Урок 7

Взаимодействие с базой данных

Взаимодействие с базой данных. Обзор технологии ADO.Net.

Взаимодействие с базой данных. Обзор технологии ADO.NET

ADO.NET – это семейство технологий, которые позволяют разработчикам .NET-приложений взаимодействовать с данными, используя стандартные и структурированные подходы.

Библиотека **ADO.Net** подключается к проекту добавлением ссылки на пространство имен **System.Data**.

ADO.NET осуществляет управление как внутренними данными (созданными в памяти компьютера и использующимися внутри приложения), так и внешними, находящимися вне приложения – например, в базе данных или текстовых файлах.

Вне зависимости от источника данных, **ADO.NET** представляет данные в коде приложения в табличном виде, в виде строк и столбцов.

До выпуска компанией Microsoft .NET-фреймворка одной из основных технологий доступа к данным, используемой в приложениях, было **ADO** (ActiveX Data Object). После выхода **.NET**-фреймворка **ADO.NET** стала преемницей **ADO**.

При взаимодействии с внешними источниками данных **ADO.NET** может использовать технологию работы с отсоединенными данными. При использовании более ранних технологий разработчики обычно создавали постоянное подключение к БД и использовали различные способы блокировки записей, чтобы безопасно и корректно изменять данные. Но с приходом эры Интернета стратегия поддержания открытых соединений для каждого из множества одновременных HTTP-запросов к веб-приложению показала свою нежизнеспособность. Для **ADO.NET** предпочтительной стратегией является открытое подключение до запроса к БД и его моментальное закрытие после выполнения запроса.

Основные компоненты ADO.NET

Пространство имен **System.Data** включает множество отдельных классов **ADO.NET**, которые работают вместе и обеспечивают доступ к табличным данным. Библиотека **ADO.NET** включает в себя две группы классов: для работы с данными внутри приложения и с внешними данными.

Центральным объектом библиотеки является **DataTable**. Схожий по назначению с таблицами в БД, **DataTable** управляет актуальными данными, с которыми вы работаете. Каждый объект **DataTable** содержит 0 или более строк данных.

Таблицы содержат описание элементов **DataColumn**, каждый из которых описывает значения, хранящиеся в строках таблиц. **DataColumn** содержит описание типа данных, хранящейся в колонке информации. Каждой строке в таблице соответствует объект **DataRow**. **ADO.NET** содержит методы для добавления, удаления, изменения и получения отдельной строки **DataTable**. Для таблиц, связанных с внешними источниками данных, любые изменения могут быть повторены во внешних источниках. Имеется возможность установления связей между **DataTable** с использованием объектов **DataRelation**. С помощью объектов **Constraint** произвольные ограничения могут быть наложены на таблицы и составляющие их данные. Объект **DataView** реализует представление в отдельной **DataTable**. Таблицы могут быть объединены в **DataSet**. Однако, если планируется использовать в работе только одну таблицу БД, эффективнее ограничиться использованием **DataTable**.

Для соединения с внешними данными, находящимися в БД, **ADO.NET** содержит множество провайдеров данных. Для соединения с БД, не имеющих собственного провайдера, используются общие провайдеры **ODBC** или **OLEDB**, которые тоже входят в состав **ADO.NET**. Некоторые провайдеры данных распространяются самими производителями БД. Так обстоит ситуация с провайдером **Oracle**. Все взаимодействия с внешними источниками данных осуществляются с помощью объекта **Connection**. **ADO.NET** использует технологию пула подключений для повышения скорости соединения с БД.

SQL-запросы помещаются в объекты **Command** для отправки источнику данных. Объект **Command** может содержать необязательные объекты **Parameter**, которые позволяют вызывать хранимые процедуры или параметризованные запросы.

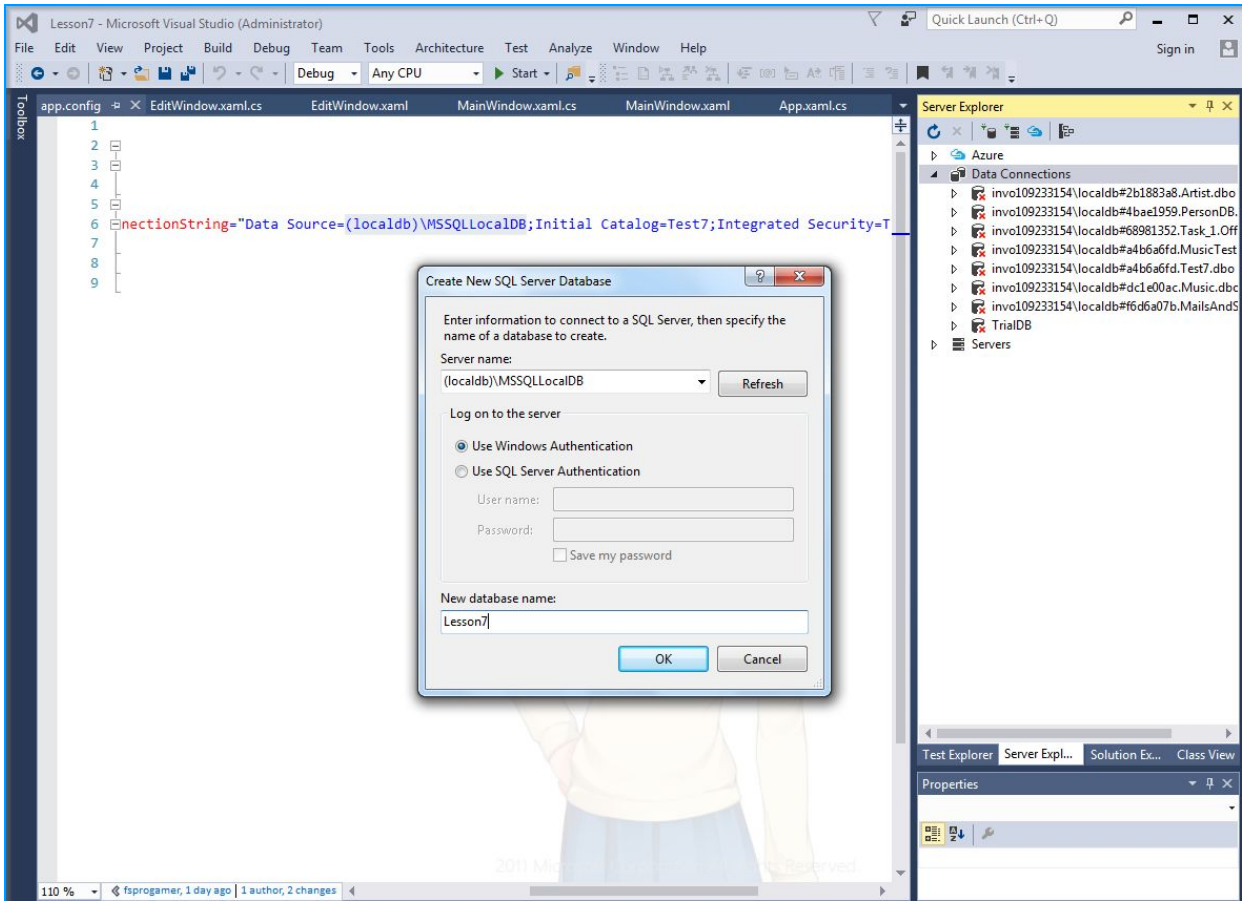
Объект **DataAdapter** содержит в себе стандартные запросы для взаимодействия с БД, тем самым исключая необходимость написания однообразных запросов, требующихся для чтения или записи отдельных строк.

