDENTITY()";  using (SqlConnection connection = new SqlConnection(connectionString))  {      connection.Open();      SqlCommand command = new SqlCommand(sqlExpression, connection);      // создаем параметр для имени      SqlParameter nameParam = new SqlParameter("@name", name);      // добавляем параметр к команде      command.Parameters.Add(nameParam);      // создаем параметр для возраста      SqlParameter ageParam = new SqlParameter("@age", age);      // добавляем параметр к команде      command.Parameters.Add(ageParam);      // параметр для id      SqlParameter idParam = new SqlParameter      {          ParameterName = "@id",          SqlDbType = SqlDbType.Int,          Direction = ParameterDirection.Output // параметр выходной      };      command.Parameters.Add(idParam);        command.ExecuteNonQuery();         // получим значения выходного параметра      Console.WriteLine("Id нового объекта: {0}", idParam.Value);  } |

Здесь к sql-выражению добавляется операция присвоения параметру id идентификатора добавленной строки: SET @id=SCOPE\_IDENTITY(). Для определения параметра используются его свойства. ParameterName задает имя параметра, SqlDbType указывает на тип параметра, а свойствоDirection определяет тип параметра.

После выполнения команды параметр получает значение, которое мы можем получить через его свойство Value:

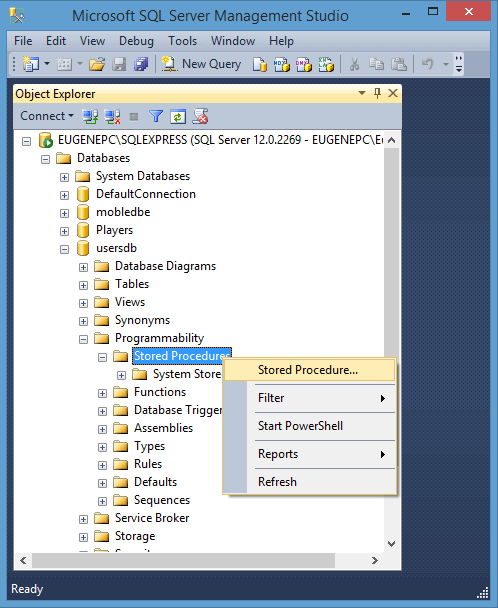
|  |  |
| --- | --- |
| 1 | Console.WriteLine("Id нового объекта: {0}", idParam.Value); |

**Работа с хранимыми процедурами**

Хранимые процедуры являются еще одной формой выполнения запросов к базе данных. Но по сравнению с ранее рассмотренными запросами, которые посылаются из приложения базе данных, хранимые процедуры определяются на сервере и предоставляют большую производительность и являются более безопасными.

Объект SqlCommand имеет встроенную поддержку хранимых процедур. В частности у него определено свойство CommandType, которое в качестве значения принимает значение из перечисления System.Data.CommandType. И значение System.Data.CommandType.StoredProcedure как раз указывает, что будет использоваться хранимая процедура.

Но чтобы использовать хранимые процедуры, нам надо их вначале создать. Для этого перейдем в SQL Server Management Studio к нашей базе данных usersdb, раскроем ее узел и далее выберем Programmability->Stored Procedures. Нажмем на этот узел правой кнопкой мыши и в контекстном меню выберем пункт **Stored Procedure...**:

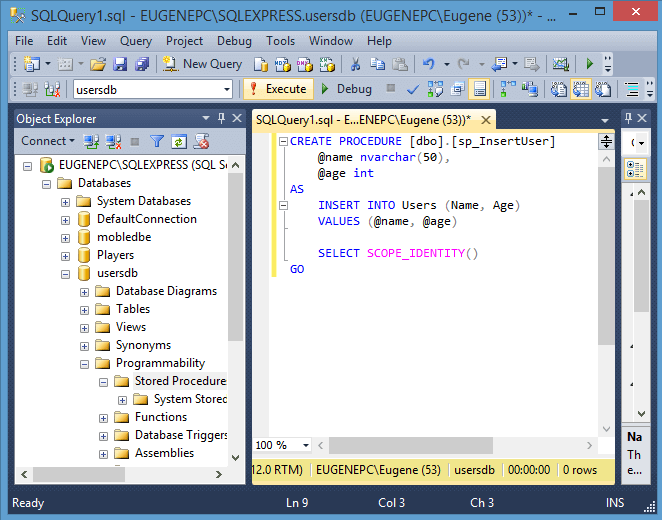


В центральной части программы открывает код процедуры, который генерируется по умолчанию. Заменим этот код следующим:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | CREATE PROCEDURE [dbo].[sp\_InsertUser]      @name nvarchar(50),      @age int  AS      INSERT INTO Users (Name, Age)      VALUES (@name, @age)        SELECT SCOPE\_IDENTITY()  GO |

Эта процедура выполняет добавление данных. После выражения CREATE PROCEDURE идет название процедуры. Процедура называется "sp\_InsertUser", и по этому названию мы ее будем вызывать в коде C#. Название процедуры может быть любое.

