_ Введение в ADO.NET

Для хранения данных используются различные системы управления базами данных: MS SQL Server, Oracle, MySQL и так далее. И большинство крупных приложений так или иначе используют для хранения данных эти системы управления базами данных. Однако чтобы осуществлять связь между базой данных и приложением на C# необходим посредник. И именно таким посредником является технология **ADO.NET.**

ADO.NET предоставляет собой технологию работы с данными, которая основана на платформе .NET Framework. Эта технология представляет набор классов, через которые можно отправлять запросы к базам данных, устанавливать подключения, получать ответ от базы данных и производить ряд других операций.
Причем систем управления баз данных может быть множество. В своей сущности они могут различаться. MS SQL Server, например, для создания запросов использует язык T-SQL, а MySQL и Oracle применяют язык PL-SQL. Разные системы баз данных могут иметь разные типы данных. Также могут различаться какие-то другие моменты. Однако функционал ADO.NET построен таким образом, чтобы предоставить разработчикам унифицированный интерфейс для работы с самыми различными СУБД.

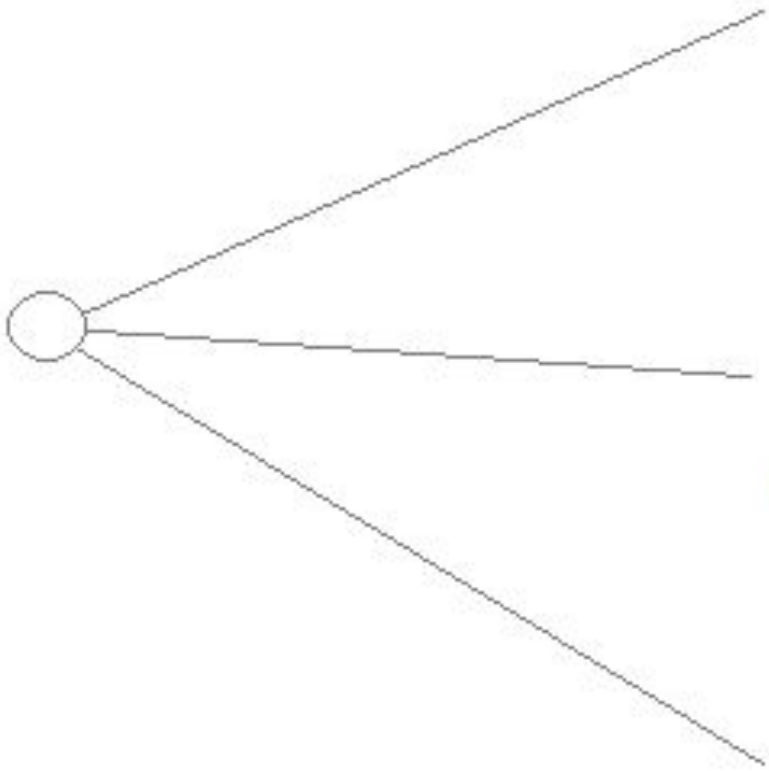
Основу интерфейса взаимодействия с базами данных в **ADO.NET** представляет ограниченный круг объектов: Connection, Command, DataReader, DataSet и DataAdapter.

С помощью объекта **Connection** происходит установка подключения к источнику данных.

- Объект **Command** позволяет выполнять операции с данными из БД.
- Объект **DataReader** считывает полученные в результате запроса данные.
- Объект **DataSet** предназначен для хранения данных из БД и позволяет работать с ними независимо от БД.
- Объект **DataAdapter** является посредником между DataSet и источником данных.

Главным образом, через эти объекты и будет идти работа с базой данных. Однако чтобы использовать один и тот же набор объектов для разных источников данных, необходим соответствующий **провайдер данных**. Через провайдер данных в ADO.NET и осуществляется взаимодействие с базой данных. Причем для каждого источника данных в ADO.NET может быть свой провайдер, который собственно и определяет конкретную реализацию вышеуказанных классов.

DBUtils.cs



По умолчанию в **ADO.NET** имеются следующие встроенные провайдеры:

- Провайдер для MS SQL Server
- Провайдер для OLE DB (Предоставляет доступ к некоторым старым версиям MS SQL Server, а также к БД Access, DB2, MySQL и Oracle)
- Провайдер для ODBC (Провайдер для тех источников данных, для которых нет своих провайдеров)
- Провайдер для Oracle
- Провайдер EntityClient. Провайдер данных для технологии ORM Entity Framework
- Провайдер для сервера SQL Server Compact 4.0

Кроме этих провайдеров, которые являются встроенными, существует также множество других, предназначенных для различных баз данных, например, для MySQL.

Основные пространства имен, которые используются в ADO.NET:

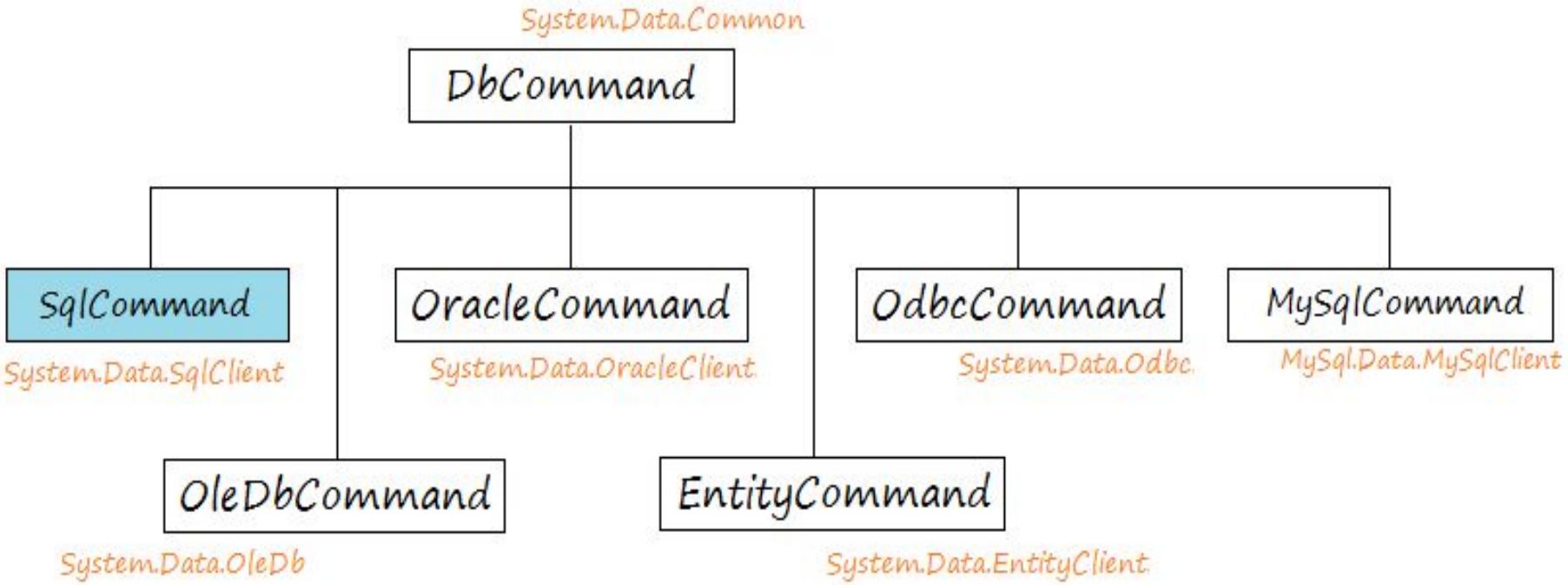
- **System.Data**: определяет классы, интерфейсы, делегаты, которые реализуют архитектуру ADO.NET
- **System.Data.Common**: содержит классы, общие для всех провайдеров ADO.NET
- **System.Data.Design**: определяет классы, которые используются для создания своих собственных наборов данных
- **System.Data.Odbc**: определяет функциональность провайдера данных для ODBC
- **System.Data.OleDb**: определяет функциональность провайдера данных для OLE DB
- **System.Data.Sql**: хранит классы, которые поддерживают специфичную для SQL Server функциональность
- **System.Data.OracleClient**: определяет функциональность провайдера для баз данных Oracle
- **System.Data.SqlClient**: определяет функциональность провайдера для баз данных MS SQL Server
- **System.Data.SqlServerCe**: определяет функциональность провайдера для SQL Server Compact 4.0
- **System.Data.SqlTypes**: содержит классы для типов данных MS SQL Servera
- **Microsoft.SqlServer.Server**: хранит компоненты для взаимодействия SQL Server и среды CLR

Функционально классы ADO.NET можно разбить на два уровня: подключенный и отключенный. Каждый провайдер данных .NET реализует свои версии объектов **Connection**, **Command**, **DataReader**, **DataAdapter** и ряда других, который составляют подключенный уровень. С помощью них устанавливается подключение к БД и выполняется с ней взаимодействие.