Объект **DataReader** поддерживает быстрое чтение данных без возможности внесения изменений.

Создание тестовой БД MSSQL Server

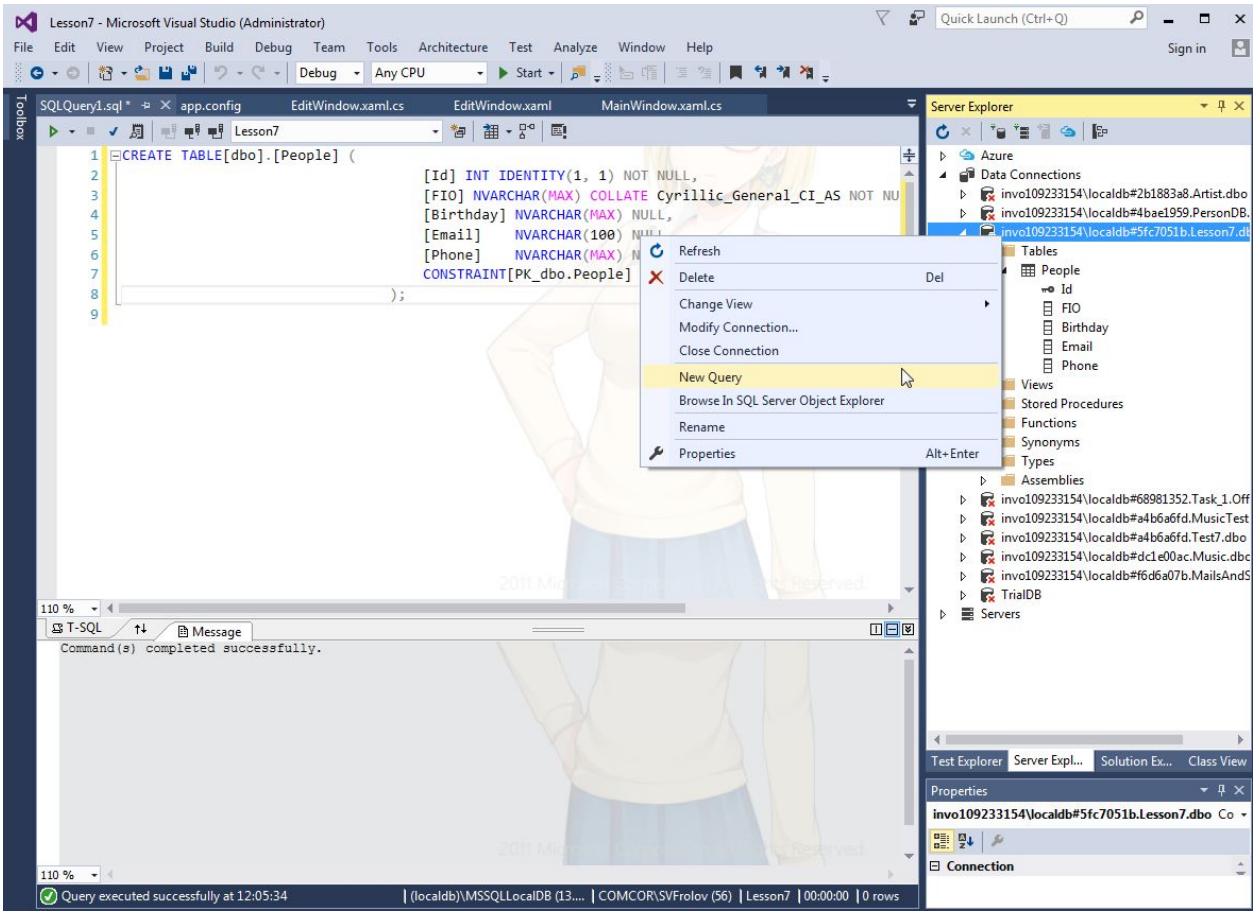
С помощью **VisualStudio** на закладке **ServerExplorer** создаем экземпляр локальной базы данных.

Название базы данных – **Lesson7**, имя сервера **(localdb)\MSSQLLocalDB**.



Создаем таблицу **People** в базе данных **Lesson7**.

```
CREATE TABLE[dbo].[People] (
                                    [Id] INT IDENTITY(1, 1) NOT NULL,
                                    [FIO] NVARCHAR(MAX) COLLATE
Cyrillic_General_CI_AS NOT NULL,
                                    [Birthday] NVARCHAR(MAX) NULL,
                                    [Email]    NVARCHAR(100) NULL,
                                    [Phone]    NVARCHAR(MAX) NULL,
                                   CONSTRAINT[PK_dbo.People] PRIMARY KEY
CLUSTERED([Id] ASC)
                                );
```



Установление соединения с внешним источником данных

Библиотека **ADO.NET** обеспечивает доступ к множеству различных внешних источников данных. Этими источниками могут быть как локальные файлы известных форматов, так и удаленные реляционные базы различных производителей. Для доступа к таким источникам данных приложение должно иметь возможность указать расположение ресурса, сообщить формат данных и обеспечить передачу параметров для авторизации доступа к данным. Такого рода информация передается с помощью строки подключения: форматированной текстовой строки, которая описывает необходимые параметры подключения.

Строка подключения содержит множество элементов, разделенных символом «;». Каждый элемент представляет собой пару «ключ-значение», которые описывают один из требуемых параметров подключения. Синтаксис строки: **key1=value1;…keyN= valueN**.

Типичные элементы:

- описание расположения БД (файловое или сетевое);
- идентификатор пользователя и пароль для доступа к источнику данных;
- значение таймаута для выполнения длительных запросов и другие значения, требующиеся для установления и конфигурирования подключения.

