**Работа с SqlDataAdapter и DataSet**

**SqlDataAdapter и DataSet**

Ранее для получения данных мы использовали объект SqlDataReader, с помощью которого построчно можно перебрать ответ от сервера базы данных. Но есть и другой способ, который демонстрирует использование объектов SqlDataAdapter и DataSet. DataSet представляет хранилище данных, с которыми можно работать независимо от наличия подключения, а SqlDataAdapter заполняет DataSet данными из БД.

Для получения данных через объект SqlDataAdapter необходимо организовать подключение к БД и выполнить команду SELECT. Есть несколько способов создания SqlDataAdapter:

|  |  |
| --- | --- |
| 1  2  3  4 | SqlDataAdapter adapter = new SqlDataAdapter();  SqlDataAdapter adapter = new SqlDataAdapter(command);  SqlDataAdapter adapter = new SqlDataAdapter(sql, connection);  SqlDataAdapter adapter = new SqlDataAdapter(sql, connectionString); |

* Можно использовать конструктор без параметров, а команду SELECT и подключение установить позже
* Можно передать в конструктор объект SqlCommand
* Можно в конструкторе установить sql-выражение SELECT и объект SqlConnection
* Можно в конструкторе установить sql-выражение SELECT и строку подключения

Рассмотрим, как получить данные в DataSet через SqlDataAdapter. Для работы с DataSet особенно удобно использовать элементы управления, которые могут заполняться из внешнего источника данных, например, DataGridView в Windows Forms. Поэтому создадим новый проект по типу Windows Forms Application.

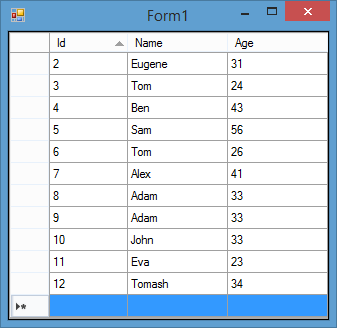
Добавим на единственную форму в проекте элемент DataGridView и определим следующий код формы:

|  |  |
| --- | --- |
| 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 | using System.Data;  using System.Windows.Forms;  using System.Data.SqlClient;    namespace AdoNetWinFormsApp  {      public partial class Form1 : Form      {          public Form1()          {              InitializeComponent();                string connectionString = @"Data Source=.\SQLEXPRESS;Initial Catalog=usersdb;Integrated Security=True";              string sql = "SELECT \* FROM Users";              using (SqlConnection connection = new SqlConnection(connectionString))              {                  connection.Open();                  // Создаем объект DataAdapter                  SqlDataAdapter adapter = new SqlDataAdapter(sql, connection);                  // Создаем объект Dataset                  DataSet ds = new DataSet();                  // Заполняем Dataset                  adapter.Fill(ds);                  // Отображаем данные                  dataGridView1.DataSource = ds.Tables[0];              }          }      }  } |

В конструкторе формы в DataGridView загружаются данные. Для загрузки данных создается объект SqlDataAdapter, который принимает объект подключения и sql-выражение SELECT. Затем создается объект DataSet и с помощью метода adapter.Fill() в него загружаются данные. Дальше происходит установка источника данных для DataGridView:

|  |  |
| --- | --- |
| 1 | dataGridView1.DataSource = ds.Tables[0]; |

В качестве источника устанавливается одна из таблиц в DataSet. Каждая таблица представляет объект DataTable, и в DataSet может быть определено несколько таких таблиц. Но в данном случае при выборке в DataSet есть только одна таблица, которую мы можем получить из коллекции Tables по индексу.



## Постраничный просмотр в SqlDataAdapter

В прошлой теме была рассмотрена загрузка данных через SqlDataAdapter в DataSet с последующим отображением в DataGridView. Однако если в базе данных очень много строк, то для более комфортной работы может потребоваться разбить эти данные на отдельные куски или страницы. Используя DataGridView очень легко сделать постраничный просмотр объект. Для этого определим на форме DataGridView и две кнопки - backButton и nextButton соответственно для перехода назад и вперед.

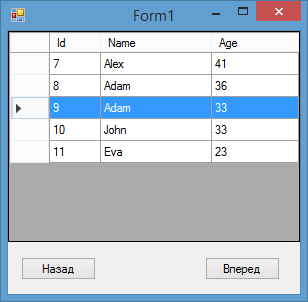
Затем определим следующий код формы:

|  |  |
| --- | --- |
| 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 | using System;  using System.Data;  using System.Data.SqlClient;  using System.Windows.Forms;    namespace PagingApp  {      public partial class Form1 : Form      {          int pageSize = 5; // размер страницы          int pageNumber = 0; // текущая страница          string connectionString = @"Data Source=.\SQLEXPRESS;Initial Catalog=usersdb;Integrated Security=True";          SqlDataAdapter adapter;          DataSet ds;          public Form1()          {              InitializeComponent();                dataGridView1.SelectionMode = DataGridViewSelectionMode.FullRowSelect;              dataGridView1.AllowUserToAddRows = false;                using (SqlConnection connection = new SqlConnection(connectionString))              {                  adapter = new SqlDataAdapter(GetSql(), connection);                    ds = new DataSet();                  adapter.Fill(ds, "Users");                  dataGridView1.DataSource = ds.Tables[0];                  dataGridView1.Columns["Id"].ReadOnly = true;              }          }          // обработчик кнопки Вперед          private void nextButton\_Click(object sender, EventArgs e)          {              if (ds.Tables["Users"].Rows.Count < pageSize) return;                pageNumber++;              using (SqlConnection connection = new SqlConnection(connectionString))              {                  adapter = new SqlDataAdapter(GetSql(), connection);                    ds.Tables["Users"].Rows.Clear();                    adapter.Fill(ds,"Users");              }          }          // обработчик кнопки Назад          private void backButton\_Click(object sender, EventArgs e)          {              if (pageNumber==0) return;              pageNumber--;                using (SqlConnection connection = new SqlConnection(connectionString))              {                  adapter = new SqlDataAdapter(GetSql(), connection);                    ds.Tables["Users"].Rows.Clear();                    adapter.Fill(ds, "Users");              }          }            private string GetSql()          {              return "SELECT \* FROM Users ORDER BY Id OFFSET ((" + pageNumber + ") \* " + pageSize + ") " +                  "ROWS FETCH NEXT " + pageSize + "ROWS ONLY";          }      }  } |