Процедура имеет два входных параметра: @name и @age. Через эти параметры будут передаваться значения для имени и возраста пользователя. В самом теле процедуры после выражения AS идет стандартное sql-выражение INSERT, которое выполняет добавление данных. И в конце с помощью выражения SELECT возвращается результат. Выражение SCOPE\_IDENTITY() возвращает id добавленной записи, поэтому на выходе из процедуры мы получим id новой записи. И завершается процедура ключевым словом GO.



И затем нажмем на кнопку Execute. После этого в базу данных добавляется хранимая процедура.

Подобным образом добавим еще одну процедуру, которая будет возвращать объекты:

|  |  |
| --- | --- |
| 1  2  3  4 | CREATE PROCEDURE [dbo].[sp\_GetUsers]  AS      SELECT \* FROM Users  GO |

И также для ее добавления нажмем на кнопку Execute.

Теперь перейдем к коду C# и определим следующую программу:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72 | class Program  {      static string connectionString = @"Data Source=.\SQLEXPRESS;Initial Catalog=usersdb;Integrated Security=True";      static void Main(string[] args)      {          Console.Write("Введите имя пользователя:");          string name = Console.ReadLine();            Console.Write("Введите возраст пользователя:");          int age = Int32.Parse(Console.ReadLine());            AddUser(name, age);          Console.WriteLine();          GetUsers();            Console.Read();      }      // добавление пользователя      private static void AddUser(string name, int age)      {          // название процедуры          string sqlExpression = "sp\_InsertUser";            using (SqlConnection connection = new SqlConnection(connectionString))          {              connection.Open();              SqlCommand command = new SqlCommand(sqlExpression, connection);              // указываем, что команда представляет хранимую процедуру              command.CommandType = System.Data.CommandType.StoredProcedure;              // параметр для ввода имени              SqlParameter nameParam = new SqlParameter              {                  ParameterName = "@name",                  Value = name              };              // добавляем параметр              command.Parameters.Add(nameParam);              // параметр для ввода возраста              SqlParameter ageParam = new SqlParameter              {                  ParameterName = "@age",                  Value = age              };              command.Parameters.Add(ageParam);                var result = command.ExecuteScalar();              // если нам не надо возвращать id              //var result = command.ExecuteNonQuery();                Console.WriteLine("Id добавленного объекта: {0}", result);          }      }        // вывод всех пользователей      private static void GetUsers()      {          // название процедуры          string sqlExpression = "sp\_GetUsers";            using (SqlConnection connection = new SqlConnection(connectionString))          {              connection.Open();              SqlCommand command = new SqlCommand(sqlExpression, connection);              // указываем, что команда представляет хранимую процедуру              command.CommandType = System.Data.CommandType.StoredProcedure;              var reader = command.ExecuteReader();                if (reader.HasRows)              {                  Console.WriteLine("{0}\t{1}\t{2}", reader.GetName(0), reader.GetName(1), reader.GetName(2));                    while (reader.Read())                  {                      int id = reader.GetInt32(0);                      string name = reader.GetString(1);                      int age = reader.GetInt32(2);                      Console.WriteLine("{0} \t{1} \t{2}", id, name, age);                  }              }              reader.Close();          }      }  } |

Для упрощения кода обращения к процедурам здесь вынесены в отдельные методы. В методе AddUser вызывается процедура sp\_InsertUser. Ее название передается в конструктор объекта SqlCommand также, как и обычное sql-выражение. И с помощью выраженияcommand.CommandType = System.Data.CommandType.StoredProcedure устанавливается, что это выражение система будет рассматривать как хранимую процедуру.