На сайте http://www.connectionstrings.com представлены примеры строк подключения практически ко всем БД и другим источникам данных.

Строка подключения к MS SQL Server

```
connectionString="data source=(LocalDb)\MSSQLLocalDB;initial
catalog=Lesson7;integrated security=True;providerName="System.Data.SqlClient"
```

Три ключевые параметра позволяют **ADO.NET** осуществлять подключение:

- **data source** указывает расположение сервера, к которому осуществляется доступ. Специальное выражение **LocalDB** сообщает **ADO.NET**, что БД **MS SQL** расположена на локальной рабочей станции;
- **initial catalog** указывает название БД на сервере, которая будет использоваться по умолчанию;
- **integrated security**. Если значение параметра равно **true**, **ADO.NET** будет использовать авторизацию Windows для доступа к БД. Если значение равно **false**, нужно добавить в строку подключения пары «ключ-значение» для **User ID** (имя пользователя на SQL Server) и **Password** (пароль пользователя).

Наиболее предпочтительным местом для хранения строки подключения является конфигурационный файл приложения.

Подключение к SQLServer с помощью провайдера данных

Для подключения к **SQLServer** с помощью ADO.NET требуется три компонента: работающая БД **SQLServer**, объект **SqlConnection** и правильная строка подключения.

```
string connectionString = @"Data Source = (localdb)\MSSQLLocalDB; Initial Catalog
= Lesson7; Integrated Security = True";

using (SqlConnection connection = new SqlConnection(connectionString))
{
     connection.Open();
      …
}
```

Пул подключений

Традиционные клиент-серверные десктопные приложения обычно устанавливают соединение с БД при старте и держат подключение открытым до закрытия приложения. Однако для остальных типов приложений, особенно веб, рекомендованной стратегией является кратковременное подключение к БД (на время выполнения запроса к ней) и немедленное закрытие соединения. При использовании такой стратегии **ADO.NET** позволяет сократить время открытия подключения к БД за счет пула подключений, т.е повторного использования объектов **Connection**, связанных с ранее закрытыми подключениями к БД.

SQLServer поддерживает отдельные пулы для разных строк подключения. Каждый пул подключений может содержать более одного экземпляра активных подключений.

Возможно отключение пула для определенной строки подключений с помощью добавления в нее выражения **Pooling=false**.

Запросы данных

Язык SQL является универсальным средством взаимодействия с реляционными БД. Хотя большинство СУБД включают в себя средства управления данными и таблицами, возможности этих средств могут быть реализованы с помощью запросов SQL. Начиная с создания таблиц и заканчивая запросами к множеству таблиц, SQL включает в себя достаточный набор типов данных и операторов для управления БД и ее содержимым.

Операторы SQL включают в себя:

- операторы SQL-запросов – для получения данных;
- операторы управления данными – для изменения данных БД;
- операторы определения данных – команды для изменения таблиц и других структур данных;
- хранимые процедуры – именованные блоки исполняемого кода.

ADO.NET позволяет реализовать все перечисленные типы операторов с помощью класса **System.Data.SqlClient.SqlCommand**. Класс реализует один или более SQL-операторов.

SQLCommand

Последовательность операций для вызова **SqlCommand**:

- Создать объект **SqlCommand**;
- Присвоить текст SQL-оператора свойству **CommandText**;
- Присвоить объект **SqlConnection** для уже открытого подключения свойству **Connection**;
- Установить дополнительные параметры, если необходимо;
- Вызвать один из синхронных или асинхронных методов «**Execute…**».

Свойство **CommandText** может принимать два типа данных:

- Операторы **SQL (CommandType.Text)** – значение по умолчанию. Обычно применяется один SQL-запрос, но поддерживается и использование нескольких, разделенных точкой с запятой;
- Хранимые процедуры (**CommandType.StoredProcedure**) – текст, содержащий название хранимой процедуры.

ExecuteNonQuery

Метод **ExecuteNonQuery** отправляет значение **CommandText** источнику данных через заранее открытое подключение. Любые ошибки, включая сформированные источником данных, вызывают исключение. Метод **ExecuteNonQuery** возвращает число обработанных строк или -1, если ни одна строка не обработана.

Подключение, которое используется для выполнения SQL-запроса, должно оставаться открытым в течение выполнения SQL-запроса. Чтобы прервать выполнение запроса, необходимо вызвать метод **Cancel** класса **SqlCommand**.

```
string createExpression = @"CREATE TABLE[dbo].[People] (
                                    [Id] INT IDENTITY(1, 1) NOT NULL,
                                    [FIO] NVARCHAR(MAX) COLLATE
Cyrillic_General_CI_AS NOT NULL,
                                    [Birthday] NVARCHAR(MAX) NULL,
                                    [Email]    NVARCHAR(100) NULL,
                                    [Phone]    NVARCHAR(MAX) NULL,
                                   CONSTRAINT[PK_dbo.People] PRIMARY KEY
CLUSTERED([Id] ASC)
                                );";
string createStoredProc = @"CREATE PROCEDURE [dbo].[sp_GetPeople] AS SELECT * FROM
People;";
            string sqlExpression = @"INSERT INTO People (FIO,
Birthday,Email,Phone) VALUES ( N'Иванов Иван Иванович', '18.10.2001',
'somebody@gmail.com', '89164444444' );
                                     INSERT INTO People(FIO, Birthday, Email,
Phone) VALUES( N'Петров Петр Петрович', '15.01.2001', 'somebody@mail.com',
'8916555555')";

            using (SqlConnection connection = new SqlConnection(connectionString))
            {
                connection.Open();

                SqlCommand command = new SqlCommand(createExpression, connection);
                int number = command.ExecuteNonQuery();

                SqlCommand command = new SqlCommand(createStoredProc, connection);
                int number = command.ExecuteNonQuery();

                command = new SqlCommand(sqlExpression, connection);
                number = command.ExecuteNonQuery();
            }
```

ExecuteScalar

Метод **ExecuteScalar** так же отправляет значение **CommandText** источнику, как **ExecuteNonQuery**. Кроме этого, метод позволяет вернуть единственное значение, сформированное запросом. **ExecuteScalar** возвращает значение типа **System.Object**, поэтому требуется явное приведение к ожидаемому типу данных.

```
string sqlExpression = "SELECT COUNT(*) FROM People";
            using (SqlConnection connection = new SqlConnection(connectionString))
            {
                connection.Open();
                SqlCommand command = new SqlCommand(sqlExpression, connection);
                int cnt = Convert.ToInt32(command.ExecuteScalar());
            }
```

