**Лабораторная работа**

**Использование ENTITY FRAMEWORK и LINQ для работы с базами данных.**

**Цель работы.** Ознакомиться с возможностями ENTITY FRAMEWORK и получить навыки написания LINQ запросов к объектам, связанным с таблицами базы данных СУБД MS SQL Server.

**1.ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

**1.1. Entity Framework Core**

**Entity Framework Core (EF Core**) представляет собой объектно-ориентированную, легковесную и расширяемую технологию от компании Microsoft для доступа к данным. EF Core является ORM-инструментом (object-relational mapping - отображения данных на реальные объекты). EF Core позволяет работать базами данных, но представляет собой более высокий уровень абстракции и позволяет работать с данными независимо от типа хранилища. Если на физическом уровне мы оперируем таблицами, индексами, первичными и внешними ключами, но на концептуальном уровне, который нам предлагает Entity Framework, мы уже работаем с объектами.

Entity Framework Core поддерживает множество различных систем баз данных. Таким образом, мы можем через EF Core работать с любой СУБД, если для нее имеется нужный провайдер.

По умолчанию на данный момент Microsoft предоставляет ряд встроенных провайдеров: для работы с MS SQL Server, для SQLite, для PostgreSQL. Также имеются провайдеры от сторонних поставщиков, например, для MySQL.

EF Core предоставляет универсальный API для работы с данными.

Центральной концепцией Entity Framework является понятие сущности. Сущность определяет набор данных, которые связаны с определенным объектом. Поэтому данная технология предполагает работу не с таблицами, а с объектами и их коллекциями.

Любая сущность, как и любой объект из реального мира, обладает рядом свойств. Например, если сущность описывает человека, то мы можем выделить такие свойства, как имя, фамилия, рост, возраст. Свойства необязательно представляют простые данные типа int или string, но могут также представлять и более комплексные типы данных. И у каждой сущности может быть одно или несколько свойств, которые будут отличать эту сущность от других и будут уникально определять эту сущность. Подобные свойства называют ключами.

При этом сущности могут быть связаны ассоциативной связью один-ко-многим, один-ко-одному и многие-ко-многим, подобно тому, как в реальной базе данных происходит связь через внешние ключи.

Отличительной чертой Entity Framework Core, как технологии ORM, является использование запросов LINQ для выборки данных из БД. С помощью LINQ мы можем создавать различные запросы на выборку объектов, в том числе связанных различными ассоциативными связями. А Entity Framework при выполнении запроса транслирует выражения LINQ в выражения, понятные для конкретной СУБД (как правило, в выражения SQL).

**Модели, Fluent API и аннотации данных**

Все сущности, с которыми работает Entity Framework, определяются в виде классов моделей. При этом EF Core использует ряд условностей для сопоставления классов моделей с таблицами. Например, названия столбцов должны соответствовать названиям свойств и т.д. В этом случае Entity Framework сможет сопоставить столбцы таблицы и свойства модели.

Однако с помощью таких механизмов, как Fluent API и аннотации данных мы можем добавить дополнительные правила конфигурации, либо переопределить используемые условности.

**Сопоставление типов**

Типы SQL Serverа и C# сопоставляются следующим образом:

1. int : int

2. bit : bool

3. char : string

4. date : DateTime

5. datetime : DateTime

6. datetime2 : DateTime

7. decimal : decimal

8. float : double

9. money : decimal

10. nchar : string

11. ntext : string

12. numeric : decimal

13. nvarchar : string

14. real : float

15. smallint : short

16. text : string

17. tinyint : byte

18. varchar : string

NULL и NOT NULL

Все первичные ключи по умолчанию имеют определение NOT NULL.

Столбцы, сопоставляемые со свойствами ссылочных типов (string, array), в базе данных имеют определение NULL, а все значимые типы (DateTime, bool, char, decimal, int, double, float) - NOT NULL.