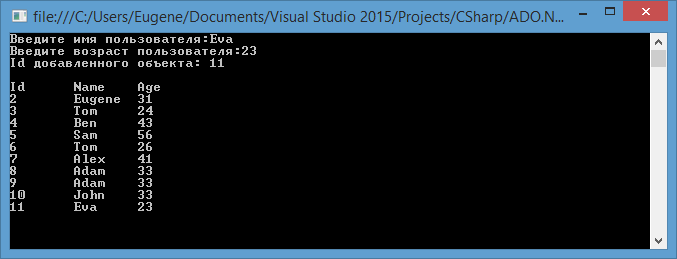
Поскольку процедура получает данные через параметры, то соответственно нам надо определить эти параметры с помощью объектов**SqlParameter**. Ему передается название параметра и значение. Названия параметров должны соответствовать тем названиям, которые мы определили в коде процедуры.

С помощью метода command.Parameters.Add() параметры добавляются к процедуре. И затем происходит выполнение.

Так как в коде процедуры добавления мы определили возвращение id новой записи, то есть возвращение скалярного значения, то для выполнения команды и его получения мы можем использовать метод ExecuteScalar(). Но мы также можем использовать и методExecuteNonOuery(), только он вернет количество добавленных записей, а не id.

В случае второго метода все еще проще: объекту команды просто передается название процедуры, и так как процедура фактически выполняет выражение SELECT и возвращает набор данных, то для выполнения команды мы можем использовать метод ExecuteReader(). И с помощью ридера получить все данные.

Запустим программу и введем какие-либо данные на добавление:



**Выходные параметры хранимых процедур**

Хранимая процедура может использовать не только входные параметры, через которые передаются значения в процедуру, но и выходные параметры. Выходные параметры по сути представляют результат работы процедуры, те значения, которые она возвращает пользователю.

Опять же возьмем наш базу данных и определим в ней следующую хранимую процедуру:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | CREATE PROCEDURE [dbo].[sp\_GetAgeRange]      @name nvarchar(50)='%',      @minAge int out,      @maxAge int out  AS      SELECT @minAge = MIN(Age), @maxAge = MAX(Age) FROM Users WHERE Name LIKE @name  GO |

В этой процедуре определено три параметра, с помощью которых мы будем получать минимальный и максимальный возраст в базе данных. Через параметр @name мы будем получать имя пользователя, для которого осуществляется поиск минимального и максимального возраста. Причем в данном случае параметр по умолчанию имеет значение '%'. Данное значение указывает на произвольную строку. То есть если значение для этого параметра передано, то процедура будет находить возраст только для пользователей с этим именем. Если же значение для параметра не передано, то процедура будет находить минимальный и максимальный возраст всех пользователей вне зависимости от имени.

Параметры @minAge и @maxAge являются выходными благодаря указанию ключевого слова out в их определении. Через них мы собственно и будем получать минимальный и максимальный возраст.

Теперь перейдем к программе на C# и определим в ней следующий код:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59 | using System;  using System.Data.SqlClient;  using System.Data;    namespace AdoNetConsoleApp  {      class Program      {          static string connectionString = @"Data Source=.\SQLEXPRESS;Initial Catalog=usersdb;Integrated Security=True";          static void Main(string[] args)          {              Console.Write("Введите имя пользователя:");              string name = Console.ReadLine();                GetAgeRange(name);                Console.Read();          }            private static void GetAgeRange(string name)          {              string sqlExpression = "sp\_GetAgeRange";                using (SqlConnection connection = new SqlConnection(connectionString))              {                  connection.Open();                  SqlCommand command = new SqlCommand(sqlExpression, connection);                  command.CommandType = CommandType.StoredProcedure;                    // если имя определено, добавляем параметр для ввода имени                  if(!String.IsNullOrEmpty(name))                  {                      SqlParameter nameParam = new SqlParameter                      {                          ParameterName = "@name",                          Value = name                      };                      command.Parameters.Add(nameParam);                  }                    // опеределяем первый выходной параметр                  SqlParameter minAgeParam = new SqlParameter                  {                      ParameterName = "@minAge",                      SqlDbType = SqlDbType.Int // тип параметра                  };                  // указываем, что параметр будет выходным                  minAgeParam.Direction = ParameterDirection.Output;                  command.Parameters.Add(minAgeParam);                    // опеределяем второй выходной параметр                  SqlParameter maxAgeParam = new SqlParameter                  {                      ParameterName = "@maxAge",                      SqlDbType = SqlDbType.Int                  };                  maxAgeParam.Direction = ParameterDirection.Output;                  command.Parameters.Add(maxAgeParam);                    command.ExecuteNonQuery();                    Console.WriteLine("Минимальный возраст: {0}", command.Parameters["@minAge"].Value);                  Console.WriteLine("Максимальный возраст: {0}", command.Parameters["@maxAge"].Value);              }          }      }  } |

Выходные параметры команды определяются также, как и входные за тем исключением, что для свойства Direction нам надо установить значение **ParameterDirection.Output**. По умолчанию все параметры имеют значение ParameterDirection.Input, которое указывает, что они являются входными.