Object	SQL Server	OLE DB	ODBC
Connection	SqlConnection	OleDbConnection	OdbcConnection
Command	SqlCommand	OleDbCommand	OdbcCommand
Data reader	SqlDataReader	OleDbDataReader	OdbcDataReader
Data adapter	SqlDataAdapter	OleDbDataAdapter	OdbcDataAdapter

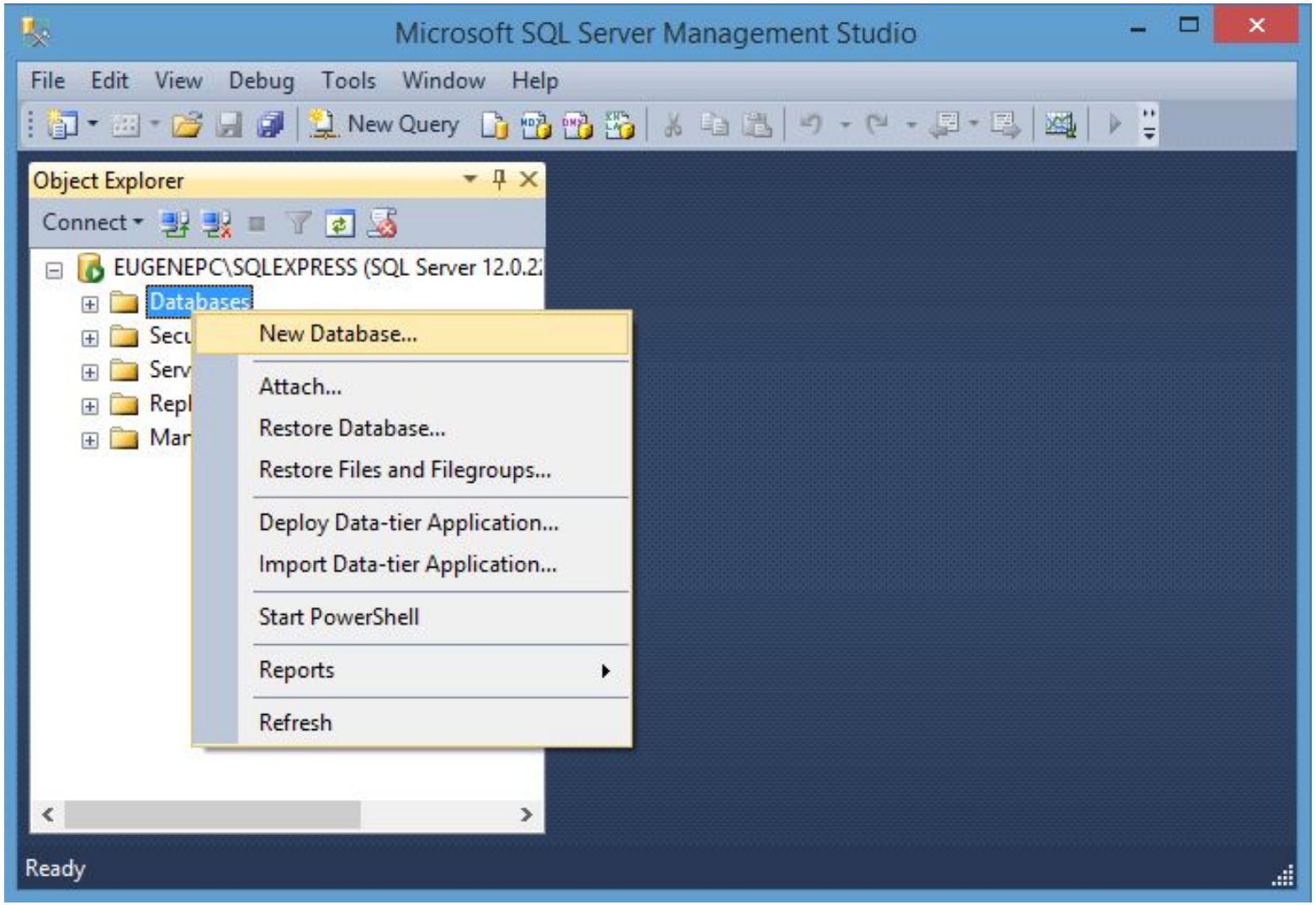
Как правило, реализации этих объектов для каждого конкретного провайдера в своем названии имеют префикс, который указывает на провайдер.

Другие классы, такие как **DataSet**, **DataTable**, **DataRow**, **DataColumn** и ряд других составляют отключенный уровень, так как после извлечения данных в **DataSet** можно работать с этими данными независимо от того, установлено ли подключение или нет. То есть после получения данных из БД приложение может быть отключено от источника данных.