Переменная pageSize определяет количество строк на одной странице, а переменная pageNumber будет хранить номер текущей просматриваемой страницы.

Запрос к базе данных будет создаваться с помощью функции GetSql().

В обработчике кнопки Вперед увеличивается текущая страница на единицу и производится запрос. В обработчике кнопки Назад уменьшается счетчик текущей страницы.

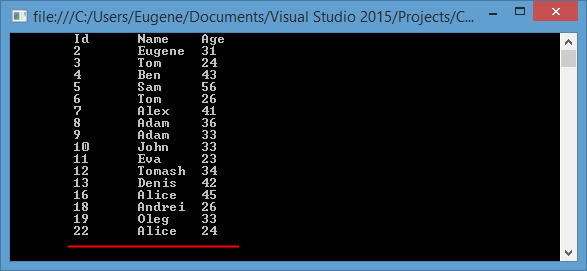


## SqlCommandBuilder и сохранение изменений DataSet в базе данных

Получив данные в DataSet, мы можем производить с ними различными операции: удалять, изменять, добавлять новые записи. Однако все делаемые нами изменения автоматически не будут сохраняться в БД. Для этого нам еще надо вызвать метод **Update** объекта SqlDataAdapter, который заполнял DataSet.

Для модификации данных в БД в соответствии с изменениями в DataSet SqlDataAdapter использует команды InsertCommand, UpdateCommand иDeleteCommand. Мы можем сами определить для этих команд sql-выражения, либо мы можем воспользоваться классом **SqlCommandBuilder**, который позволяет автоматически сгенерировать нужные выражения. Используем SqlCommandBuilder:

|  |  |
| --- | --- |
| 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 | static string connectionString = @"Data Source=.\SQLEXPRESS;Initial Catalog=usersdb;Integrated Security=True";  static void Main(string[] args)  {      string sql = "SELECT \* FROM Users";      using (SqlConnection connection = new SqlConnection(connectionString))      {          connection.Open();          SqlDataAdapter adapter = new SqlDataAdapter(sql, connection);          DataSet ds = new DataSet();          adapter.Fill(ds);            DataTable dt = ds.Tables[0];          // добавим новую строку          DataRow newRow = dt.NewRow();          newRow["Name"] = "Alice";          newRow["Age"] = 24;          dt.Rows.Add(newRow);            // создаем объект SqlCommandBuilder          SqlCommandBuilder commandBuilder = new SqlCommandBuilder(adapter);          adapter.Update(ds);          // альтернативный способ - обновление только одной таблицы          //adapter.Update(dt);          // заново получаем данные из бд          // очищаем полностью DataSet          ds.Clear();          // перезагружаем данные          adapter.Fill(ds);            foreach (DataColumn column in dt.Columns)              Console.Write("\t{0}", column.ColumnName);          Console.WriteLine();          // перебор всех строк таблицы          foreach (DataRow row in dt.Rows)          {              // получаем все ячейки строки              var cells = row.ItemArray;              foreach (object cell in cells)                  Console.Write("\t{0}", cell);              Console.WriteLine();          }      }      Console.Read();  } |



Здесь после загрузки данных создается новая строка, которая затем добавляется в DataTable. При вызове у адаптера метода Update()происходит анализ изменений, которые произошли. И после этого выполняется соответствующая команда. В данном случае так как идет добавление новой строки, то будет выполняться команда InsertCommand. Однако в данном коде мы нигде явным образом не задаем эту команду, за нас все автоматически делает SqlCommandBuilder. Для применения этого класса достаточно вызвать его конструктор, в который передается нужный адаптер:

|  |  |
| --- | --- |
| 1 | SqlCommandBuilder commandBuilder = new SqlCommandBuilder(adapter); |

Причем больше нигде в коде вы этот объект не вызываем.

При необходимости мы можем получить sql-выражения используемых команд:

|  |  |
| --- | --- |
| 1  2  3 | Console.WriteLine(commandBuilder.GetUpdateCommand().CommandText);  Console.WriteLine(commandBuilder.GetInsertCommand().CommandText);  Console.WriteLine(commandBuilder.GetDeleteCommand().CommandText); |

В моем случае команда обновления будет выглядеть так:

|  |  |
| --- | --- |
| 1 | UPDATE [Users] SET [Name]=@p1, [Age]=@p2 WHERE (([Id]=@p3) AND ([Name]=@p4) AND ([Age]=@p5)) |

Команда вставки:

|  |  |
| --- | --- |
| 1 | INSERT INTO [Users] ([Name],[Age]) VALUES (@p1, @p2) |

Команда удаления:

|  |  |
| --- | --- |
| 1 | DELETE FROM [Users] WHERE (([Id]=@p1) AND ([Name]=@p2) AND ([Age]=@p3)) |