Метод **ExecuteScalar** позволяет возвращать значение определенного поля (обычно первичного ключа) из добавленной записи.

```
            string  sqlExpression = @"INSERT INTO People (FIO,
Birthday,Email,Phone) output INSERTED.ID VALUES ('Сидоров Сидор Сидорович',
'16.10.2007', 'somebody@gmail.com', '89164444444' );";
             using (SqlConnection connection = new SqlConnection(connectionString))
             {
                 connection.Open();
                 SqlCommand command = new SqlCommand(sqlExpression, connection);
                 var vId = Convert.ToInt32(command.ExecuteScalar());
             }
```

ExecuteReader

Для получения результата SQL-запроса или хранимой процедуры, состоящего из одной или более строк, необходимо использовать метод **ExecuteReader** класса **SqlCommand**. Этот метод возвращает объект типа **System.Data.SqlClient.SqlDataReader**, который позволяет осуществлять построчное чтение полученного результата.

Для создания **SqlDataReader** необходимо добавить текст SQL-запроса и подключение в объект **SqlCommand**, а затем вызвать метод **ExecuteReader**.

```
string sqlExpression = "SELECT * FROM People";
            using (SqlConnection connection = new SqlConnection(connectionString))
            {
                connection.Open();
                SqlCommand command = new SqlCommand(sqlExpression, connection);
                SqlDataReader reader =
command.ExecuteReader(CommandBehavior.CloseConnection);

                if (reader.HasRows)         // Если есть данные
                {
                    while (reader.Read())   // Построчно считываем данные
                    {
                        var vId = Convert.ToInt32(reader.GetValue(0));
                        var vFIO = reader.GetString(1);
                        var vEmail = reader["Email"];
                       var vPhone = reader.GetString(reader.GetOrdinal("Phone"));
                    }
                }
                reader.Close();
            }
string sqlExpression = "[dbo].[sp_GetPeople]";
            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;
                SqlDataReader reader =
command.ExecuteReader(CommandBehavior.CloseConnection);

                if (reader.HasRows)        // Если есть данные
                {
                    while (reader.Read()) // Построчно считываем данные
                    {
                        var vId = Convert.ToInt32(reader.GetValue(0));
                        var vFIO = reader.GetString(1);
                        var vEmail = reader["Email"];
                       var vPhone = reader.GetString(reader.GetOrdinal("Phone"));
                    }
                }
                reader.Close();
            }
```

SqlDataReader при каждом чтении возвращает одну строку в виде коллекции значений колонок. Для получения первой и каждой последующей строки необходимо вызывать метод **Read**. Он возвращает **false**, если все строки результата запроса прочитаны.

Свойство **HasRows** показывает, содержит ли результат запроса хотя бы одну строку.

Для прекращения работы с **SqlDataReader** необходимо всегда вызывать методы **Close** или **Dispose**. **SqlServer** не позволяет использовать одновременно больше одного **SqlDataReader**. Выполнение **ExecuteNonQuery** одновременно с **SqlDataReader** также невозможно.

После закрытия **SqlDataReader связанное** с ним подключение остается открытым. Но если создавать объект SqlDataReader с параметром **CommandBehavior.CloseConnection**, подключение будет

закрываться автоматически после закрытия **SqlDataReader**. **SqlDataReader reader = command.ExecuteReader(CommandBehavior.CloseConnection)**.

SqlDataReader осуществляет однонаправленное чтение данных. После прохода по всем записям с помощью метода **Read** невозможно вернуться к первой строке результатов SQL-запроса и повторить проход. Для этого придется создать новый **SqlDataReader**.

Доступ к отдельным значениям в строках результата SQL-запроса осуществляется по индексу или по названию. Возвращаемые значения имеют тип **Object** и требуют приведения типов для присвоения переменным в коде.

Поля могут содержать значение **DBNull.Value**. Для проверки на наличие в поле такого значения у **SqlDataReader** существует метод **IsDBNull**.

Для строго типизированного доступа к значениям полей **SqlDataReader** представляет множество методов чтения данных с названиями, соответствующими возвращаемым типам данных:

- GetBoolean;
- GetByte;
- GetBytes;
- GetChar;
- GetChars;
- GetDateTime;
- GetDateTimeOffset;
- GetDouble;
- GetFloat;
- GetGuid;
- GetInt16;
- GetInt32;
- GetInt64;
- GetString;
- GetTimeSpan.

Метод чтения данных должен соответствовать типу данных поля, из которого извлекаются данные.

SqlParameter

Передача параметров в SQL-запросы и хранимые процедуры реализуется с помощью класса **System.Data.SqlClient.SqlParameter**. **SqlCommand** содержит коллекцию объектов **SqlParameter**. Каждый параметр SQL-запроса должен быть объявлен в виде объекта **SqlParameter** с указанием необходимых параметров и добавлен в коллекцию **SqlCommand.Parameters**. Во время выполнения запроса **ADO.NET** выполняет передачу текста SQL-запроса и коллекции параметров в БД для обработки.

Каждый параметр включает в себя: название параметра (совпадающее с именем параметра в SQL-запросе), тип данных параметра (для строковых параметров необходимо указать максимальную длину строки), значение параметра.

Чтобы добавить параметр в запрос, необходимо создать объект **SqlParameter** и добавить его ссылку в объект **SqlCommand**.