Если свойство имеет тип Nullable<T>, то оно сопоставляется со столбцом с определением NULL.

Ключи

Entity Framework требует наличия первичного ключа, так как это позволяет ему отслеживать объекты. По умолчанию в качестве ключей EF рассматривает свойства с именем Id или [Название\_типа]Id (например, PostId в классе Post).

Как правило, ключи имеют тип int или GUID, но также могут представлять и любой другой примитивный тип.

**Подключение к базе данных**

Основу функциональности Entity Framework Core для работы с MS SQL Server составляют классы, которые располагаются в пространстве имен Microsoft.EntityFrameworkCore. Среди всего набора классов этого пространства имен следует выделить следующие:

* DbContext: определяет контекст данных, используемый для взаимодействия с базой данных
* DbSet/DbSet<TEntity>: представляет набор объектов, которые хранятся в базе данных
* DbContextOptionsBuilder: устанавливает параметры подключения

В любом приложении, работающим с БД через Entity Framework, нам нужен будет объект контекста (объект класса, производного от DbContext).

Для взаимодействия с базой данных для контекста данных должна быть определена **конфигурация подключения**. Для ее установки можно применять два способа:

1. Переопределение у класса контекста данных метода OnConfiguring()
2. Передача конфигурации в конструктор базового класса DbContext

Переопределение у класса контекста данных метода OnConfiguring:

|  |
| --- |
| public class ApplicationContext: DbContext  {      public DbSet<User> Users { get; set; }        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)      {         optionsBuilder.UseSqlServer(@"Server=(localdb)\mssqllocaldb;  Database=helloappdb;Trusted\_Connection=True;");      }    } |
|  |

В этот метод передается объект DbContextOptionsBuilder, который позволяет создать параметры подключения. Для их создания вызывается метод UseSqlServer, который указывает на конкретного провайдера и его параметр содержит строку подключения.

**Строка подключения** разбивается на несколько частей:

* Server: название сервера. В данном случае используется специальный движок MS SQL Server - localdb, который предназначен специально для нужд разработки. Для полноценного MS SQL Server Express этот параметр, как правило, имеет значение .\SQLEXPRESS
* Database: название файла базы данных без расширения mdf
* Trusted\_Connection: устанавливает проверку подлинности

Второй способ предполагает передачу в конструктор базового класса объекта DbContextOptions, который инкапсулирует параметры конфигурации. Для применения этого способа изменим класс контекста следующим образом:

|  |
| --- |
| namespace HelloApp  {    public class ApplicationContext: DbContext    {        public DbSet<User> Users { get; set; }          public ApplicationContext(DbContextOptions<ApplicationContext> options): base(options)        {        }    }  } |

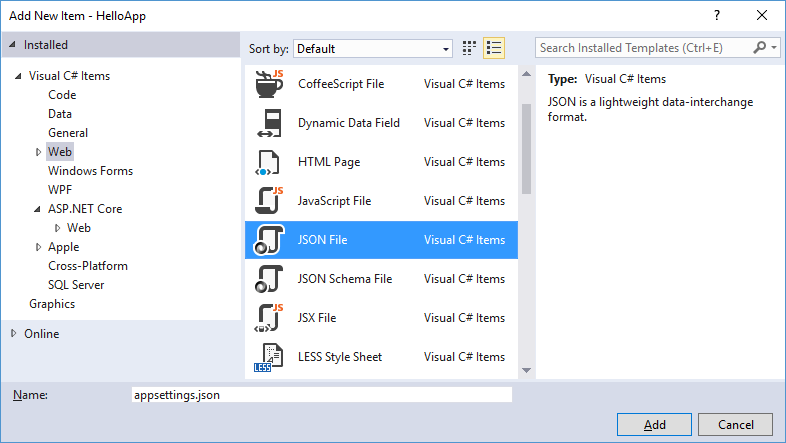
Тогда можно создать объект класса контекста следующим образом:

|  |
| --- |
| var optionsBuilder = new DbContextOptionsBuilder<ApplicationContext>();    var options = optionsBuilder          .UseSqlServer(@"Server=(localdb)\mssqllocaldb;  Database=helloappdb;Trusted\_Connection=True;")          .Options;    using (ApplicationContext db = new ApplicationContext(options))  {      var users = db.Users.ToList();      foreach (User u in users)          Console.WriteLine($"{u.Id}.{u.Name} - {u.Age}");  } |
|  |
|  |