## Обновление БД из DataSet вручную

Хотя в предыдущей теме объект SqlCommandBuilder позволял нам автоматически создать все нужные выражения для обновления данных в БД из DataSet, но все же этот способ имеет свои недостатки. Например, взглянем на следующий кусочек кода, который использовался в прошлой теме:

|  |  |
| --- | --- |
| 1  2  3  4 | SqlCommandBuilder commandBuilder = new SqlCommandBuilder(adapter);  adapter.Update(ds);  ds.Clear();  adapter.Fill(ds); |

После обновления происходит очистка DataSet и перезагрузка данных, что снижает производительность приложения. Хотя в DataTable у нас уже добавлена строка с новыми данными, и в ней не хватает только id - значения, которое генерируется самой базой данных при добавлении. Без id нам трудно будет управлять данными, мы не сможем их автоматически через тот же SqlCommandBuilder обновлять или удалять. Поэтому в идеале хотелось бы избежать и перезагрузки данных и в то же время получить id новой записи при выполнении метода adapter.Update. И более гибкий способ состоит в том, что мы сами вручную определяем все те выражения, которые будут выполняться.

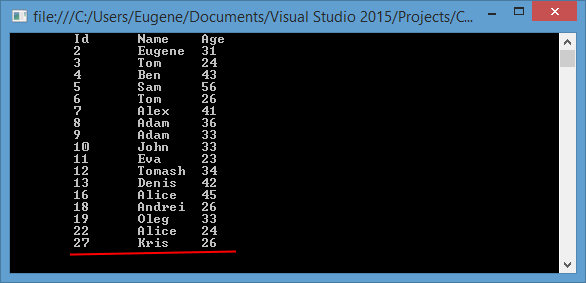
Итак, добавим в базу данных следующую хранимую процедуру:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | CREATE PROCEDURE [dbo].[sp\_CreateUser]      @name nvarchar(50),      @age int,      @Id int out  AS      INSERT INTO Users (Name, Age)      VALUES (@name, @age)        SET @Id=SCOPE\_IDENTITY()  GO |

В качестве входных параметров она принимает имя и возраст пользователя и возвращает его id.

Теперь изменим код из прошлой темы:

|  |  |
| --- | --- |
| 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 | static string connectionString = @"Data Source=.\SQLEXPRESS;Initial Catalog=usersdb;Integrated Security=True";  static void Main(string[] args)  {      string sql = "SELECT \* FROM Users";      using (SqlConnection connection = new SqlConnection(connectionString))      {          connection.Open();          SqlDataAdapter adapter = new SqlDataAdapter(sql, connection);          SqlCommandBuilder commandBuilder = new SqlCommandBuilder(adapter);            // устанавливаем команду на вставку          adapter.InsertCommand = new SqlCommand("sp\_CreateUser", connection);          // это будет зранимая процедура          adapter.InsertCommand.CommandType = CommandType.StoredProcedure;          // добавляем параметр для name          adapter.InsertCommand.Parameters.Add(new SqlParameter("@name", SqlDbType.NVarChar, 50, "Name"));          // добавляем параметр для age          adapter.InsertCommand.Parameters.Add(new SqlParameter("@age", SqlDbType.Int, 0, "Age"));          // добавляем выходной параметр для id          SqlParameter parameter = adapter.InsertCommand.Parameters.Add("@Id", SqlDbType.Int, 0, "Id");          parameter.Direction = ParameterDirection.Output;            DataSet ds = new DataSet();          adapter.Fill(ds);            DataTable dt = ds.Tables[0];          // добавим новую строку          DataRow newRow = dt.NewRow();          newRow["Name"] = "Kris";          newRow["Age"] = 26;          dt.Rows.Add(newRow);            adapter.Update(ds);          ds.AcceptChanges();          foreach (DataColumn column in dt.Columns)              Console.Write("\t{0}", column.ColumnName);          Console.WriteLine();          // перебор всех строк таблицы          foreach (DataRow row in dt.Rows)          {              // получаем все ячейки строки              var cells = row.ItemArray;              foreach (object cell in cells)                  Console.Write("\t{0}", cell);              Console.WriteLine();          }      }      Console.Read();  } |



Здесь также используется SqlCommandBuilder, но теперь из-за переустановки свойства adapter.InsertCommand выполнение добавления нового объекта будет переопределено.

При определении выходного параметра мы указываем, что он будет называться "@Id" и будет возвращать значение для столбца Id в DataTable:

|  |  |
| --- | --- |
| 1 | adapter.InsertCommand.Parameters.Add("@Id", SqlDbType.Int, 0, "Id"); |

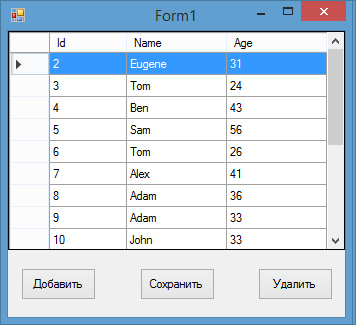
В итоге при выполнении метода adapter.Update() id автоматически попадет в DataTable. При этом операции по обновлению и удалению мы можем использовать те же, что предоставляет SqlCommandBuilder.

И после обновления базы данных с помощью метода AcceptChanges() объекта DataSet производится принятие всех изменений в DataSet ко всем измененным строкам.

## Все операции с БД в графическом приложении

Ранее мы рассмотрели, как удобно загружать данные в приложении Windows Forms в элемент DataGridView через DataSet. Теперь определим полнофункциональную форму, через которую мы сможем производить все стандартные CRUD операции в базе данных.