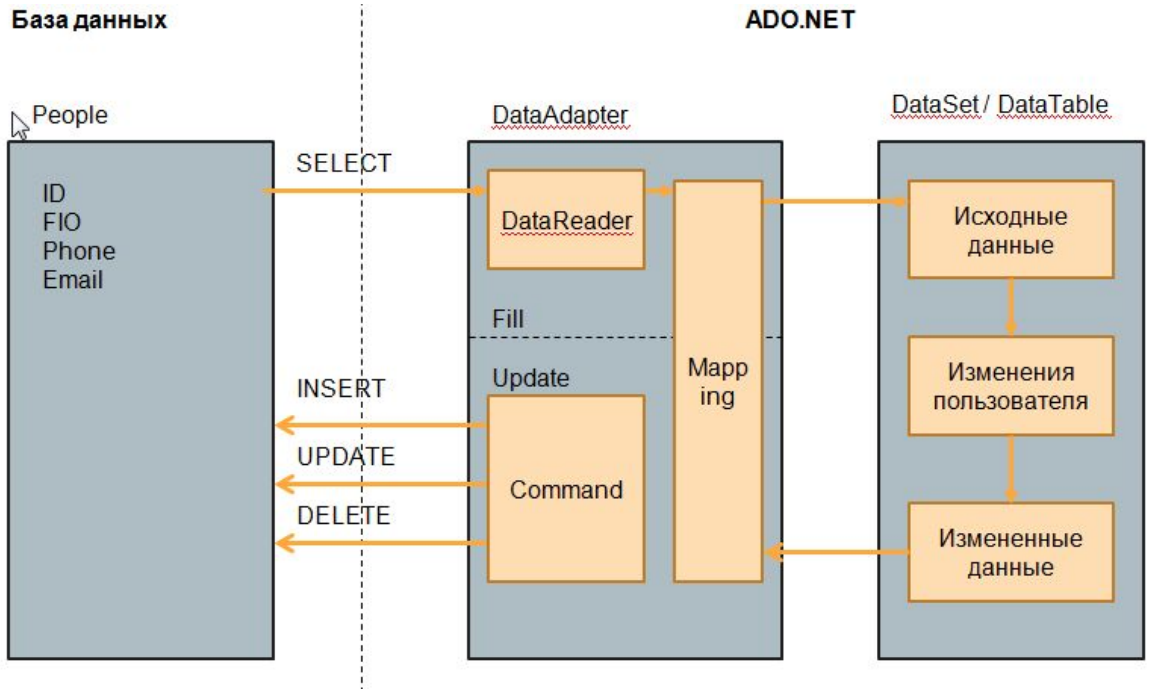
```
string sqlWhereExpression = "SELECT COUNT(*) FROM People where Birthday =
@Birthday";
            using (SqlConnection connection = new SqlConnection(connectionString))
            {
                connection.Open();
                SqlCommand command = new SqlCommand(sqlWhereExpression,
connection);
                SqlParameter param = new SqlParameter("@Birthday",
SqlDbType.NVarChar, -1);
                param.Value = "18.10.2001";
                command.Parameters.Add(param);

                var vId = Convert.ToInt32(command.ExecuteScalar());
            }
```

Существует упрощенный вариант добавления параметра в коллекцию **SqlCommand.Parameters**. Это метод **AddWithValue**, который принимает в качестве параметров имя параметра и его значение.

```
using (SqlConnection connection = new SqlConnection(connectionString))
      {
         connection.Open();
         SqlCommand command = new SqlCommand(sqlWhereExpression, connection);
         command.Parameters.AddWithValue("@Birthday", "18.10.2001");

         var vId = Convert.ToInt32(command.ExecuteScalar());
      }
```

DataAdapter

Класс **DataAdapter** с помощью SQL-запросов связывает таблицы во внешней БД с локальными таблицами **DataTable**, входящими в **DataSet**. Всякий раз, когда требуется передать данные из БД в **DataSet**, **DataAdapter** выполняет метод **Fill**, который отправляет SQL-запрос в БД и помещает результат запроса в **DataTable**. После этого можно выполнить изменение данных в **DataTable**. Когда же придет время сохранять данные из **DataSet** в БД, **DataAdapter** выполнит метод **Update**, который отправит подходящий запрос **INSERT**, **UPDATE**, **DELETE** в БД, чтобы привести данные БД в соответствие с их локальной копией.



System.Data.SqlClient.SqlDataAdapter представляет собой реализацию **DataAdapter** для **SQLServer**.

SqlDataAdapter поддерживает три основные возможности:

- Получение записей – **SqlDataAdapter** с помощью объекта **DataReader** получает записи из БД. Для этого необходимо указать запрос **SELECT** и строку подключения к БД;
- Изменение записей – для изменения данных в БД требуется указать соответствующие операторы **INSERT**, **UPDATE**, **DELETE**. Операторы могут быть указаны вручную или сформированы автоматически на основе оператора **SELECT**.
- Связывание имен таблиц и столбцов – **SqlDataAdapter** содержит информацию о соответствии имен таблиц и столбцов в БД с именами таблиц и колонок в локальном **DataSet** или **DataTable**.

```
string sql = "SELECT * FROM People";
          using (SqlConnection connection = new SqlConnection(connectionString))
           {
               SqlDataAdapter adapter = new SqlDataAdapter();
               adapter.SelectCommand = new SqlCommand(sql, connection);
               DataTable dt = new DataTable();
               adapter.Fill(dt);
           }
```

Если подключение к БД еще не открыто, метод **Fill** открывает подключение и закрывает его после получения результата.

```
        string sql = "SELECT * FROM People";
          SqlDataAdapter adapter = new SqlDataAdapter(sql, connectionString);
          DataTable dt = new DataTable();
          adapter.Fill(dt);
```

На основании полученных из БД данных **SqlDataAdapter** создает структуру объекта **DataTable**, в которую помещаются данные.

Аналогично с помощью **SqlDataAdapter** возможно заполнение данными объекта **DataSet**.

```
string sql = "SELECT * FROM People";
  SqlDataAdapter adapter = new SqlDataAdapter(sql, connectionString);
  DataSet ds = new DataSet();
  adapter.Fill(ds);
```

Метод **Fill** при использовании с **DataSet** позволяет загрузить данные из нескольких таблиц одновременно – если запрос в **SelectCommand** содержит несколько выражений **SELECT** или вызов хранимой процедуры, возвращающей несколько наборов данных.

Когда данные в **DataTable** изменены, тот же самый **SqlDataAdapter**, с помощью которого были получены данные, может отправить их обратно в БД.

Свойство **SqlDataAdapter.SelectCommand** управляет только движением данных из внешнего источника в **DataTable** или **DataSet**. Для движения данных в обратном направлении требуется задать три свойства **SqlDataAdapter**: **InsertCommand**, **UpdateCommand** и **DeleteCommand**. Каждое из них содержит SQL-запрос или вызов хранимой процедуры, ссылку на **SqlConnection** и параметры. Если для **SelectCommand** параметры являются необязательным, то для **InsertCommand**, **UpdateCommand** и **DeleteCommand** параметры являются неотъемлемой частью.

```
        SqlDataAdapter adapter = new SqlDataAdapter();
                SqlCommand command = new SqlCommand("SELECT * FROM People",
connection);
                adapter.SelectCommand = command;
                // insert
                command = new SqlCommand(@"INSERT INTO People (FIO, Birthday,
Email, Phone) VALUES (@FIO, @Birthday, @Email, @Phone); SET @ID = @@IDENTITY;",
connection);
                command.Parameters.Add("@FIO", SqlDbType.NVarChar, -1, "FIO");
                command.Parameters.Add("@Birthday", SqlDbType.NVarChar, -1,
"Birthday");
                command.Parameters.Add("@Email", SqlDbType.NVarChar, 100,
"Email");
                command.Parameters.Add("@Phone", SqlDbType.NVarChar, -1, "Phone");
                SqlParameter param = command.Parameters.Add("@ID", SqlDbType.Int,
0, "ID");
                param.Direction = ParameterDirection.Output;
                adapter.InsertCommand = command;
                // update
                command = new SqlCommand(@"UPDATE People SET FIO = @FIO, Birthday
= @Birthday WHERE ID = @ID", connection);
                command.Parameters.Add("@FIO", SqlDbType.NVarChar, -1, "FIO");
                command.Parameters.Add("@Birthday", SqlDbType.NVarChar, -1,
"Birthday");
                param = command.Parameters.Add("@ID", SqlDbType.Int, 0, "ID");
                param.SourceVersion = DataRowVersion.Original;
                adapter.UpdateCommand = command;
                // delete
                command = new SqlCommand("DELETE FROM People WHERE ID = @ID",
connection);
                param = command.Parameters.Add("@ID", SqlDbType.Int, 0, "ID");
                param.SourceVersion = DataRowVersion.Original;
```