После выполнения команды с помощью метода ExecuteNonOuery() мы можем получить параметры по названию или по их индексу в коллекции параметров: command.Parameters["название параметра"]. И чтобы получить само значение параметра, надо использовать его свойство Value.

Теперь мы можем ввести имя пользователя, либо просто нажать на Enter, и процедура вернет нам выходные параметры. При этом может возникнуть ситуация, что пользователя с введенным именем не окажется в системе, и тогда параметры будут содержать пустые значения.

**Транзакции**

Транзакции позволяют выполнять ряд операций в виде одного целостного пакета. И если хотя бы одна из этих операций завершится неудачно, то произойдет откат выполнения остальных операций.

Для начала транзакции надо вызвать метод BeginTransaction() объекта SqlConnection:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | string connectionString = @"Data Source=.\SQLEXPRESS;Initial Catalog=usersdb;Integrated Security=True";  using (SqlConnection connection = new SqlConnection(connectionString))  {      connection.Open();      SqlTransaction transaction = connection.BeginTransaction();        SqlCommand command = connection.CreateCommand();      command.Transaction = transaction;        try      {          // выполняем две отдельные команды          command.CommandText = "INSERT INTO Users (Name, Age) VALUES('Tim', 34)";          command.ExecuteNonQuery();          command.CommandText = "INSERT INTO Users (Name, Age) VALUES('Kat', 31)";          command.ExecuteNonQuery();            // подтверждаем транзакцию          transaction.Commit();          Console.WriteLine("Данные добавлены в базу данных");      }      catch (Exception ex)      {          Console.WriteLine(ex.Message);          transaction.Rollback();      }  } |

Чтобы все операции с объектом SqlCommand выполнялись как одна транзакция, надо присвоить объект транзакции его свойству Transaction:

|  |  |
| --- | --- |
| 1 | command.Transaction = transaction; |

Для завершения всех операции после их выполнения вызывается метод Commit() объекта SqlTransaction:

|  |  |
| --- | --- |
| 1 | transaction.Commit(); |

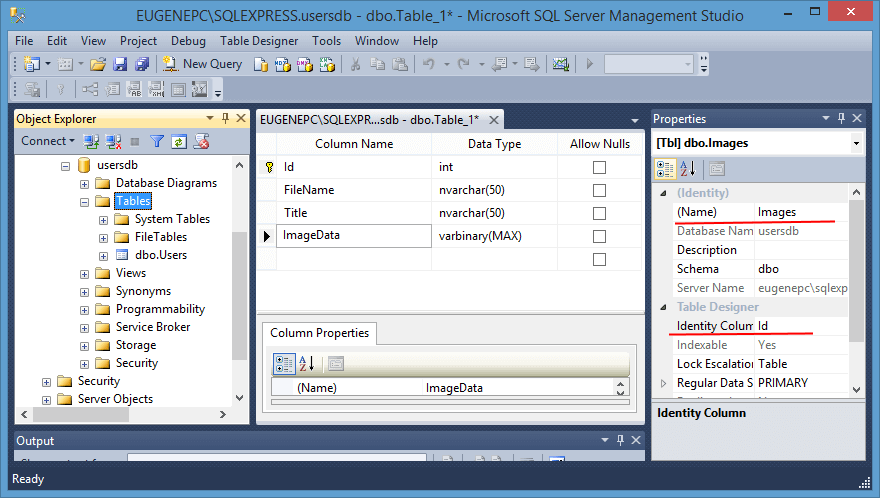
Если в ходе выполнения произошла ошибка, то мы можем откатить транзакцию, вызвав метод Rollback():