Здесь также применяется метод UseSqlServer класса DbContextOptionsBuilder для создания конфигурации по строке подключения. Результат этой операции - объект DbContextOptions затем передается в контекст данных. А контекст данных далее передает этот параметр в конструктор базового класса.

Оба выше представленных способа вполне работают, однако в том определении, в котором они представлены, они имеют один недостаток - строка подключения жестко определена в коде C#. И было бы неплохо, если бы она была бы определена в каком-нибудь внешнем файле подключения, где мы ее могли бы поменять без перекомпиляции приложения.

Для этого добавим в проект новый элемент JSON File, который назовем appsettings.json:



Определим в файле appsettings.json следующий код:

|  |
| --- |
| {    "ConnectionStrings": {      "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=helloappdb;Trusted\_Connection=True;"    }  } |

Здесь опять же определена наша строка подключения.

Чтобы ее использовать, нам надо добавить в проект через Nuget ряд пакетов:

* Microsoft.Extensions.Configuration
* Microsoft.Extensions.Configuration.FileExtensions
* Microsoft.Extensions.Configuration.Json

Далее для работы с БД в классе Program определим следующий код:

|  |
| --- |
| using System;  using System.Linq;  using Microsoft.Extensions.Configuration;  using Microsoft.EntityFrameworkCore;  using System.IO;    namespace HelloApp  {      public class Program      {          public static void Main(string[] args)          {              var builder = new ConfigurationBuilder();              // установка пути к текущему каталогу              builder.SetBasePath(Directory.GetCurrentDirectory());              // получаем конфигурацию из файла appsettings.json              builder.AddJsonFile("appsettings.json");              // создаем конфигурацию              var config = builder.Build();              // получаем строку подключения              string connectionString = config.GetConnectionString("DefaultConnection");                var optionsBuilder = new DbContextOptionsBuilder<ApplicationContext>();              var options = optionsBuilder                  .UseSqlServer(connectionString)                  .Options;                using (ApplicationContext db = new ApplicationContext(options))              {                  var users = db.Users.ToList();                  foreach (User u in users)                  {                      Console.WriteLine($"{u.Id}.{u.Name} - {u.Age}");                  }              }              Console.Read();          }      }  } |

Для создания конфигурации применяется класс ConfigurationBuilder. Метод AddJsonFile() добавляет все настройки из файла конфигурации. С помощью метода Build() создается объект конфигурации, из которого мы можем получить строку подключения:

|  |
| --- |
| string connectionString = config.GetConnectionString("DefaultConnection"); |

Для получения строки подключения используется ее имя - "DefaultConnection", которое указано в appsettings.json.

В остальном работа с контекстом данных будет протекать также.

**Fluent API**

Fluent API представляет набор методов, которые определяют сопоставление между классами и их свойствами и таблицами и их столбцами. Как правило, функционал Fluent API задействуется при переопределении метода OnModelCreating:

public class ApplicationContext: DbContext

{

public DbSet<User> Users { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseSqlServer(@"Server=(localdb)\mssqllocaldb;Database=helloappdb;Trusted\_Connection=True;");

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

// использование Fluent API

base.OnModelCreating(modelBuilder);

}

}

**Аннотации**

Аннотации представляют настройку классов моделей с помощью атрибутов. Большинство подобных атрибутов располагаются в пространстве System.ComponentModel.DataAnnotations, которое нам надо подключить в файл c# перед использованием аннотаций. Например:

using System.ComponentModel.DataAnnotations;

namespace HelloApp

{

public class User

{

public int Id { get; set; }

[Required]

public string Name { get; set; }

public int Age { get; set; }

}

}

В данном случае атрибут Required представляет аннотацию, которая указывает, что свойство Name обязательно должно иметь значение.