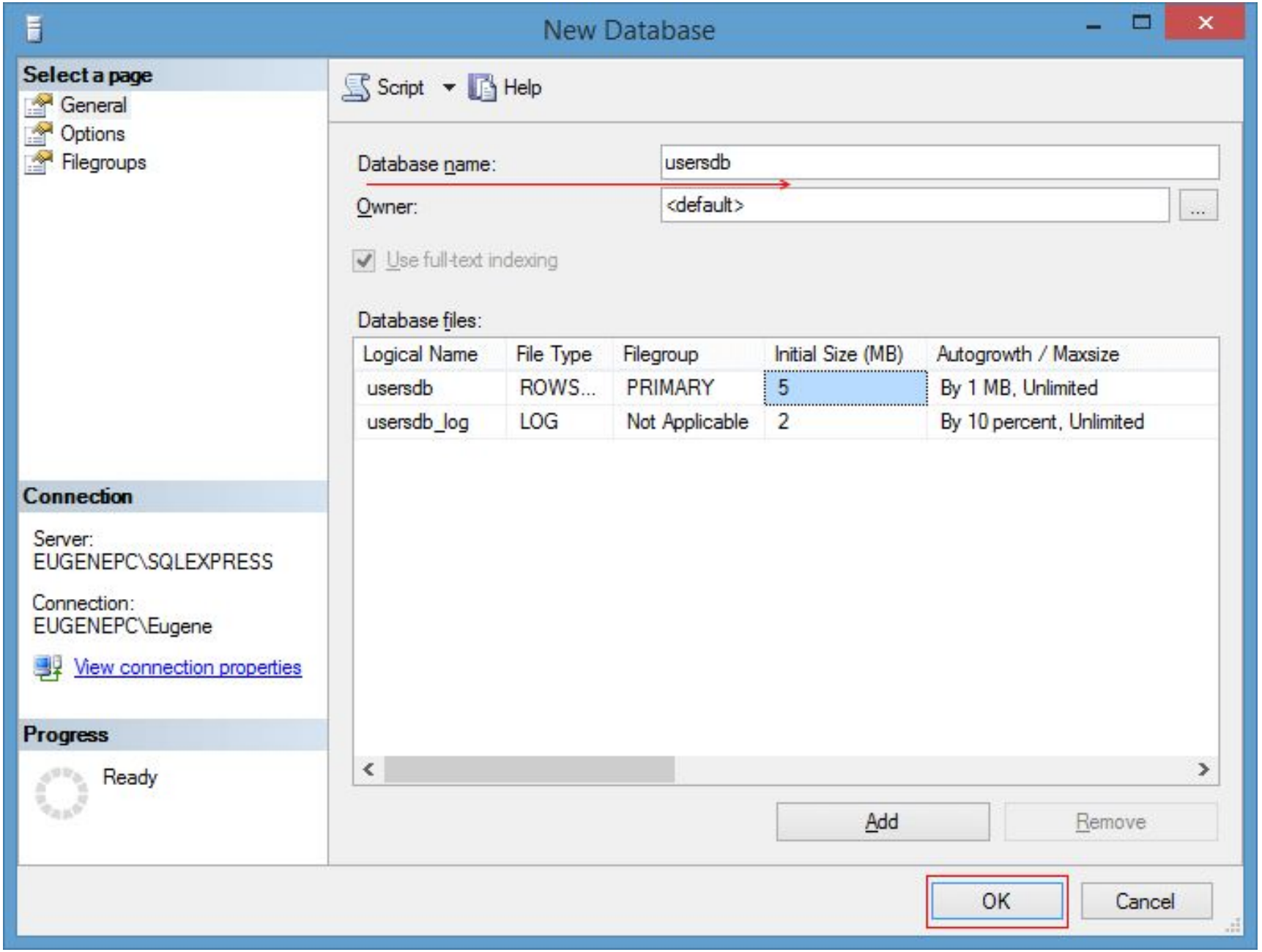


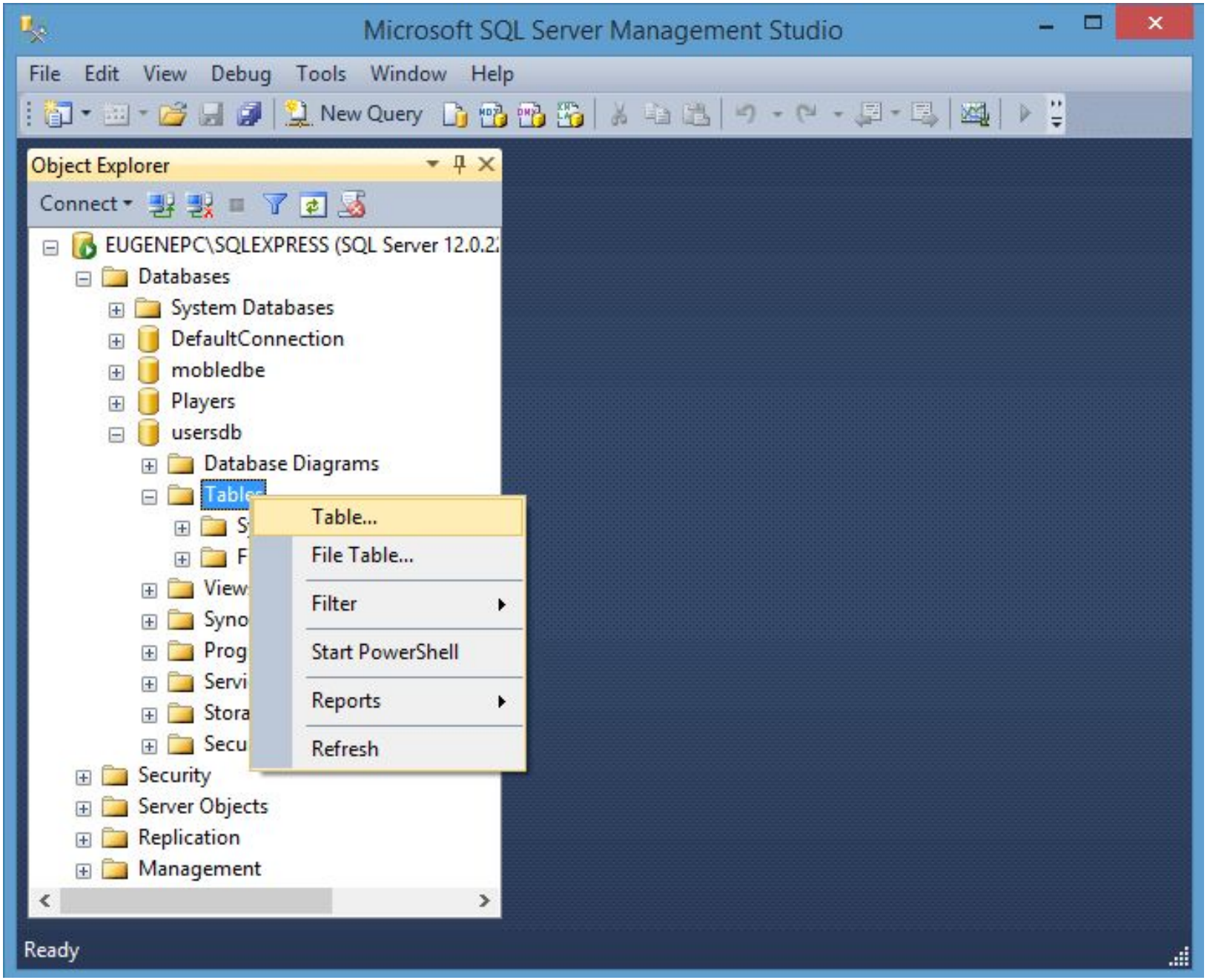
В **C#** чтобы манипулировать с базой данных **SQL Server**, например **query**, **insert**, **update**, **delete** используется объект **SqlCommand**, **SqlCommand** расширенный класс из **DbCommand**.

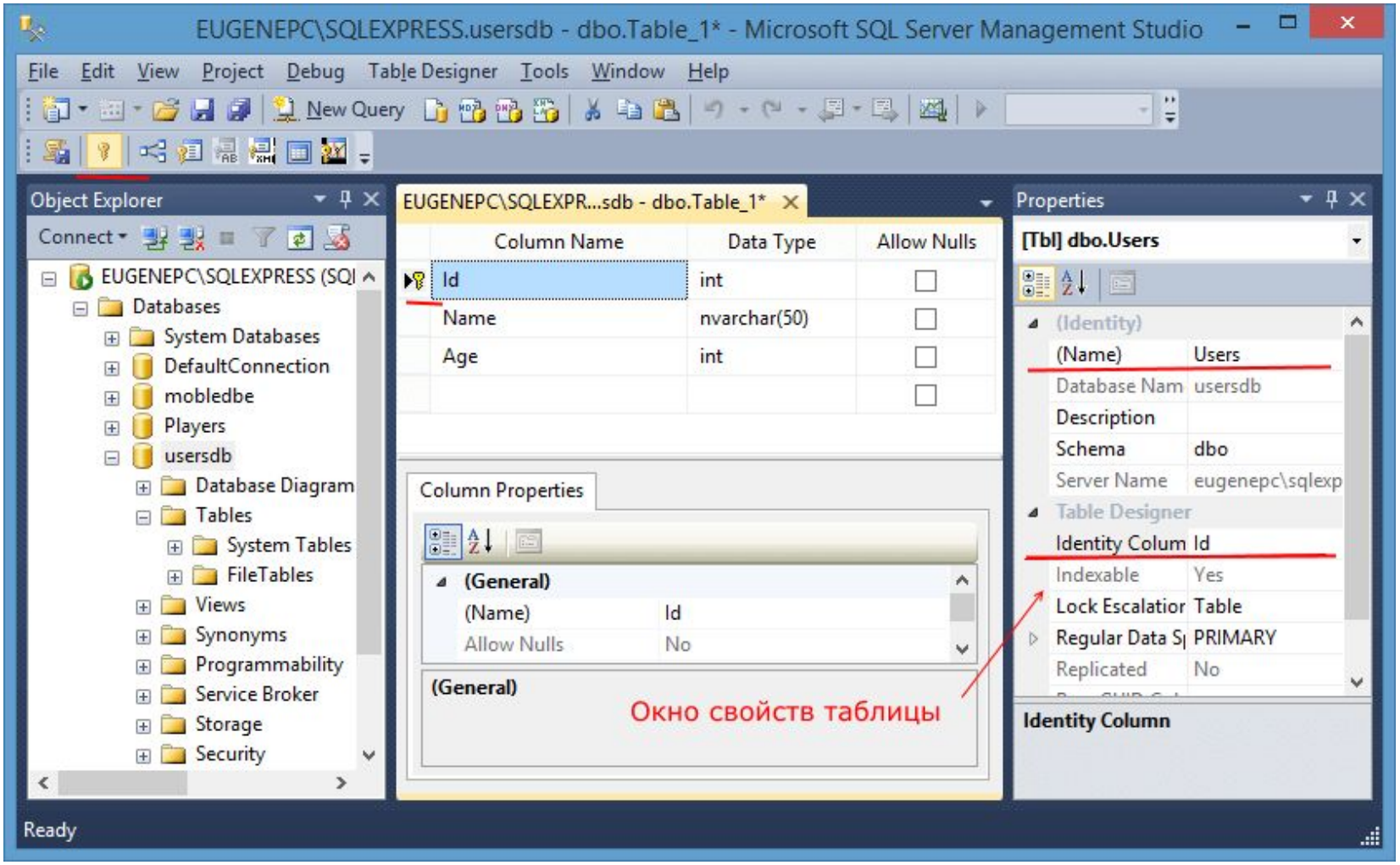
В случае, когда нужен **query**, **insert**, **update** или **delete** в **Oracle Database** нужно использовать **OracleCommand**, или с **MySQL** это **MySQLCommand**. К сожалению будет трудно если хотите использовать исходный код для разных баз данных.