Итак, определим форму, на которой будет элемент DataGridView и три кнопки для добавления, удаления и сохранения изменений. Форма в итоге будет выглядеть примерно следующим образом:



Код формы будет выглядеть 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  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 | using System;  using System.Data;  using System.Windows.Forms;  using System.Data.SqlClient;    namespace AdoNetWinFormsApp  {      public partial class Form1 : Form      {          DataSet ds;          SqlDataAdapter adapter;          SqlCommandBuilder commandBuilder;          string connectionString = @"Data Source=.\SQLEXPRESS;Initial Catalog=usersdb;Integrated Security=True";          string sql = "SELECT \* FROM Users";            public Form1()          {              InitializeComponent();                dataGridView1.SelectionMode = DataGridViewSelectionMode.FullRowSelect;              dataGridView1.AllowUserToAddRows = false;                using (SqlConnection connection = new SqlConnection(connectionString))              {                  connection.Open();                  adapter = new SqlDataAdapter(sql, connection);                    ds = new DataSet();                  adapter.Fill(ds);                  dataGridView1.DataSource = ds.Tables[0];                  // делаем недоступным столбец id для изменения                  dataGridView1.Columns["Id"].ReadOnly = true;              }            }          // кнопка добавления          private void addButton\_Click(object sender, EventArgs e)          {              DataRow row = ds.Tables[0].NewRow(); // добавляем новую строку в DataTable              ds.Tables[0].Rows.Add(row);          }          // кнопка удаления          private void deleteButton\_Click(object sender, EventArgs e)          {              // удаляем выделенные строки из dataGridView1              foreach(DataGridViewRow row in dataGridView1.SelectedRows)              {                  dataGridView1.Rows.Remove(row);              }          }          // кнопка сохранения          private void saveButton\_Click(object sender, EventArgs e)          {              using (SqlConnection connection = new SqlConnection(connectionString))              {                  connection.Open();                  adapter = new SqlDataAdapter(sql, connection);                  commandBuilder = new SqlCommandBuilder(adapter);                  adapter.InsertCommand = new SqlCommand("sp\_CreateUser", connection);                  adapter.InsertCommand.CommandType = CommandType.StoredProcedure;                  adapter.InsertCommand.Parameters.Add(new SqlParameter("@name", SqlDbType.NVarChar, 50, "Name"));                  adapter.InsertCommand.Parameters.Add(new SqlParameter("@age", SqlDbType.Int, 0, "Age"));                    SqlParameter parameter = adapter.InsertCommand.Parameters.Add("@Id", SqlDbType.Int, 0, "Id");                  parameter.Direction = ParameterDirection.Output;                    adapter.Update(ds);              }          }      }  } |

Здесь для добавления объекта мы будем обращаться к хранимой процедуре sp\_CreateUser, которая была добавлена в базу данных в прошлой теме.

В конструкторе данные загружаются в DataSet, первая таблица которого устанавливается в качестве источника данных для dataGridView1:

|  |  |
| --- | --- |
| 1 | dataGridView1.DataSource = ds.Tables[0]; |

Также в конструкторе устанавливается полное выделение строки и запрет на ручное добавление новых строк:

|  |  |
| --- | --- |
| 1  2 | dataGridView1.SelectionMode = DataGridViewSelectionMode.FullRowSelect;  dataGridView1.AllowUserToAddRows = false; |

В обработчике кнопки добавления создается новая строка, которая добавляется в таблицу объекта DataSet. И так как мы ранее установили привязку к источнику данных, то автоматически новая строка также будет добавляться и в dataGridView1:

|  |  |
| --- | --- |
| 1  2  3  4  5 | private void addButton\_Click(object sender, EventArgs e)  {      DataRow row = ds.Tables[0].NewRow(); // добавляем новую строку в DataTable      ds.Tables[0].Rows.Add(row);  } |

В обработчике кнопки удаления удаляются выделенные строки в dataGridView1. Опять же в силу привязки к источнику данных будет также происходить удаление и из таблицы в DataSet:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | private void deleteButton\_Click(object sender, EventArgs e)  {      foreach(DataGridViewRow row in dataGridView1.SelectedRows)      {          dataGridView1.Rows.Remove(row);      }  } |

Для обновления на не нужна никакая кнопка, так как мы можем нажать на любую ячейку таблицы (кроме заблокированного для изменения столбца Id) и изменить в ней данные. Однако сами по себе добавление новой строки, удаление строк, изменение ячеек ни как автоматически не отразятся на базе данных. И чтобы бд синхронизировалась, пользователю надо будет нажать на кнопку сохранения, обработчик которой выглядит следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | private void saveButton\_Click(object sender, EventArgs e)  {      using (SqlConnection connection = new SqlConnection(connectionString))      {          connection.Open();          adapter = new SqlDataAdapter(sql, connection);          commandBuilder = new SqlCommandBuilder(adapter);          adapter.InsertCommand = new SqlCommand("sp\_CreateUser", connection);          adapter.InsertCommand.CommandType = CommandType.StoredProcedure;          adapter.InsertCommand.Parameters.Add(new SqlParameter("@name", SqlDbType.NVarChar, 50, "Name"));          adapter.InsertCommand.Parameters.Add(new SqlParameter("@age", SqlDbType.Int, 0, "Age"));            SqlParameter parameter = adapter.InsertCommand.Parameters.Add("@Id", SqlDbType.Int, 0, "Id");          parameter.Direction = ParameterDirection.Output;            adapter.Update(ds);      }  } |

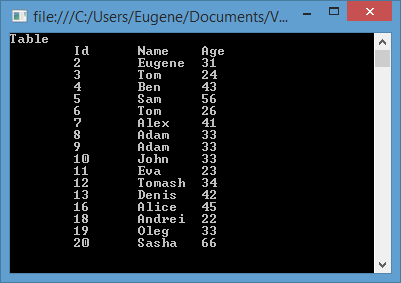
Как в прошлой теме здесь устанавливается у адаптера команда на добавление InsertCommand и затем вызывается метод Update(). В итоге мы можем добавить несколько строк, удалить, изменить, и потом один раз мы нажмем на кнопку, и все изменения будут применены к базе данных.