Таким образом, мы можем использовать три подхода к определению моделей:

* Условности (conventions)
* Fluent API
* Аннотации данных

**Миграции**

Стоит учитывать, что если мы меняем модель, например, добавляем в нее какие-то новые свойства или удаляем некоторые свойства, то чтобы база данных пришла в соответствие с моделями, необходимо заново сгенерировать и выполнить миграцию. И то же самое относится и в аннотациям данных и Fluent API: если мы добавляем к модели или удаляем какие-либо аннотации или применяем правила Fluent API, то тем самым мы также изменяем модель. И в этом случае также для того, чтобы бд соответствовала модели, надо выполнить миграции.

Для создания миграции в окне **Package Manager Console** вводится следующая команда:

|  |  |
| --- | --- |
| 1 | Add-Migration название\_миграции |

Название миграции представляет произвольное название, главное чтобы все миграции в проекте имели разные названия.

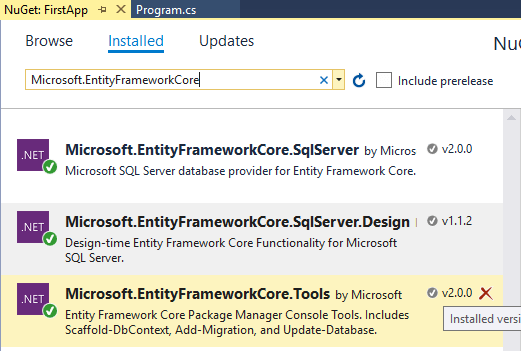
После создания миграции ее надо выполнить с помощью команды:

|  |  |
| --- | --- |
| 1 | Update-Database |

**Подключение к существующей базе данных**

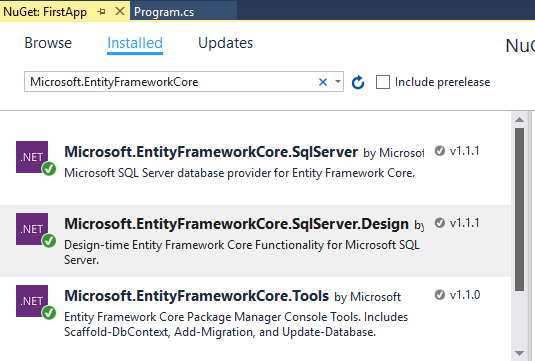
Нередко база данных для подключения уже имеется. В этом случае наша работа упрощается в том плане, что нам не надо выполнять миграции по созданию БД.

Для подключения к существующей базе данных создадим новый проект. Как и в прошлой теме это будет проект по типу **Console App (.NET Core)**, который назовем FirstApp.



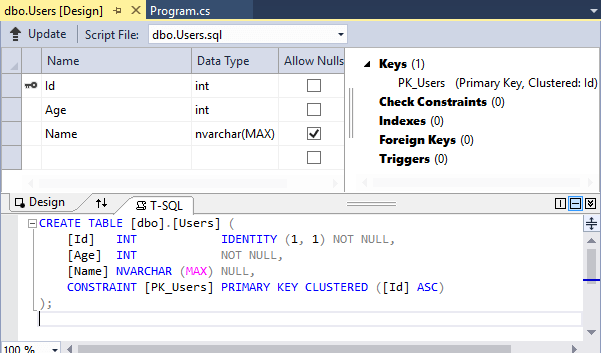
Для работы с существующей БД MS SQL Server нам надо добавить три пакета:

* **Microsoft.EntityFrameworkCore.SqlServer**
* **Microsoft.EntityFrameworkCore.Tools**
* **Microsoft.EntityFrameworkCore.SqlServer.Design**



Пакет "Microsoft.EntityFrameworkCore.SqlServer" представляет функицональность Entity Framework для работы с MS SQL Server, а два других пакета - "Microsoft.EntityFrameworkCore.Tools" и "Microsoft.EntityFrameworkCore.SqlServer.Design" понадобятся нам в дальнейшем для создания классов по базе данных, то есть reverse engineering.