После внесения изменений в локальной **DataTable**, связанной с **SqlDataAdapter**, достаточно вызвать метод **Update** для переноса изменений во внешнюю БД. В качестве параметра метода **Update** может быть использована **DataTable**, **DataSet** или массив **DataRow**.

Метод **Update** проверяет каждую запись в переданном массиве данных, определяя, нужно ли и для каких записей выполнять **INSERT**, **UPDATE**, **DELETE**. Для каждой записи, которая нуждается в изменении, **DataAdapter** формирует событие **OnRowUpdating** (перед внесением изменений) и **OnRowUpdated** (после).

Ошибки, возникающие в процессе изменения данных, вызывают исключение.

Существует возможность на основе SELECT-запроса SQL автоматически сформировать запросы **INSERT**, **UPDATE**, **DELETE** для **SqlDataAdapter**. Эта возможность реализуется с помощью **SqlCommandBuilder**.

Для его использования достаточно создать **SqlDataAdapter**, указав **SelectCommand**. Затем необходимо создать **SqlCommandBuilder** и передать в него ссылку на **SqlDataAdapter**.

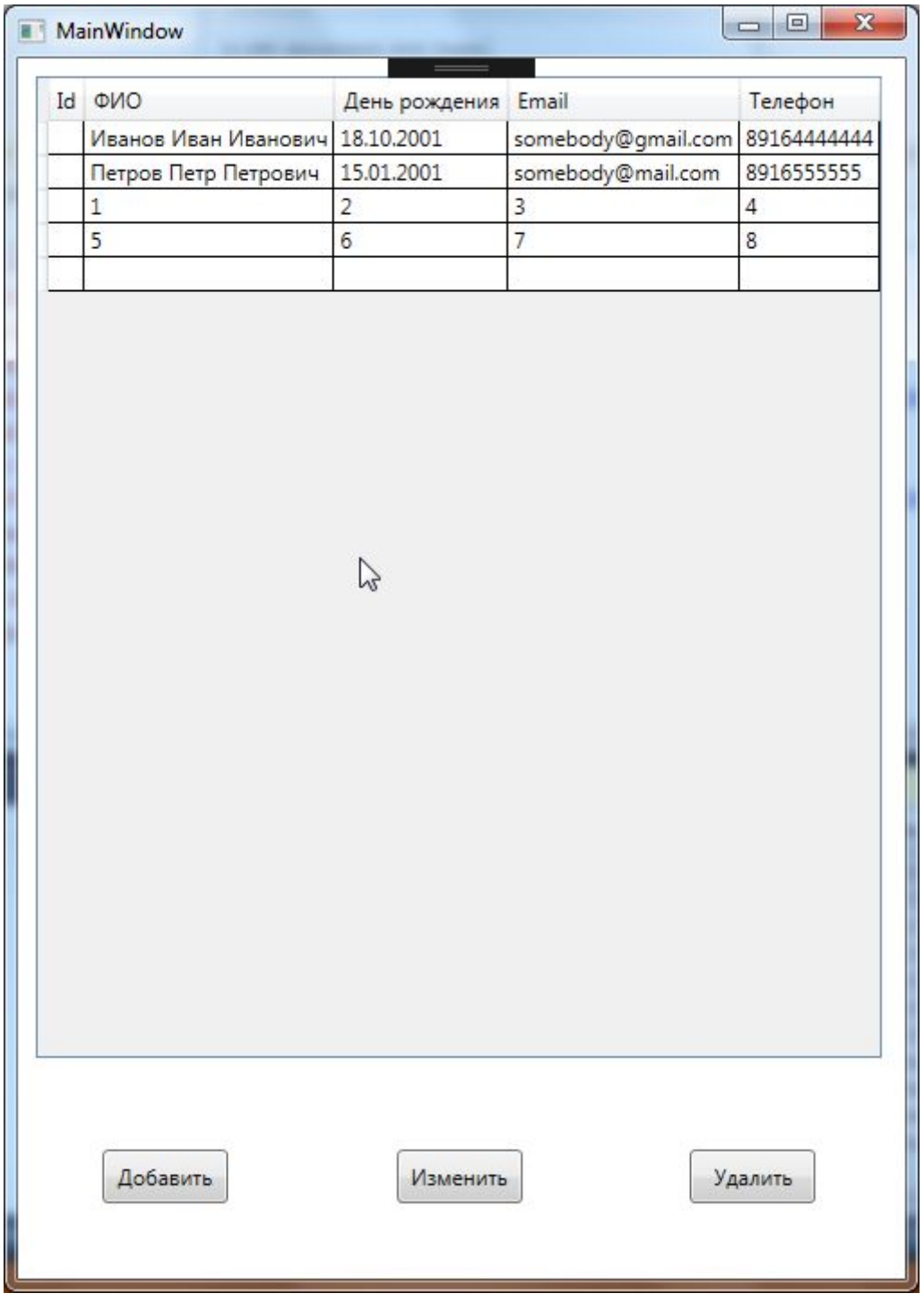
```
using (SqlConnection connection = new SqlConnection(connectionString))
 {
                SqlDataAdapter adapter = new SqlDataAdapter(sql, connection);
                DataSet ds = new DataSet();
                adapter.Fill(ds);

                DataTable dt = ds.Tables[0];
                // Добавляем новую строку
                DataRow newRow = dt.NewRow();
                newRow["FIO"] = "Александров Александр Александрович";
                newRow["Birthday"] = "25.04.2002";
                dt.Rows.Add(newRow);

                // Создаем объект SqlCommandBuilder
                SqlCommandBuilder commandBuilder = new SqlCommandBuilder(adapter);
                adapter.Update(ds);
}
```

SqlCommandBuilder автоматически формирует запросы **INSERT**, **UPDATE**, **DELETE** на основе запроса **SELECT** и структуры данных, полученных из БД.