## DataSet и DataTable

После получения данных из базы данных через SqlDataAdapter в DataSet мы можем локально работать с этими данными вне зависимости от наличия подключения. Более того если нам даже и не надо использовать никакую базу данных, но при этом мы хотим иметь удобный функционал для работы с данными в виде наборов таблиц, то мы также можем воспользоваться классом DataSet.

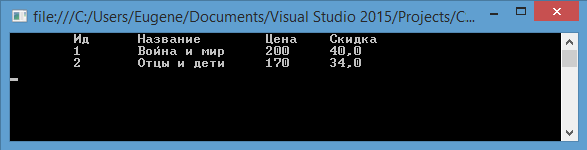
Объект DataSet содержит таблицы, которые представлены типом **DataTable**. Таблица, в свою очередь, состоит из столбцов и строк. Каждый столбец представляет объект **DataColumn**, а строка - объект **DataRow**. Все данные строки хранятся в свойстве **ItemArray**, который представляет массив объектов - значений отдельных ячеек строки. Например, получим все таблицы и выведем их содержимое:

|  |  |
| --- | --- |
| 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 sql = "SELECT \* FROM Users";      string connectionString = @"Data Source=.\SQLEXPRESS;Initial Catalog=usersdb;Integrated Security=True";      using (SqlConnection connection = new SqlConnection(connectionString))      {          connection.Open();          SqlDataAdapter adapter = new SqlDataAdapter(sql, connection);            DataSet ds = new DataSet();          adapter.Fill(ds);          // перебор всех таблиц          foreach (DataTable dt in ds.Tables)          {              Console.WriteLine(dt.TableName); // название таблицы              // перебор всех столбцов              foreach(DataColumn column in dt.Columns)                  Console.Write("\t{0}", column.ColumnName);              Console.WriteLine();              // перебор всех строк таблицы              foreach (DataRow row in dt.Rows)              {                  // получаем все ячейки строки                  var cells = row.ItemArray;                  foreach (object cell in cells)                      Console.Write("\t{0}", cell);                  Console.WriteLine();              }          }      }  } |



Теперь рассмотрим, как мы можем работать с объектами DataSet и DataTable без какой-либо базы данных. Например, создадим вручную в DataSet несколько таблиц и определим их структуру:

|  |  |
| --- | --- |
| 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 | static void Main(string[] args)  {      DataSet bookStore = new DataSet("BookStore");      DataTable booksTable = new DataTable("Books");      // добавляем таблицу в dataset      bookStore.Tables.Add(booksTable);        // создаем столбцы для таблицы Books      DataColumn idColumn = new DataColumn("Id", Type.GetType("System.Int32"));      idColumn.Unique = true; // столбец будет иметь уникальное значение      idColumn.AllowDBNull = false; // не может принимать null      idColumn.AutoIncrement = true; // будет автоинкрементироваться      idColumn.AutoIncrementSeed = 1; // начальное значение      idColumn.AutoIncrementStep = 1; // приращении при добавлении новой строки        DataColumn nameColumn = new DataColumn("Name", Type.GetType("System.String"));      DataColumn priceColumn = new DataColumn("Price", Type.GetType("System.Decimal"));      priceColumn.DefaultValue = 100; // значение по умолчанию      DataColumn discountColumn = new DataColumn("Discount", Type.GetType("System.Decimal"));      discountColumn.Expression = "Price \* 0.2";        booksTable.Columns.Add(idColumn);      booksTable.Columns.Add(nameColumn);      booksTable.Columns.Add(priceColumn);      booksTable.Columns.Add(discountColumn);      // определяем первичный ключ таблицы books      booksTable.PrimaryKey = new DataColumn[] { booksTable.Columns["Id"] };        DataRow row = booksTable.NewRow();      row.ItemArray = new object[] { null, "Война и мир", 200 };      booksTable.Rows.Add(row); // добавляем первую строку      booksTable.Rows.Add(new object[] { null, "Отцы и дети", 170 }); // добавляем вторую строку        Console.Write("\tИд \tНазвание \tЦена \tСкидка");      Console.WriteLine();      foreach (DataRow r in booksTable.Rows)      {          foreach (var cell in r.ItemArray)              Console.Write("\t{0}", cell);          Console.WriteLine();      }      Console.Read();  } |



Разберем весь код. Сначала создаются объекты DataSet и DataTable, в конструктор которых передается название. Затем создается четыре столбца. Каждый столбец в конструкторе принимает два параметра: имя столбца и его тип.

|  |  |
| --- | --- |
| 1 | DataColumn idColumn = new DataColumn("Id", Type.GetType("System.Int32")); |

Причем для столбца Id устанавливается, что значения этого столбца должны иметь уникальное значение, не должны принимать null, и их значение при добавлении нового объекта будет инкрементироваться на единицу. То есть фактически это стандартный столбец Id, как в большинстве баз данных.