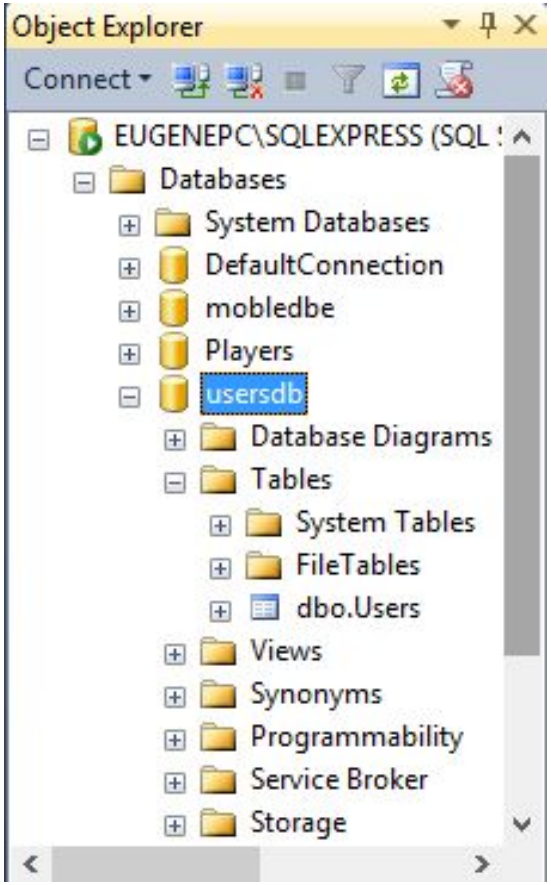


192.168.4.211 log: MSSQL207 pass- 12345

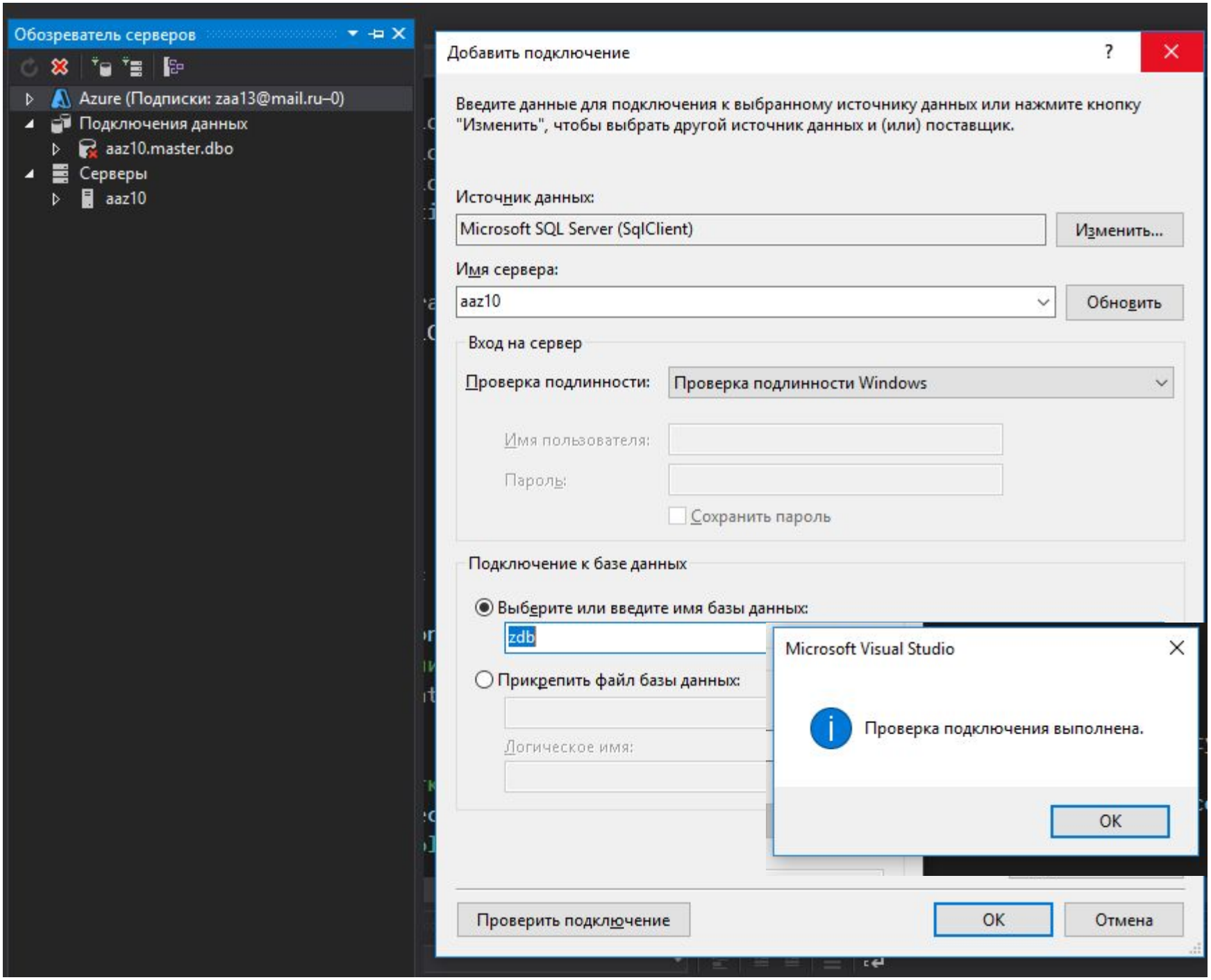

New Database
Select a page
General
Options
Filegroups
Script
Help
Database name:
usersdb
Owner:
<default>
...
Use full-text indexing
Database files:
Logical Name
File Type
Filegroup
Initial Size (MB)
Autogrowth / Maxsize
usersdb
ROWS...
PRIMARY
5
By 1 MB, Unlimited
usersdb_log
LOG
Not Applicable
2
By 10 percent, Unlimited
Add
Remove
Connection
Server:
EUGENEPC\SQLEXPRESS
Connection:
EUGENEPC\Eugene
View connection properties
Progress
Ready
OK
Cancel