К примеру возьмем базу данных, которая называется helloappdb.mdf и имеет одну таблицу Users с тремя столбцами Id, Name и Age.



Чтобы подключиться к базе данных, нам надо будет добавить в проект классы моделей, которые соответствуют определениям таблиц, и класс контекста данных, который соответствует БД. То есть в данном случае мы могли бы вручную добавить в проект классы моделей и класс контекста данных, как в прошлой теме.

Однако добавление классов вручную имеет свои недостатки. Так, если база данных имеет не одну, а множество таблиц, связанных различными ключами, то у нас может возникнуть проблема, как все эти отношения отобразить между класса на C#. Ну и кроме того, это просто долго и может занять некоторое время.

Для решения этих проблем в Entity Framework Core предусмотрена функция **Reverse Engineering**, которая позволяет автоматически создать все необходимые классы по базе данных. Чтобы воспользоваться этой функцией перейдем в Visual Studio к окну **Package Manager Console**. Его открыть можно, перейдя в меню **Tools** –> **NuGet Package Manager** –> **Package Manager Console**

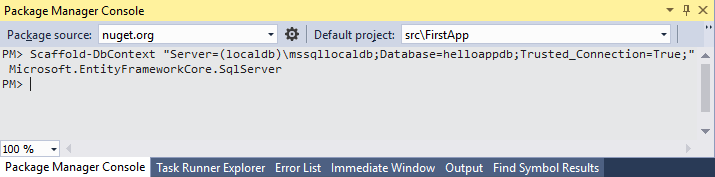
Далее в Package Manager Console выполним следующую команду:

Scaffold-DbContext "Server=(localdb)\mssqllocaldb;Database=helloappdb;Trusted\_Connection=True;" Microsoft.EntityFrameworkCore.SqlServer

или

Scaffold-DbContext "Server=(localdb)\mssqllocaldb;Database= helloappdb;Trusted\_Connection=True;" Microsoft.EntityFrameworkCore.SqlServer -ContextDir Data -OutputDir Models

Здесь в качестве параметра команде Scaffold-DbContext передается строка подключения с указанием сервера и названием базы данных.



После выполнения этой команды в проект будет добавлен класс Users:

|  |
| --- |
| public partial class Users  {  public int Id { get; set; }  public int Age { get; set; }  public string Name { get; set; }  } |

И также будет добавлен класс контекста данных, который будет называться по имени базы данных плюс суффикс "Context":

|  |
| --- |
| using System;  using Microsoft.EntityFrameworkCore;  using Microsoft.EntityFrameworkCore.Metadata;    namespace FirstApp  {  public partial class helloappdbContext : DbContext  {  protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)  {  if (!optionsBuilder.IsConfigured)  {  optionsBuilder.UseSqlServer(@"Server=(localdb)\mssqllocaldb;Database=helloappdb;Trusted\_Connection=True;");  }  }    protected override void OnModelCreating(ModelBuilder modelBuilder)  {  }    public virtual DbSet<Users> Users { get; set; }  }  } |