Далее создается еще три столбца, при этом для столбца Discount устанавливается свойство Expression, указывающее на выражение, которое будет использоваться для вычисления значения столбца:

|  |  |
| --- | --- |
| 1 | discountColumn.Expression = "Price \* 0.2"; |

То есть в данном случае значение столбца Discount равно значению столбца Price, помноженного на 0.2.

Затем устанавливается первичный ключ для таблицы с помощью свойства PrimaryKey:

|  |  |
| --- | --- |
| 1 | booksTable.PrimaryKey = new DataColumn[] { booksTable.Columns["Id"] }; |

В роли первичного ключа выступает столбец Id. Но мы также можем использовать набор различных столбцов для создания составного ключа.

После определения схемы таблицы в нее добавляются две строки:

|  |  |
| --- | --- |
| 1  2  3  4 | DataRow row = booksTable.NewRow();  row.ItemArray = new object[] { null, "Война и мир", 200 };  booksTable.Rows.Add(row); // добавляем первую строку  booksTable.Rows.Add(new object[] {null, "Отцы и дети", 170 }); |

Значения в метод booksTable.Rows.Add можно передать как напрямую в виде массива объектов, так и в виде объекта DataRow. При этом нам надо передать ровно столько значений, сколько в таблице столбцов. Однако поскольку первый столбец Id устанавливается через автоинкремент, мы можем передать значение null - оно все равно будет игнорироваться. Также мы можем опустить последний параметр для столбца Discount, так как его значение вычисляется с помощью выражения "Price \* 0.2". Более того мы даже можем опустить значение для третьего столбца Price, так как у него установлено свойство DefaultValue, которое устанавливает значение по умолчанию, если значение отсутствует:

|  |  |
| --- | --- |
| 1 | booksTable.Rows.Add(new object[] {null, "Отцы и дети"}); |

И в конце идет перебор строк таблицы.

Кроме добавления мы можем производить и другие операции со строками. Например, мы можем получить строку по индексу:

|  |  |
| --- | --- |
| 1 | DataRow row = booksTable.Rows[0]; // первая строка |

Получив строку по индексу, можно изменить ее ячейки:

|  |  |
| --- | --- |
| 1 | booksTable.Rows[0][2] = 300; //третьей ячейке первой строки присваивается значение 300 |

И также можно удалять строку:

|  |  |
| --- | --- |
| 1  2  3  4 | booksTable.Rows.RemoveAt(1); // удаление второй строки по индексу  // другой сопосб удаления  DataRow row = booksTable.Rows[0];  booksTable.Rows.Remove(row); |

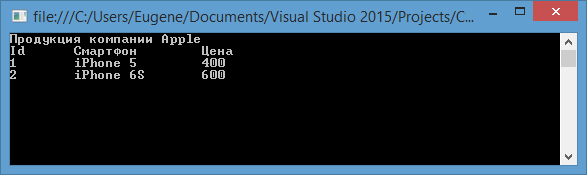
Используя метод Select() объекта DataTable мы легко можем найти строки, которые соответствуют определенному критерию. Например, получим строки, в которых цена больше 120:

|  |  |
| --- | --- |
| 1  2  3 | var selectedBooks = booksTable.Select("Price > 120");  foreach (var b in selectedBooks)      Console.WriteLine("{0} - {1}", b["Name"], b["Price"]); |

## Отношения между таблицами в DataSet

DataSet может содержать множество таблиц, которые могут быть связаны различными отношениями. Например, пусть у нас в dataset будет определена таблица производителей смартфонов и таблица самих смартфонов, которая связана с первой:

|  |  |
| --- | --- |
| 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  73  74  75  76  77  78  79  80 | static void Main(string[] args)  {      DataSet ds = new DataSet("Store");        // таблица компаний      DataTable companiesTable = new DataTable("Companies");      // два столбца таблицы Companies      DataColumn compIdColumn = new DataColumn("Id", Type.GetType("System.Int32"));      compIdColumn.Unique = true;      compIdColumn.AllowDBNull = false;      compIdColumn.AutoIncrement = true;      compIdColumn.AutoIncrementSeed = 1;      compIdColumn.AutoIncrementStep = 1;        DataColumn compNameColumn = new DataColumn("Name", Type.GetType("System.String"));      // добавляем столбцы      companiesTable.Columns.Add(compIdColumn);      companiesTable.Columns.Add(compNameColumn);      // добавляем таблицу в dataset      ds.Tables.Add(companiesTable);        // вторая таблица - смартфонов компаний      DataTable phonesTable = new DataTable("Phones");      DataColumn phoneIdColumn = new DataColumn("Id", Type.GetType("System.Int32"));      phoneIdColumn.Unique = true;      phoneIdColumn.AllowDBNull = false;      phoneIdColumn.AutoIncrement = true;      phoneIdColumn.AutoIncrementSeed = 1;      phoneIdColumn.AutoIncrementStep = 1;        DataColumn phoneNameColumn = new DataColumn("Name", Type.GetType("System.String"));      DataColumn phonePriceColumn = new DataColumn("Price", Type.GetType("System.Decimal"));      // столбец-внешний ключ      DataColumn phoneCompanyColumn = new DataColumn("CompanyId", Type.GetType("System.Int32"));      // добавляем столбцы в таблицу смартфонов      phonesTable.Columns.Add(phoneIdColumn);      phonesTable.Columns.Add(phoneNameColumn);      phonesTable.Columns.Add(phonePriceColumn);      phonesTable.Columns.Add(phoneCompanyColumn);      // добавляем таблицу смартфонов      ds.Tables.Add(phonesTable);        // установка отношений между таблицами      ds.Relations.Add("PhonesCompanies", companiesTable.Columns["Id"], phonesTable.Columns["CompanyId"]);        // Добавим ряд данных      DataRow apple = companiesTable.NewRow();      apple.ItemArray = new object[] { null, "Apple" };      companiesTable.Rows.Add(apple);      DataRow samsung = companiesTable.NewRow();      samsung.ItemArray = new object[] { null, "Samsung" };      companiesTable.Rows.Add(samsung);        DataRow iphone5 = phonesTable.NewRow();      iphone5.ItemArray = new object[] { null, "iPhone 5", 400, apple["Id"] };      phonesTable.Rows.Add(iphone5);        DataRow iphone6s = phonesTable.NewRow();      iphone6s.ItemArray = new object[] { null, "iPhone 6S", 600, apple["Id"] };      phonesTable.Rows.Add(iphone6s);        DataRow galaxy6 = phonesTable.NewRow();      galaxy6.ItemArray = new object[] { null, "Samsung Galaxy S6", 500, samsung["Id"] };      phonesTable.Rows.Add(galaxy6);        DataRow galaxyace2 = phonesTable.NewRow();      galaxyace2.ItemArray = new object[] { null, "Samsung Galaxy Ace 2", 200, samsung["Id"] };      phonesTable.Rows.Add(galaxyace2);        // выведем все смартфоны компании Apple      DataRow[] rows = apple.GetChildRows(ds.Relations["PhonesCompanies"]);      Console.WriteLine("Продукция компании Apple");      Console.WriteLine("Id \tСмартфон \tЦена");        foreach (DataRow r in rows)      {          Console.WriteLine("{0} \t{1} \t{2}", r["Id"], r["Name"], r["Price"]);      }      Console.Read();  } |