Microsoft SQL Server Management Studio
File Edit View Debug Tools Window Help
New Query
Object Explorer
Connect
EUGENEPC\SQLEXPRESS (SQL Server 12.0.2
Databases
System Databases
DefaultConnection
mobledbe
Players
usersdb
Database Diagrams
Table
Table...
File Table...
Filter
Start PowerShell
Reports
Refresh
Security
Server Objects
Replication
Management
Ready


EUGENEPC\SQLEXPRESS.usersdb - dbo.Table_1* - Microsoft SQL Server Management Studio
File Edit View Project Debug Table Designer Tools Window Help
New Query
Object Explorer
Connect
EUGENEPC\SQLEXPRESS (SQL
Databases
System Databases
DefaultConnection
mobledbe
Players
usersdb
Database Diagram
Tables
System Tables
FileTables
Views
Synonyms
Programmability
Service Broker
Storage
Security
EUGENEPC\SQLEXPR...sdb - dbo.Table_1*
Column Name
Data Type
Allow Nulls
Id
int
Name
nvarchar(50)
Age
int
Column Properties
(General)
(Name)
Id
Allow Nulls
No
(General)
Properties
[Tbl] dbo.Users
(Identity)
(Name)
Users
Database Nam
usersdb
Description
Schema
dbo
Server Name
eugenepc\sqlexp
Table Designer
Identity Colum
Id
Indexable
Yes
Lock Escalatior
Table
Regular Data S
PRIMARY
Replicated
No
Identity Column
Окно свойств таблицы
Ready



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


Обозреватель серверов
Azure (Подписки: zaa13@mail.ru–0)
Подключения данных
aaz10.master.dbo
Серверы
aaz10
Добавить подключение
Введите данные для подключения к выбранному источнику данных или нажмите кнопку "Изменить", чтобы выбрать другой источник данных и (или) поставщик.
Источник данных:
Microsoft SQL Server (SqlClient)
Изменить...
Имя сервера:
aaz10
Обновить
Вход на сервер
Проверка подлинности:
Проверка подлинности Windows
Имя пользователя:
Пароль:
Сохранить пароль
Подключение к базе данных
Выберите или введите имя базы данных:
zdb
Прикрепить файл базы данных:
Логическое имя:
Microsoft Visual Studio
Проверка подключения выполнена.
OK
Проверить подключение
OK
Отмена

Строка подключения

После определения источника данных можно к нему подключаться. Надо определить строку подключения, предоставляющая информацию о базе данных и сервере, к которым предстоит установить подключение:

```
static void Main(string[] args)
    {
string connectionString=@"Data Source=.\SQLEXPRESS;Initial Catalog=usersdb;Integrated Security=True";
    }
```

При использовании различных СУБД, 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 БД:

- **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

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.

App.config на данный момент имеет следующее определение:

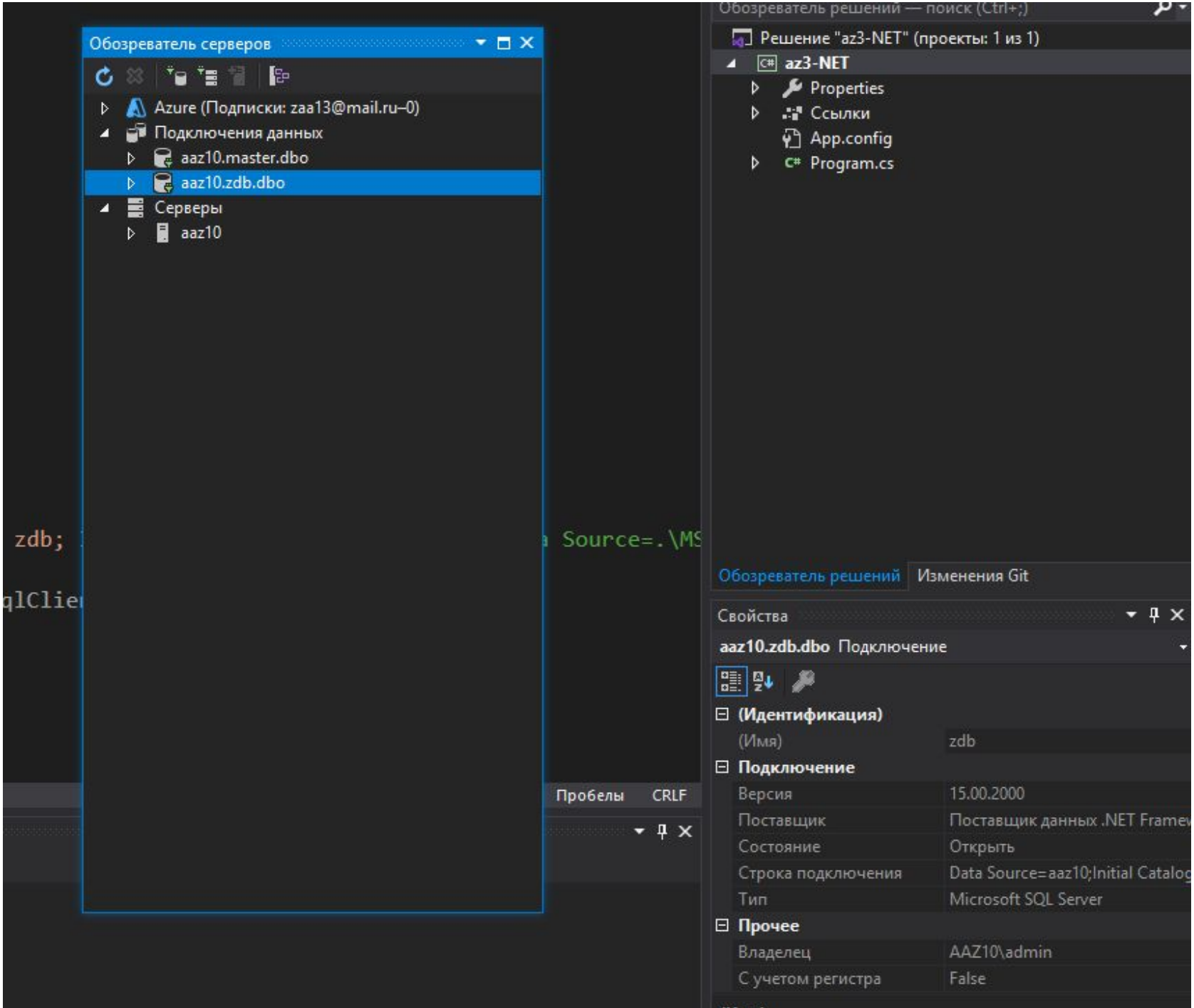
```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <startup>
        <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.6" />
    </startup>
</configuration>
```

Изменим его, добавив определение строки подключения:

```
<?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>
```

```
string connectionString =
ConfigurationManager.ConnectionStrings["DefaultConnection"].ConnectionString;
                    Console.WriteLine(connectionString);
```

Прежде всего чтобы работать с конфигурацией приложения, надо добавить в проект библиотеку **System.Configuration.dll**.
С помощью объекта ConfigurationManager.ConnectionStrings["название_строки_подключения"] можно получить строку подключения и использовать ее в приложении.


Обозреватель серверов
Azure (Подписки: zaa13@mail.ru–0)
Подключения данных
aaz10.master.dbo
aaz10.zdb.dbo
Серверы
aaz10
Обозреватель решений — поиск (Ctrl+;)
Решение "az3-NET" (проекты: 1 из 1)
az3-NET
Properties
Ссылки
App.config
Program.cs
Обозреватель решений
Изменения Git
Свойства
aaz10.zdb.dbo Подключение
(Идентификация)
(Имя) zdb
Подключение
Версия 15.00.2000
Поставщик Поставщик данных .NET Framev
Состояние Открыть
Строка подключения Data Source=aaz10;Initial Catalog
Тип Microsoft SQL Server
Прочее
Владелец AAZ10\admin
С учетом регистра False
Пробелы
CRLF

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

- **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 значение false
- **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 значение false

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 можно их передать в строку подключения через параметры user id и password:

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

Создание подключения

```
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...");
}
```

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

```
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...");
```

Получение информации о подключении

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

```
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("\tWorkstationId: {0}", connection.WorkstationId);
}
     Console.WriteLine("Подключение закрыто...");
```

Упражнение 19

1. Создать простую БД userdb, состоящую из одной таблицы с тремя столбцами: ID; Name; Age.
2. Написать программу подключения к БД, открыть ее, получить сведения об этом подключении и далее закрыть.
3. Сдать текст программы и скриншот результата ее работы. Указать исполнителя программы и номер группы.