|  |  |
| --- | --- |
| 1 | transaction.Rollback(); |

**Сохранение и извлечение файлов из базы данных**

### Сохранение в базу данных файлов

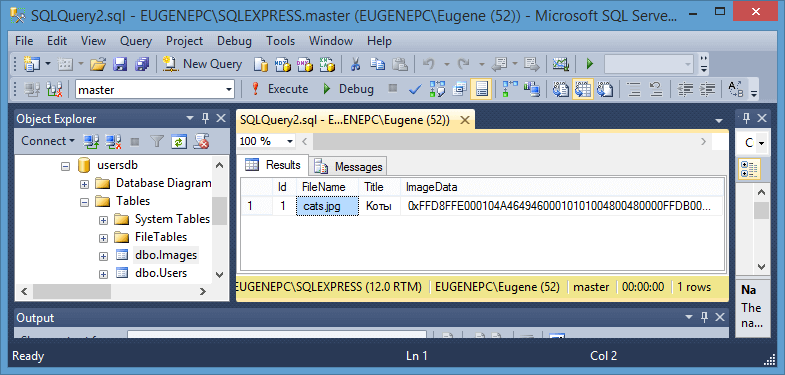
Рассмотрим, как мы можем сохранять файлы, в частности, файлы изображений в базу данных. Для этого добавим в базу данных новую таблицу Images с четырьмя столбцами: Id (первичный ключ и идентификатор, имеет тип int), FileName (будет хранить имя файла и имеет тип nvarchar), Title (будет хранить заголовок файла и также имеет тип nvarchar) и ImageData (будет содержать бинарные данные файла и имеет тип varbimary(MAX)).



Определим код, в котором будут загружаться данные в таблицу:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | private static void SaveFileToDatabase()  {      string connectionString = @"Data Source=.\SQLEXPRESS;Initial Catalog=usersdb;Integrated Security=True";      using (SqlConnection connection = new SqlConnection(connectionString))      {          connection.Open();          SqlCommand command = new SqlCommand();          command.Connection = connection;          command.CommandText = @"INSERT INTO Images VALUES (@FileName, @Title, @ImageData)";          command.Parameters.Add("@FileName", SqlDbType.NVarChar, 50);          command.Parameters.Add("@Title", SqlDbType.NVarChar, 50);          command.Parameters.Add("@ImageData", SqlDbType.Image, 1000000);            // путь к файлу для загрузки          string filename = @"C:\Users\Eugene\Pictures\cats.jpg";          // заголовок файла          string title = "Коты";          // получаем короткое имя файла для сохранения в бд          string shortFileName = filename.Substring(filename.LastIndexOf('\')+1); // cats.jpg          // массив для хранения бинарных данных файла          byte[] imageData;          using (System.IO.FileStream fs = new System.IO.FileStream(filename, FileMode.Open))          {              imageData = new byte[fs.Length];              fs.Read(imageData, 0, imageData.Length);          }          // передаем данные в команду через параметры          command.Parameters["@FileName"].Value = shortFileName;          command.Parameters["@Title"].Value = title;          command.Parameters["@ImageData"].Value = imageData;            command.ExecuteNonQuery();      }  } |

После выполнения этой программы в базе данных появится соответствующая запись:



### Извлечение файлов из базы данных

В прошлой теме мы рассмотрели, как добавить файл в базу данных. Теперь произведем обратную операцию - получим файл из БД. Вначале определим класс файла, который упростит работу с данными:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public class Image  {      public Image(int id, string filename, string title, byte[] data)      {          Id = id;          FileName = filename;          Title = title;          Data = data;      }      public int Id { get; private set; }      public string FileName { get; private set; }      public string Title { get; private set; }      public byte[] Data { get; private set; }  } |

Затем в коде программы определим следующий метод:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | private static void ReadFileFromDatabase()  {      string connectionString = @"Data Source=.\SQLEXPRESS;Initial Catalog=usersdb;Integrated Security=True";      List<Image> images = new List<Image>();      using (SqlConnection connection = new SqlConnection(connectionString))      {          connection.Open();          string sql = "SELECT \* FROM Images";          SqlCommand command = new SqlCommand(sql, connection);          SqlDataReader reader = command.ExecuteReader();            while(reader.Read())          {              int id = reader.GetInt32(0);              string filename = reader.GetString(1);              string title = reader.GetString(2);              byte[] data = (byte[])reader.GetValue(3);                Image image = new Image(id, filename, title, data);              images.Add(image);          }      }      // сохраним первый файл из списка      if(images.Count>0)      {          using (System.IO.FileStream fs = new System.IO.FileStream(images[0].FileName, FileMode.OpenOrCreate))          {              fs.Write(images[0].Data, 0, images[0].Data.Length);              Console.WriteLine("Изображение '{0}' сохранено", images[0].Title);          }      }  } |

В этом методе с помощью SqlDataReader мы получаем значения из БД и по ним создаем объект Image, который потом добавляется в список. И в конце смотрим, если в списке есть элементы, то берем первый элемент и сохраняем его на локальный компьютер. И после сохранения в папке нашей программы появится загруженный из базы данных файл.