При Reverse Engineering EF по умолчанию не применяет правила плюрализации, поэтому создается класс Users, а не User. Если такое название не устраивает, то мы можем вручную изменить на User:

|  |
| --- |
| public partial class User  {  // содержимое класса  } |

Тогда соответственно надо изменить свойство в контексте данных:

|  |
| --- |
| public virtual DbSet<User> Users { get; set; } |

И после этого мы сможем работать с базой данных. Для этого изменим код класса Program:

|  |
| --- |
| using System;  using System.Linq;    namespace FirstApp  {  class Program  {  static void Main(string[] args)  {  using (helloappdbContext db = new helloappdbContext())  {  // получаем объекты из бд и выводим на консоль  var users = db.Users.ToList();  Console.WriteLine("Список объектов:");  foreach (User u in users)  {  Console.WriteLine($"{u.Id}.{u.Name} - {u.Age}");  }  }  Console.ReadKey();  }  }  } |

**1.2. Выполнение типовых операций с использованием DbContext API**

Большинство операций с данными так или иначе представляют собой CRUD операции (Create, Read, Update, Delete), то есть создание, получение, обновление и удаление. Entity Framework Core позволяет легко выполнять все эти действия.

Для примера создадим проект по типу **Console App (.NET Core)**. И после создания проекта сразу добавим в него функциональность EF Core. Для этого в проект через NuGet пакеты **Microsoft.EntityFrameworkCore.SqlServer** и **Microsoft.EntityFrameworkCore.Tools**.

Затем добавим в проект класс User, объекты которого будут храниться в базе данных:

|  |
| --- |
| public class User  {  public int Id { get; set; }  public string Name { get; set; }  public int Age { get; set; }  } |

И добавим класс контекста данных ApplicationContext:

|  |
| --- |
| using Microsoft.EntityFrameworkCore;    namespace HelloApp  {  public class ApplicationContext : DbContext  {  public DbSet<User> Users { get; set; }    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)  {  optionsBuilder.UseSqlServer(@"Server=(localdb)\mssqllocaldb;Database=helloappdb; Trusted\_Connection=True;");  }  }  } |

Затем выполним миграции для создания базы данных. Для этого перейдем в Visual Studio к окну Package Manager Console и последовательно выполним следующие команды:

|  |
| --- |
| Add-Migration Initial  Update-Database |

После создания базы данных определим в классе Program все базовые операции с данными:

|  |
| --- |
| using System;  using System.Linq;    namespace HelloApp  {  public class Program  {  public static void Main(string[] args)  {  // Добавление  using (ApplicationContext db = new ApplicationContext())  {  User user1 = new User { Name = "Tom", Age = 33 };  User user2 = new User { Name = "Alice", Age = 26 };    // Добавление  db.Users.Add(user1);  db.Users.Add(user2);  db.SaveChanges();  }    // получение  using (ApplicationContext db = new ApplicationContext())  {  // получаем объекты из бд и выводим на консоль  var users = db.Users.ToList();  Console.WriteLine("Данные после добавления:");  foreach (User u in users)  {  Console.WriteLine($"{u.Id}.{u.Name} - {u.Age}");  }  }    // Редактирование  using (ApplicationContext db = new ApplicationContext())  {  // получаем первый объект  User user = db.Users.FirstOrDefault();  if(user!=null)  {  user.Name = "Bob";  user.Age = 44;  //обновляем объект  //db.Users.Update(user);  db.SaveChanges();  }  // выводим данные после обновления  Console.WriteLine("\nДанные после редактирования:");  var users = db.Users.ToList();  foreach (User u in users)  {  Console.WriteLine($"{u.Id}.{u.Name} - {u.Age}");  }  }    // Удаление  using (ApplicationContext db = new ApplicationContext())  {  // получаем первый объект  User user = db.Users.FirstOrDefault();  if (user != null)  {  //удаляем объект  db.Users.Remove(user);  db.SaveChanges();  }  // выводим данные после обновления  Console.WriteLine("\nДанные после удаления:");  var users = db.Users.ToList();  foreach (User u in users)  {  Console.WriteLine($"{u.Id}.{u.Name} - {u.Age}");  }  }  Console.Read();  }  }  } |

И после выполнения мы получим следующий консольный вывод:

Данные после добавления

1.Tom - 33

2.Alice - 26

Данные после редактирования

1.Bob - 44

2.Alice - 26

Данные после удаления

2.Alice - 26

1. **Добавление**

Для добавления объекта используется метод **Add**, определенный у класса DbSet, в который передается добавляемый объект:

|  |
| --- |
| db.Users.Add(user2);  db.SaveChanges(); |

Метод Add устанавливает значение Added в качестве состояния нового объекта. Поэтому метод db.SaveChanges() сгенерирует выражение INSERT для вставки модели в таблицу.