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:

```
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

```
string connectionString = @"Data Source=.\SQLEXPRESS;Initial Catalog=usersdb;Integrated Security=True";
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.Close(); // 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);}
```

Выполнение команд и SqlCommand

```
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;
}
```

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

Или так:

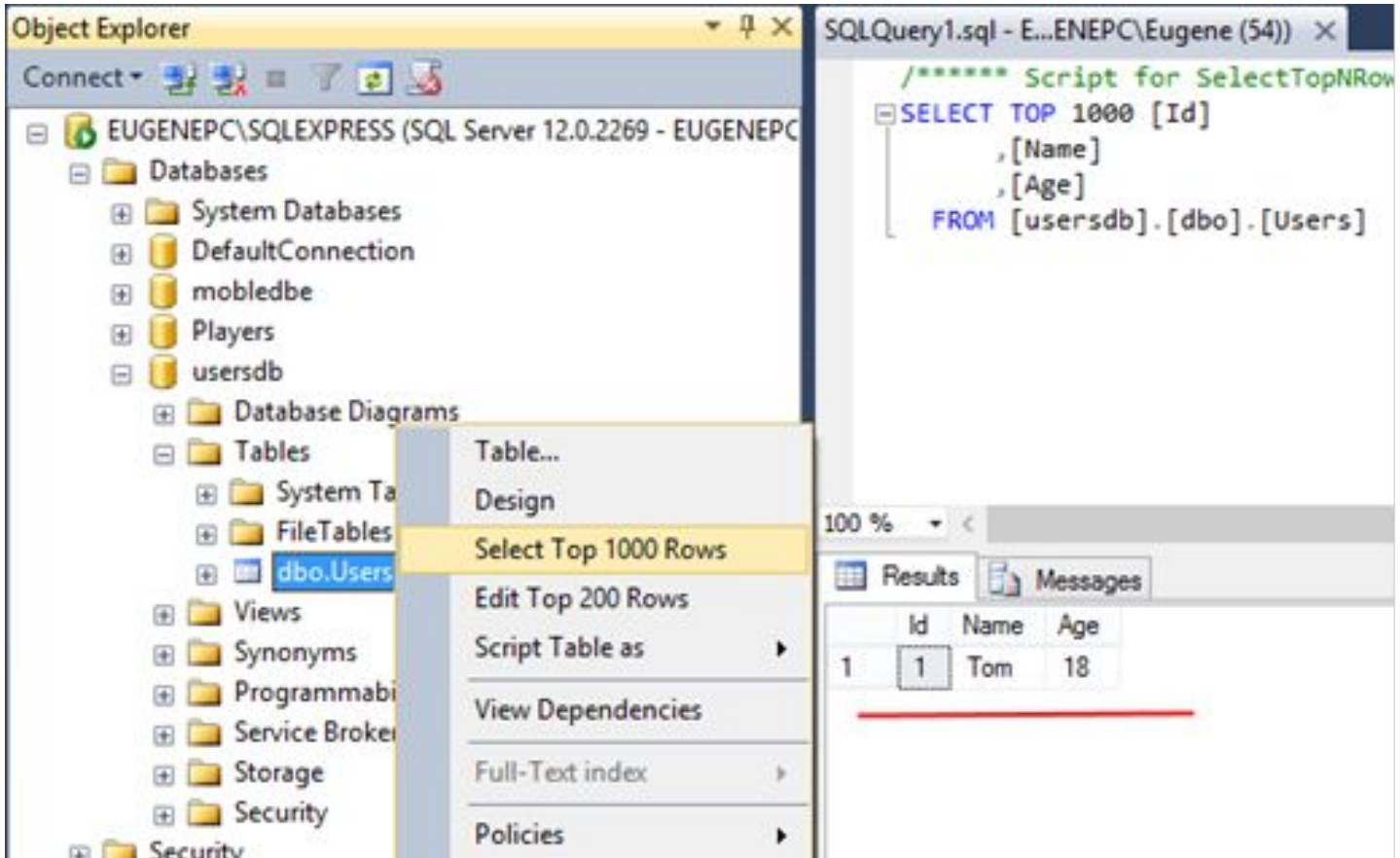
```
string connectionString = @"Data Source=.\SQLEXPRESS;Initial Catalog=usersdb;Integrated Security=True";
string sqlExpression = "SELECT * FROM Users";
using (SqlConnection connection = new SqlConnection(connectionString))
{    connection.Open();
   SqlCommand command = new SqlCommand(sqlExpression, connection);
}
```

Чтобы выполнить команду, необходимо применить один из методов SqlCommand:

- ✔ **ExecuteNonQuery**: 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

```
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);
      }    }
```



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

UPDATE название_таблицы
SET столбец1=значение1, столбец2=значение2, столбецN=значениеN
WHERE некоторый_столбец=некоторое_значение

```
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);}
```

Удаление

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

```
DELETE FROM таблица
WHERE столбец = значение
```

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

```
string connectionString = @"Data Source=.\SQLEXPRESS;Initial Catalog=usersdb;Integrated Security=True";
string sqlExpression = "DELETE  FROM 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);
}
```

Чтение результатов запроса и SqlDataReader

Если надо считывать данные, которые хранятся в таблице, то потребуется метод - **ExecuteReader**(). Этот метод возвращает объект **SqlDataReader**, который используется для чтения данных.

```
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();    }
```

Упражнение 20

1. Написать программу, в которой с консоли вводится номер, имя, возраст и записывается в БД.
2. При каждом нажатии кнопки пробел на консоль выводится содержимое таблицы в БД.
3. Для удаления записи набрать на консоли DELETE и номер удаляемой записи. Проверить результат операции.
4. Показать текст программы и скриншот результата ее работы. Указать исполнителя программы и номер группы.

Асинхронное чтение

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

```
static void Main (string[] args)
{   ReadDataAsync().GetAwaiter();}

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

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, если в третьем столбце хранится возраст пользователя, который представляет целое число, и в программе хотели бы его использовать как целое число. Так как **GetValue** возвращает объект типа object, то, чтобы его использовать, к примеру, как число, нам надо его привести к типу **int**. Для этого можно выбрать другой путь - использовать типизированные методы.

```
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);  }              }
```

здесь **GetInt32**() и **GetString**(), которые возвращают объекты типа **int** и **string** соответственно

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[]	GetBytes и GetStream
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	Object	GetValue
long varchar	String и Char[]	GetString и GetChars
timestamp	Byte[]	GetBytes

Получение скалярных значений

При отправке запросов можно 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.

```
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.

Прямая вставка записи в таблицу

```
INSERT INTO table_name (column1, column2, column3, ...)
    VALUES (value1, value2, value3, ...);
```

Или

```
INSERT INTO table_name     VALUES (value1, value2, value3, ...);
```

```
SqlConnection con = new SqlConnection(@"Data Source=.\SQLEXPRESS;Initial Catalog=ComputerSho
Integrated Security=True");
string query = "INSERT INTO Products (Name,Price,Date) VALUES('LED Screen', '$120',
'27January 2017')";
            SqlCommand cmd = new SqlCommand(query, con);
            try
            {       con.Open();
                cmd.ExecuteNonQuery();
                Console.WriteLine("Records Inserted Successfully");
            }
            catch (SqlException e)
            {  Console.WriteLine("Error Generated. Details: " + e.ToString());
            }
            finally    {                con.Close();}
```

ПАРАМЕТРИЗОВАННЫЙ ЗАПРОС

Как правило всегда предпочтительно использовать параметризованный запрос вместо простого SQL-запроса, потому что он предотвращает атаки SQL- инъекции.