Ключевым моментом здесь является установка отношения между таблицами:

|  |  |
| --- | --- |
| 1 | ds.Relations.Add("PhonesCompanies", companiesTable.Columns["Id"], phonesTable.Columns["CompanyId"]); |

Первым параметром здесь идет название отношения, по которому потом на него можно будет ссылаться. Вторым параметром идет главный столбец - id компании в таблице компаний. Третий параметр - подчиненный столбец - id компании в таблице смартфонов. Подчиненный столбец зависит от главного. Вместо этих параметров мы можем передать объект DataRelation, который также будет иметь данные значения.

Затем используя отношение, мы можем получить для одной строки из таблицы компаний связанные строки из таблицы смартфонов:

|  |  |
| --- | --- |
| 1 | DataRow[] rows = apple.GetChildRows(ds.Relations["PhonesCompanies"]); |

### Внешние ключи

Кроме простого отношения между таблицами мы также можем задать внешние ключи, как в обычных базах данных. Для этого непосредственно перед строкой:

|  |  |
| --- | --- |
| 1 | ds.Relations.Add("PhonesCompanies", companiesTable.Columns["Id"], phonesTable.Columns["CompanyId"]); |

Добавим следующий код:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | ForeignKeyConstraint foreignKey = new ForeignKeyConstraint(companiesTable.Columns["Id"], phonesTable.Columns["CompanyId"])  {      ConstraintName = "PhonesCompaniesForeignKey",      DeleteRule = Rule.SetNull,      UpdateRule = Rule.Cascade  };  // добавляем внешний ключ в dataset  ds.Tables["Phones"].Constraints.Add(foreignKey);  // применяем внешний ключ  ds.EnforceConstraints = true;    ds.Relations.Add("PhonesCompanies", companiesTable.Columns["Id"], phonesTable.Columns["CompanyId"]); |

Внешний ключ представляет объект ForeignKeyConstraint, в конструктор которого передается два параметра: главный столбец и зависимый столбец. В данном случае это те же самые столбцы.

Также у него мы можем установить ряд свойств. Через свойство ConstraintName устанавливается название внешнего ключа. СвойстваDeleteRule и UpdateRule определют поведение при удалении или обновлении данных соответственно в главной таблице (то есть в данном случае в таблице компаний). Эти свойства принимают одно из значений перечисления Rule:

* Cascade: происходит автоматическое изменение или удаление строк в подчиненной таблице
* None: строки в подчиненной таблице не изменяются
* SetDefault: происходит установка значений по умолчанию для связанных строк (которое определяется через DataColumn.DefaultValue)
* SetNull: для связанных строк устанавливается значение DBNull

С помощью метода ds.Tables["Phones"].Constraints.Add(foreignKey); внешний ключ добавляется к таблице смартфонов. И затем через выражение ds.EnforceConstraints = true; происходит применение внешнего ключа.

И если мы удалим, например, одну из строк из таблицы компаний, то в таблице смартфонов все строки, которые связаны с удаленной, в столбце CompanyId будут иметь значение null.

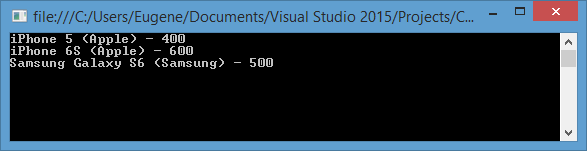
## LINQ to DataSet

Одним из преимуществ использования DataSet является то, что мы можем использовать для работы с данными в DataSet технологию LINQ.