Если нам надо добавить сразу несколько объектов, то мы можем воспользоваться методом **AddRange()**:

|  |
| --- |
| User user1 = new User { Name = "Tom", Age = 33 };  User user2 = new User { Name = "Alice", Age = 26 };  db.Users.AddRange(user1, user2); |

1. **Удаление**

Удаление производится с помощью метода **Remove**:

|  |
| --- |
| db.Users.Remove(user);  db.SaveChanges(); |

Данный метод установит статус объекта в Deleted, благодаря чему Entity Framework при выполнении метода db.SaveChanges() сгенерирует SQL-выражение DELETE.

Если необходимо удалить сразу несколько объектов, то можно использовать метод **RemoveRange()**:

|  |
| --- |
| User user1 = db.Users.FirstOrDefault();  User user2 = db.Users.LastOrDefault();  db.Users.RemoveRange(user1, user2); |

**Редактирование**

При изменении объекта Entity Framework сам отслеживает все изменения, и когда вызывается метод **SaveChanges()**, будет сформировано SQL-выражение UPDATE для данного объекта, которое обновит объект в базе данных.

Но надо отметить, что в данном случае действие контекста данных ограничивается пределами конструкции using. Но рассмотрим другой пример. Мы получаем объект в одном месте,а обновляем в другом. Например:

|  |
| --- |
| User user = null;  using (ApplicationContext db = new ApplicationContext())  {  // получаем объект  user = db.Users.FirstOrDefault();  Console.WriteLine("Данные до редактирования:");  var users = db.Users.ToList();  foreach (User u in users)  {  Console.WriteLine($"{u.Id}.{u.Name} - {u.Age}");  }  }  //...................    // Редактирование  using (ApplicationContext db = new ApplicationContext())  {  // Редактирование  if (user != null)  {  user.Name = "Sam";  user.Age = 33;  }  db.SaveChanges();  // выводим данные после обновления  Console.WriteLine("\nДанные после редактирования:");  var users = db.Users.ToList();  foreach (var u in users)  {  Console.WriteLine($"{u.Id}.{u.Name} - {u.Age}");  }  } |

Несмотря на то, что объект user не равен null, имеется в базе данных, но во втором блоке using обновления соответствующего объекта в БД не произойдет. И в этом случае нам надо использовать метод Update:

|  |
| --- |
| // Редактирование  using (ApplicationContext db = new ApplicationContext())  {  // Редактирование  if (user != null)  {  user.Name = "Sam";  user.Age = 33;  db.Users.Update(user);  }  db.SaveChanges();  // выводим данные после обновления  Console.WriteLine("\nДанные после редактирования:");  var users = db.Users.ToList();  foreach (var u in users)  {  Console.WriteLine($"{u.Id}.{u.Name} - {u.Age}");  }  } |

При необходимости обновить одновременно несколько объектов, применяется метод **UpdateRange()**:

|  |
| --- |
| db.Users.UpdateRange(user1, user2); |

**1.3. GitHub Actions**

GitHub Actions — это встроенная в платформу GitHub система для автоматизации рабочих процессов, включая сборку, тестирование и деплой программного обеспечения. С её помощью разработчики могут автоматически выполнять задачи, связанные с управлением и разработкой проектов, прямо в репозиториях GitHub.