```
SqlConnection con = new SqlConnection(@"Data Source=.\SQLEXPRESS;Initial Catalog=ComputerShop;Integrated Security=True");
            //Replaced Parameters with Value
string query = "INSERT INTO Products (Name, Price, Date) VALUES(@Name, @Price, @Date)";
            SqlCommand cmd = new SqlCommand(query, con);
            //Pass values to Parameters
            cmd.Parameters.AddWithValue("@Name", "USB Keyboard");
            cmd.Parameters.AddWithValue("@Price", "$20");
            cmd.Parameters.AddWithValue("@Date", "25 May 2017");
            try  {   con.Open();
                cmd.ExecuteNonQuery();
                Console.WriteLine("Records Inserted Successfully");              }
            catch (SqlException e)
{ Console.WriteLine("Error Generated. Details: " + e.ToString());              }
            finally              {                  con.Close();}
```

КОПИРОВАНИЕ ОДНОЙ ТАБЛИЦЫ В ДРУГУЮ ТАБЛИЦУ

В большинстве случаев вам нужно скопировать данные одной таблицы в другую таблицу. Здесь создан еще один элемент таблицы, который будет содержать скопированные данные из таблицы PRODUCTS.

```
SqlConnection con = new SqlConnection(@"Data Source=.\SQLEXPRESS;Initial
Catalog=ComputerShop;Integrated  Security=True");
        //Replaced Parameters with Value
        string query = "INSERT INTO Items(Name,Price,Date) SELECT Name,Price,Date
FROM Products";
        SqlCommand cmd = new SqlCommand(query, con);
        try  {
           con.Open();
           cmd.ExecuteNonQuery();
           Console.WriteLine("Records Inserted Successfully");             }
        catch (SqlException e)
        {               Console.WriteLine("Error Generated. Details: " +
e.ToString());             }
        finally
        {               con.Close();}
```

Упражнение 21

1. Создать текстовый файл с расширением .xml записать туда текст:

```
<?xml version="1.0" encoding="utf-8"?>
<phones>
  <phone name="iPhone 6">
    <company>Apple</company>
    <price>40000</price>
  </phone>
  <phone name="Samsung Galaxy S5">
    <company>Samsung</company>
    <price>33000</price>
  </phone>
</phones>
```

2. В MSSQL подготовить БД с таблицей соответствующей структуре данных прочитанной из xml файла.

3. Написать программу, которая читает xml файл, при нажатии кнопки Enter выводит на экран содержимое xml файла и заполнит БД данными из этого файла. (Для тех, кто учится не формально написать в программе процедуру создания таблицы в выбранной БД)

3. При повторном нажатии Enter прочитать содержимое БД и вывести на консоль в виде таблицы.

4. Показать текст программы, скриншот результата ее работы и скриншот таблицы в БД. Указать исполнителя программы и номер группы.

Платформа **ASP.NET Core** представляет технологию от компании Microsoft, предназначенную для создания различного рода веб-приложений: от небольших веб-сайтов до крупных веб-порталов и веб-сервисов.

ASP.NET Core является продолжением развития платформы ASP.NET, это не просто очередной релиз. ASP.NET Core фактически означает революцию всей платформы, ее качественное изменение.

ASP.NET Core теперь полностью является opensource-фреймворком. Все исходные файлы фреймворка доступны на GitHub.

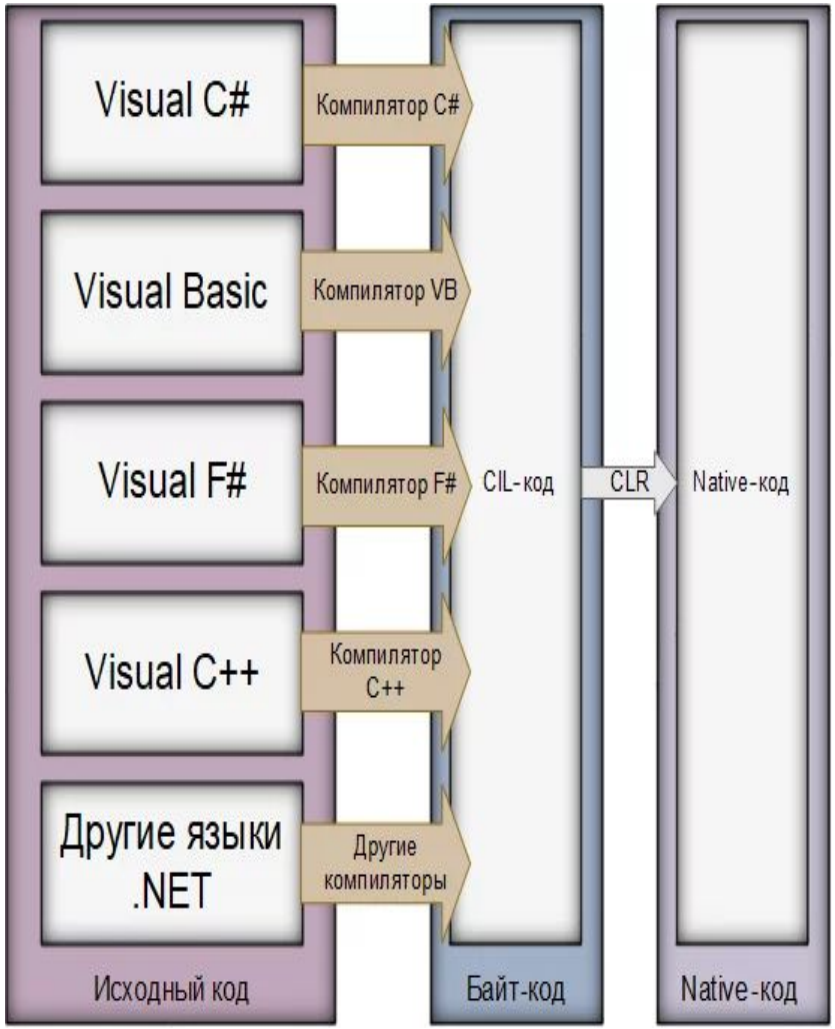
ASP.NET Core может работать поверх кросс-платформенной среды .NET Core, которая может быть развернута на основных популярных операционных системах: Windows, Mac OS, Linux. С помощью ASP.NET Core можем создавать кросс-платформенные приложения. И хотя Windows в качестве среды для разработки и развертывания приложения до сих пор превалирует, но теперь уже не ограничены только этой операционной системой. То есть можем запускать веб-приложения не только на ОС Windows, но и на Linux и Mac OS. А для развертывания веб-приложения можно использовать традиционный IIS, либо кросс-платформенный веб-сервер Kestrel.

Для чего нужен .NET

Обычным пользователям может показаться, что это какие-то программистские штуки, которые никак не влияют на их жизнь. На самом деле в этом есть смысл и для них.

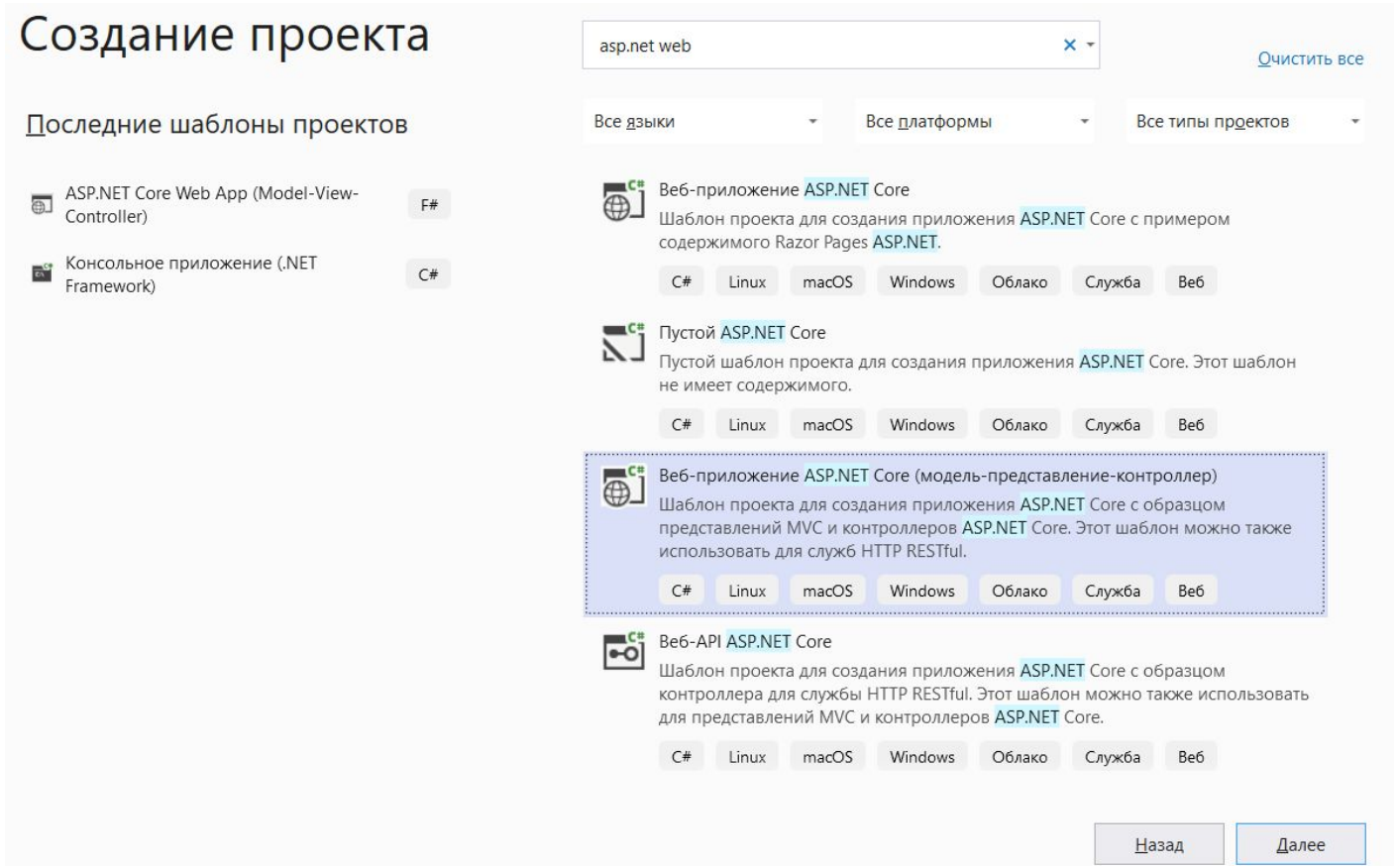
Если бы не .NET, пользователям пришлось бы устанавливать среду исполнения для программ на каждом языке. То есть чтобы запустить приложение на Visual Basic, нужно скачать среду выполнения для Visual Basic. Если же программа написана на C#, то придётся скачивать среду и для неё.

Кроме основных языков есть также и другие, которые поддерживаются *.NET*. Среди них *COBOL, Fortran, Haskell* и *Java*



CLI ***(Common Language Infrastructure*** *— общеязыковая инфраструктура)*. Она определяет, как работает *.NET* .
В *CLI* у каждого языка есть свой компилятор. Но программы компилируются не в нативный код *(исполняемый)*, а в промежуточный байт-код **CIL** ***(Common Intermediate Language*** *— общий промежуточный язык).*

Когда вы запускаете программу, написанную на одном из языков семейства *.NET,* её байт-код передаётся дальше по цепи в общеязыковую исполняющую среду CLR ***(Common Language Runtime)***. Там этот байт-код компилируется в нативный и уже начинает выполняться.
Почти по такому же принципу работает виртуальная машина Java, но программы на *.NET* быстрее запускаются, что делает их пригодными для работы не только на сервере, но и на персональных компьютерах.


Создание проекта
asp.net web
Очистить все
Последние шаблоны проектов
Все языки
Все платформы
Все типы проектов
ASP.NET Core Web App (Model-View-Controller)
F#
Консольное приложение (.NET Framework)
C#
Веб-приложение ASP.NET Core
Шаблон проекта для создания приложения ASP.NET Core с примером содержимого Razor Pages ASP.NET.
C# Linux macOS Windows Облако Служба Веб
Пустой ASP.NET Core
Пустой шаблон проекта для создания приложения ASP.NET Core. Этот шаблон не имеет содержимого.
C# Linux macOS Windows Облако Служба Веб
Веб-приложение ASP.NET Core (модель-представление-контроллер)
Шаблон проекта для создания приложения ASP.NET Core с образцом представлений MVC и контроллеров ASP.NET Core. Этот шаблон можно также использовать для служб HTTP RESTful.
C# Linux macOS Windows Облако Служба Веб
Веб-API ASP.NET Core
Шаблон проекта для создания приложения ASP.NET Core с образцом контроллера для службы HTTP RESTful. Этот шаблон можно также использовать для представлений MVC и контроллеров ASP.NET Core.
C# Linux macOS Windows Облако Служба Веб
Назад
Далее

Blazor представляет UI-фреймворк (User interface) для создания интерактивных приложений, которые могут работать как на стороне сервера, так и на стороне клиента, на платформе .NET. В своем развитии фреймворк Blazor испытал большое влияние современных фреймворков для создания клиентских приложений - Angular, React, VueJS. В частности, это проявляется в роли компонентов при построении пользовательского интерфейса. В то же время **и на стороне клиента, и на стороне сервера при определении кода в качестве языка программирования применяется C#, вместо JavaScript.** А для описания визуального интерфейса используются стандартные HTML и CSS.

Blazor предоставляет разработчикам следующие преимущества:

- Написание кода веб-приложений с помощью C# вместо JavaScript
- Использование возможностей экосистемы .NET, в частности, библиотек .NET при создании приложений, безопасности и производительности платформы .NET
- Клиентская и серверная части приложения могут использовать общую логику
- Использование Visual Studio в качестве инструмента для разработки, который имет встроенные шаблоны для упрощения создания приложения

Функционально на текущий момент Blazor подразделяется на две подсистемы:

- **Blazor Server**: позволяет создавать серверные приложения и поддерживается ASP.NET Core
- **Blazor WebAssembly**: позволяет создавать одностраничные интерактивные приложения клиентской стороны, которые запускаются в браузере пользователя и работают с помощью технологии WebAssembly

Blazor WebAssembly

Blazor WebAssembly позволяет создавать интерактивные одностраничные приложения, которые запускаются на браузере пользователя с помощью технологии WebAssembly. При построении и запуске приложения Blazor WebAssembly файлы с кодом C# и Razor компилируются в сборки .NET. Затем Blazor WebAssembly (а если точнее скрипт **blazor.webassembly.js**) загружает среду выполнения .NET, сборки и их зависимости и настраивает среду выполнения .NET для выполнения сборок. Посредством взаимодействия с JavaScript фреймворк Blazor WebAssembly может обращаться к DOM и API браузера.
Одним из преимуществ Blazor WebAssembly является то, что он может оптимизировать загружаемые сборки. В частности, при публикации приложения неиспользуемый код убирается линкером (компоновщиком) IL (Intermediate Language). Все необходимые файлы среды выполнения .NET и загружаемых сборок кэшируются в браузере.
При этом Blazor WebAssembly не зависит от сервера.

Все необходимые файлы загружаются браузером, и после загрузки файлов приложение работает полностью на стороне браузера и совершенно не зависит от сервера