Есть несколько ограничений на использование **SqlCommandBuilder**:

- **SqlCommandBuilder** может быть использован только с запросами к одной таблице;
- структура запрашиваемых данных должна содержать как минимум один первичный или уникальный ключ;
- если изменяется **SelectCommand**, связанная с **SqlDataAdapter** необходимо вызвать метод **SqlCommandBuilder.RefreshSchema**, чтобы повторно сформировать запросы на изменение;
- **SqlCommandBuilder** формирует только еще не определенные запросы на изменение.

Пример использования, проект ADO



App.config

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <configSections>
    </configSections>
    <connectionStrings>
                        <add   name="ADO.Properties.Settings.Test7ConnectionString"
connectionString="Data                          Source=(localdb)\MSSQLLocalDB;Initial
Catalog=Test7;Integrated Security=True;Pooling=False"
            providerName="System.Data.SqlClient" />
    </connectionStrings>
</configuration>
```

MainWindow.xaml

```
<Window x:Class="ADO.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:data="clr-namespace:System.Data;assembly=System.Data"
        mc:Ignorable="d"
        Title="MainWindow" Loaded="Window_Loaded" Width="600" MaxWidth="600"
MinHeight="300" MinWidth="600">
    <Window.Resources>
    </Window.Resources>
    <Grid HorizontalAlignment="Left">
        <Grid.RowDefinitions>
            <RowDefinition Height="5*"></RowDefinition>
            <RowDefinition></RowDefinition>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto"></ColumnDefinition>
            <ColumnDefinition Width="Auto"></ColumnDefinition>
            <ColumnDefinition Width="Auto"></ColumnDefinition>
        </Grid.ColumnDefinitions>
        <DataGrid x:Name="peopleDataGrid" Grid.ColumnSpan="3"
AutoGenerateColumns="False" EnableRowVirtualization="True" ItemsSource="{Binding}"
Margin="10" HorizontalAlignment="Center" IsReadOnly="True" >
            <DataGrid.Columns>
                <DataGridTextColumn x:Name="idColumn" Binding="{Binding Id}"
Header="Id" IsReadOnly="True" Width="Auto"/>
                <DataGridTextColumn x:Name="fIOColumn" Binding="{Binding FIO}"
Header="ФИО" Width="Auto"/>
                <DataGridTextColumn x:Name="birthdayColumn" Binding="{Binding
Birthday}" Header="День рождения" Width="Auto"/>
                <DataGridTextColumn x:Name="emailColumn" Binding="{Binding Email}"
Header="Email" Width="Auto"/>
                <DataGridTextColumn x:Name="phoneColumn" Binding="{Binding Phone}"
Header="Телефон" Width="Auto"/>
            </DataGrid.Columns>
        </DataGrid>

        <Button x:Name="addButton" Content="Добавить" Click="addButton_Click"
Grid.Column="0" Grid.Row="1" Width="70" Height="30" HorizontalAlignment="Center"
Margin="10" VerticalAlignment="Center"/>
        <Button x:Name="updateButton" Content="Изменить"
Click="updateButton_Click" Grid.Column="1" Grid.Row="1" Width="70" Height="30"
HorizontalAlignment="Center" Margin="10" VerticalAlignment="Center"/>
        <Button x:Name="deleteButton" Content="Удалить" Click="deleteButton_Click"
Grid.Column="2" Grid.Row="1" Width="70" Height="30" Margin="10"
HorizontalAlignment="Center" VerticalAlignment="Center"/>
    </Grid>
</Window>
```

MainWindow.xaml.cs

```
using System.Data;
using System.Data.SqlClient;
using System.Windows;
namespace ADO
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        SqlConnection connection;
        SqlDataAdapter adapter;
        DataTable dt;
        public MainWindow()
        {
            InitializeComponent();
        }
        private void Window_Loaded(object sender, RoutedEventArgs e)
        {
                                                    string    connectionString    =
Properties.Settings.Default.Test7ConnectionString;
            connection = new SqlConnection(connectionString);
            adapter = new SqlDataAdapter();
              SqlCommand command = new SqlCommand("SELECT ID, FIO, Birthday, Email,
Phone FROM People", connection);
            adapter.SelectCommand = command;
            //insert
              command = new SqlCommand(@"INSERT INTO People (FIO, Birthday, Email,
Phone)  VALUES  (@FIO,  @Birthday,  @Email,  @Phone);  SET  @ID  =  @@IDENTITY;",
connection);
            command.Parameters.Add("@FIO", SqlDbType.NVarChar, -1, "FIO");
                      command.Parameters.Add("@Birthday",  SqlDbType.NVarChar,  -1,
"Birthday");
            command.Parameters.Add("@Email", SqlDbType.NVarChar, 100, "Email");
            command.Parameters.Add("@Phone", SqlDbType.NVarChar, -1, "Phone");
              SqlParameter param = command.Parameters.Add("@ID", SqlDbType.Int, 0,
"ID");
            param.Direction = ParameterDirection.Output;
            adapter.InsertCommand = command;
            // update
               command = new SqlCommand(@"UPDATE People SET FIO = @FIO, Birthday =
@Birthday, Email = @Email, Phone = @Phone WHERE ID = @ID", connection);
            command.Parameters.Add("@FIO", SqlDbType.NVarChar, -1, "FIO");
                      command.Parameters.Add("@Birthday",  SqlDbType.NVarChar,  -1,
"Birthday");
            command.Parameters.Add("@Email", SqlDbType.NVarChar, 100, "Email");
            command.Parameters.Add("@Phone", SqlDbType.NVarChar, -1, "Phone");
            param = command.Parameters.Add("@ID", SqlDbType.Int, 0, "ID");
            param.SourceVersion = DataRowVersion.Original;
            adapter.UpdateCommand = command;
            //delete
                  command  =  new  SqlCommand("DELETE  FROM  People  WHERE  ID  =  @ID",
connection);
            param = command.Parameters.Add("@ID", SqlDbType.Int, 0, "ID");
            param.SourceVersion = DataRowVersion.Original;
            adapter.DeleteCommand = command;
            dt = new DataTable();
```