Например, выведем из таблицы объекты по какому-нибудь условию:

|  |  |
| --- | --- |
| 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 | DataSet ds = new DataSet("Store");  // таблица компаний  DataTable companiesTable = new DataTable("Companies");  DataColumn compIdColumn = new DataColumn("Id", Type.GetType("System.Int32"));  DataColumn compNameColumn = new DataColumn("Name", Type.GetType("System.String"));  // добавляем столбцы  companiesTable.Columns.Add(compIdColumn);  companiesTable.Columns.Add(compNameColumn);  // добавляем таблицу в dataset  ds.Tables.Add(companiesTable);    // вторая таблица - смартфонов компаний  DataTable phonesTable = new DataTable("Phones");  DataColumn phoneIdColumn = new DataColumn("Id", Type.GetType("System.Int32"));  DataColumn phoneNameColumn = new DataColumn("Name", Type.GetType("System.String"));  DataColumn phonePriceColumn = new DataColumn("Price", Type.GetType("System.Decimal"));  DataColumn phoneCompanyColumn = new DataColumn("CompanyId", Type.GetType("System.Int32"));  // добавляем столбцы в таблицу смартфонов  phonesTable.Columns.Add(phoneIdColumn);  phonesTable.Columns.Add(phoneNameColumn);  phonesTable.Columns.Add(phonePriceColumn);  phonesTable.Columns.Add(phoneCompanyColumn);  // добавляем таблицу смартфонов  ds.Tables.Add(phonesTable);    // Добавим ряд данных  companiesTable.Rows.Add(new object[] { 1, "Apple" });  companiesTable.Rows.Add(new object[] { 2, "Samsung" });    phonesTable.Rows.Add(new object[] { 1, "iPhone 5", 400, 1 });  phonesTable.Rows.Add(new object[] { 2, "iPhone 6S", 600, 1});  phonesTable.Rows.Add(new object[] { 3, "Samsung Galaxy S6", 500, 2 });  phonesTable.Rows.Add(new object[] { 4, "Samsung Galaxy Ace 2", 200, 2});    var query = from phone in ds.Tables["Phones"].AsEnumerable()              from company in ds.Tables["Companies"].AsEnumerable()              where (int)phone["CompanyId"] == (int)company["Id"]              where (decimal)phone["Price"] >200              select new { Model = phone["Name"], Price = phone["Price"], Company = company["Name"] };    foreach (var phone in query)      Console.WriteLine("{0} ({1}) - {2}", phone.Model, phone.Company, phone.Price); |

Результат работы



В данном случае отображаются все смартфоны стоимостью больше 200 единиц. Чтобы проводить операции LINQ над таблицами, их надо привести к объекту IEnumerable: ds.Tables["Phones"].AsEnumerable(). По имени столбцов мы можем получить значения в соответствующих ячейках строк: phone["Price"].

Чтобы избежать необходимости преобразований в запросах, мы можем использовать типизированный метод Field<T>(), в который передается название столбца, для которого надо получить значение:

|  |  |
| --- | --- |
| 1  2  3  4  5 | var query = from phone in ds.Tables["Phones"].AsEnumerable()              from company in ds.Tables["Companies"].AsEnumerable()              where phone.Field<Int32>("CompanyId") == company.Field<Int32>("Id")              where phone.Field<Decimal>("Price") > 200              select new { Model = phone.Field<string>("Name"), Price = phone["Price"], Company = company["Name"] }; |

## DataSet и XML

Встроенные возможности DataSet позволяют производить некоторые операции с xml-документами, в частности, сохранять данные из dataset в xml и, наоборот, загружать из xml в dataset. Подобные возможности позволяют при необходимости использовать dataset в качестве посредника между базой данных и xml-файлами.

Сохранение данных из dataset в xml осуществляется с помощью метода **WriteXml**. Например, загрузим данные из БД в DataSet и затем сохраним их в xml:

|  |  |
| --- | --- |
| 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 sql = "SELECT \* FROM Users";  using (SqlConnection connection = new SqlConnection(connectionString))  {      connection.Open();      SqlDataAdapter adapter = new SqlDataAdapter(sql, connection);        DataSet ds = new DataSet("Users");      DataTable dt = new DataTable("User");      ds.Tables.Add(dt);      adapter.Fill(ds.Tables["User"]);        ds.WriteXml("usersdb.xml");      Console.WriteLine("Данные сохранены в файл");  } |

В метод WriteXml() передается имя файла, который после записи будет выглядеть примерно 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  32  33  34  35  36  37  38 | <?xml version="1.0" standalone="yes"?>  <Users>    <User>      <Id>2</Id>      <Name>Eugene</Name>      <Age>31</Age>    </User>    <User>      <Id>3</Id>      <Name>Tom</Name>      <Age>24</Age>    </User>    <User>      <Id>4</Id>      <Name>Ben</Name>      <Age>43</Age>    </User>    <User>      <Id>5</Id>      <Name>Sam</Name>      <Age>56</Age>    </User>    <User>      <Id>6</Id>      <Name>Tom</Name>      <Age>26</Age>    </User>    <User>      <Id>7</Id>      <Name>Alex</Name>      <Age>41</Age>    </User>    <User>      <Id>8</Id>      <Name>Adam</Name>      <Age>36</Age>    </User>  </Users> |

Для считывания данных из xml применяется метод **ReadXml()**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | DataSet ds = new DataSet();  ds.ReadXml("usersdb.xml");  // выбираем первую таблицу  DataTable dt = ds.Tables[0];    foreach (DataColumn column in dt.Columns)      Console.Write("\t{0}", column.ColumnName);  Console.WriteLine();  // перебор всех строк таблицы  foreach (DataRow row in dt.Rows)  {      var cells = row.ItemArray;      foreach (object cell in row.ItemArray)          Console.Write("\t{0}", cell);      Console.WriteLine();  } |