```
            adapter.Fill(dt);
            peopleDataGrid.DataContext = dt.DefaultView;
        }
        private void addButton_Click(object sender, RoutedEventArgs e)
        {
            // Добавим новую строку
            DataRow newRow = dt.NewRow();
            EditWindow editWindow = new EditWindow(newRow);
            editWindow.ShowDialog();
            if (editWindow.DialogResult.HasValue && editWindow.DialogResult.Value)
            {
                dt.Rows.Add(editWindow.resultRow);
                adapter.Update(dt);
            }
        }
       private void updateButton_Click(object sender, RoutedEventArgs e)
        {
            DataRowView newRow = (DataRowView)peopleDataGrid.SelectedItem;
            newRow.BeginEdit();
            EditWindow editWindow = new EditWindow(newRow.Row);
            editWindow.ShowDialog();
            if (editWindow.DialogResult.HasValue && editWindow.DialogResult.Value)
            {
                newRow.EndEdit();
                adapter.Update(dt);
            }
            else
            {
                newRow.CancelEdit();
            }
        }
        private void deleteButton_Click(object sender, RoutedEventArgs e)
        {
            DataRowView newRow = (DataRowView)peopleDataGrid.SelectedItem;

            newRow.Row.Delete();
            adapter.Update(dt);
        }
    }
}
```

EditWindow.xaml

```
<Window x:Class="ADO.EditWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:ADO"
        mc:Ignorable="d"
        Title="EditWindow" Height="300" Width="300" Loaded="Window_Loaded">
    <Window.Resources>
    </Window.Resources>
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*"/>
            <ColumnDefinition Width="Auto"/>
        </Grid.ColumnDefinitions>
```

```
        <Grid.RowDefinitions>
            <RowDefinition Height="4*"/>
            <RowDefinition Height="Auto"/>
        </Grid.RowDefinitions>
        <Grid x:Name="editGrid" Grid.ColumnSpan="2">
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="Auto"/>
                <ColumnDefinition Width="Auto"/>
            </Grid.ColumnDefinitions>
            <Grid.RowDefinitions>
                <RowDefinition Height="Auto"/>
                <RowDefinition Height="Auto"/>
                <RowDefinition Height="Auto"/>
                <RowDefinition Height="Auto"/>
                <RowDefinition Height="Auto"/>
            </Grid.RowDefinitions>
            <Label Content="ФИО:" Grid.Column="0" HorizontalAlignment="Left"
Margin="3" Grid.Row="1" VerticalAlignment="Center"/>
            <TextBox x:Name="fIOTextBox" Grid.Column="1"
HorizontalAlignment="Left" Height="23" Margin="3" Grid.Row="1"
VerticalAlignment="Center" Width="120"/>
            <Label Content="День рождения:" Grid.Column="0"
HorizontalAlignment="Left" Margin="3" Grid.Row="2" VerticalAlignment="Center"/>
            <TextBox x:Name="birthdayTextBox" Grid.Column="1"
HorizontalAlignment="Left" Height="23" Margin="3" Grid.Row="2"
VerticalAlignment="Center" Width="120"/>
            <Label Content="Email:" Grid.Column="0" HorizontalAlignment="Left"
Margin="3" Grid.Row="3" VerticalAlignment="Center"/>
            <TextBox x:Name="emailTextBox" Grid.Column="1"
HorizontalAlignment="Left" Height="23" Margin="3" Grid.Row="3"
VerticalAlignment="Center" Width="120"/>
            <Label Content="Телефон:" Grid.Column="0" HorizontalAlignment="Left"
Margin="3" Grid.Row="4" VerticalAlignment="Center"/>
            <TextBox x:Name="phoneTextBox" Grid.Column="1"
HorizontalAlignment="Left" Height="23" Margin="3" Grid.Row="4"
VerticalAlignment="Center" Width="120"/>
        </Grid>
        <Button x:Name="saveButton" IsDefault="True" Content="Сохранить"
Margin="10" Grid.Row="1" HorizontalAlignment="Center"  VerticalAlignment="Bottom"
Width="75" Click="saveButton_Click"/>
        <Button x:Name="cancelButton" IsCancel="True" Content="Отменить"
Margin="10" Grid.Row="1" Grid.Column="1" HorizontalAlignment="Center"
VerticalAlignment="Bottom" Width="75" Click="cancelButton_Click"/>
    </Grid>
</Window>
```

EditWindow.xaml.cs

```
using System.Data;
using System.Windows;
namespace ADO
{
    /// <summary>
    /// Interaction logic for EditWindow.xaml
    /// </summary>
    public partial class EditWindow : Window
    {
        public DataRow resultRow { get; set; }
        public EditWindow(DataRow dataRow)
        {
            InitializeComponent();
            resultRow = dataRow;
        }
        private void Window_Loaded(object sender, RoutedEventArgs e)
        {
            fIOTextBox.Text = resultRow["FIO"].ToString();
            birthdayTextBox.Text = resultRow["Birthday"].ToString();
            emailTextBox.Text = resultRow["Email"].ToString();
            phoneTextBox.Text = resultRow["Phone"].ToString();
        }
        private void saveButton_Click(object sender, RoutedEventArgs e)
        {
            resultRow["FIO"] = fIOTextBox.Text;
            resultRow["Birthday"] = birthdayTextBox.Text;
            resultRow["Email"] = emailTextBox.Text;
            resultRow["Phone"] = phoneTextBox.Text;
            DialogResult = true;
        }
        private void cancelButton_Click(object sender, RoutedEventArgs e)
        {
            DialogResult = false;
        }
    }
}
```

Практическое задание

Изменить WPF-приложение для ведения списка сотрудников компании (из урока 5), **используя связывание данных, DataGrid и ADO.NET**.

1. Создать таблицы **Employee** и **Department** в БД **MSSQL Server** и заполнить списки сущностей начальными данными.

2. Для списка сотрудников и списка департаментов предусмотреть визуализацию (отображение). Это можно сделать, например, с использованием **ComboBox** или **ListView**.

3. Предусмотреть редактирование сотрудников и департаментов. Должна быть возможность изменить департамент у сотрудника. Список департаментов для выбора можно выводить в **ComboBox**, и все это можно выводить на дополнительной форме.

4. Предусмотреть возможность создания новых сотрудников и департаментов. Реализовать данную возможность либо на форме редактирования, либо сделать новую форму.

Дополнительные материалы

1. Anne Boehm, Ged Mead. Murach's ADO.NET 4 Database Programming with C# 2010 (Murach: Training & Reference) 4th Edition.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Anne Boehm, Ged Mead. Murach's ADO.NET 4 Database Programming with C# 2010 (Murach: Training & Reference) 4th Edition.
2. MSDN.