Основные возможности GitHub Actions:

1. Автоматизация процессов.
   * GitHub Actions позволяет автоматизировать задачи, такие как тестирование кода, сборка приложений, развертывание на серверах и отправка уведомлений. Это помогает ускорить процесс разработки и снизить вероятность ошибок.
2. Интеграция с GitHub
   * GitHub Actions тесно интегрирована с экосистемой GitHub, что упрощает настройку автоматизации в контексте репозитория. Она поддерживает триггеры на событиях, происходящих в репозитории, такие как push, pull request, создание issue и другие.
   * Использование workflows. Рабочие процессы (workflows) определяются с помощью YAML файлов, которые хранятся в репозиториях. Эти файлы описывают, какие задачи должны быть выполнены, когда они должны запускаться и на каких условиях.
3. Компоненты workflows: Jobs и Steps
   * Jobs — это наборы шагов, которые выполняются в отдельных виртуальных средах. Jobs могут быть настроены на выполнение параллельно или последовательно.
   * Steps — это отдельные задачи внутри job, которые могут включать выполнение команд shell, использование готовых actions или выполнение пользовательских скриптов.
4. Actions
   * Actions — это настраиваемые компоненты, которые могут быть использованы в steps для выполнения конкретных задач, таких как установка зависимостей, тестирование кода или деплой. GitHub предоставляет библиотеку готовых actions, доступных в GitHub Marketplace, и пользователи могут создавать свои собственные actions.
5. Использование секретов
   * GitHub Actions поддерживает хранение конфиденциальной информации, такой как API ключи и токены доступа, в секциях secrets, что обеспечивает безопасность при выполнении рабочих процессов.
6. Широкая поддержка языков и платформ
   * GitHub Actions поддерживает множество языков программирования и платформ, что делает её универсальным инструментом для различных типов проектов.

GitHub Actions стала важным инструментом для DevOps и автоматизации процессов разработки, предоставляя мощные и гибкие возможности прямо в рамках GitHub.

GitHub Actions использует несколько ключевых компонентов для создания и управления автоматизированными процессами в репозиториях. Эти компоненты включают workflows, jobs, steps и actions. Вот описание каждого из них:

**1. Workflows (Рабочие процессы)**

**Workflows** — это последовательности автоматизированных действий, которые определяют, как и когда должны выполняться задачи в проекте. Workflows определяются в YAML файлах, которые хранятся в каталоге .github/workflows вашего репозитория.

**Основные аспекты workflows:**

* **Triggers (Триггеры)**: Workflows могут быть запущены на основе различных событий, таких как push, pull request, создание issue, или даже по расписанию (например, ежедневно).
* **Объединение jobs**: Workflows могут содержать один или несколько jobs, которые выполняются в зависимости от заданных условий.

**2. Jobs (Задания)**

**Jobs** — это наборы шагов, которые выполняются в отдельной виртуальной среде. Каждый job выполняется на отдельной виртуальной машине или контейнере, что позволяет запускать их параллельно или последовательно.

**Особенности jobs:**

* **Изолированность**: Каждый job работает в своей изолированной среде, что обеспечивает независимость выполнения.
* **Платформы**: Jobs могут быть настроены для выполнения на различных операционных системах, таких как Ubuntu, Windows или macOS.
* **Зависимости**: Jobs могут быть связаны зависимостями, то есть один job может быть настроен так, чтобы выполняться только после успешного завершения другого job.

**3. Steps (Шаги)**

**Steps** — это отдельные задачи или команды, которые выполняются в рамках jobs. Каждый step выполняется последовательно, в порядке их определения в job.

**Основные элементы steps:**

* **Использование команд shell**: Steps могут включать команды shell, такие как выполнение скриптов или вызов программ.
* **Использование actions**: Steps могут использовать готовые или пользовательские actions для выполнения специфических задач, таких как установка зависимостей, тестирование или деплой.

**4. Actions**

**Actions** — это специализированные команды, которые можно использовать в steps для выполнения конкретных задач. Actions могут быть встроенными, полученными из GitHub Marketplace или созданными пользователями.

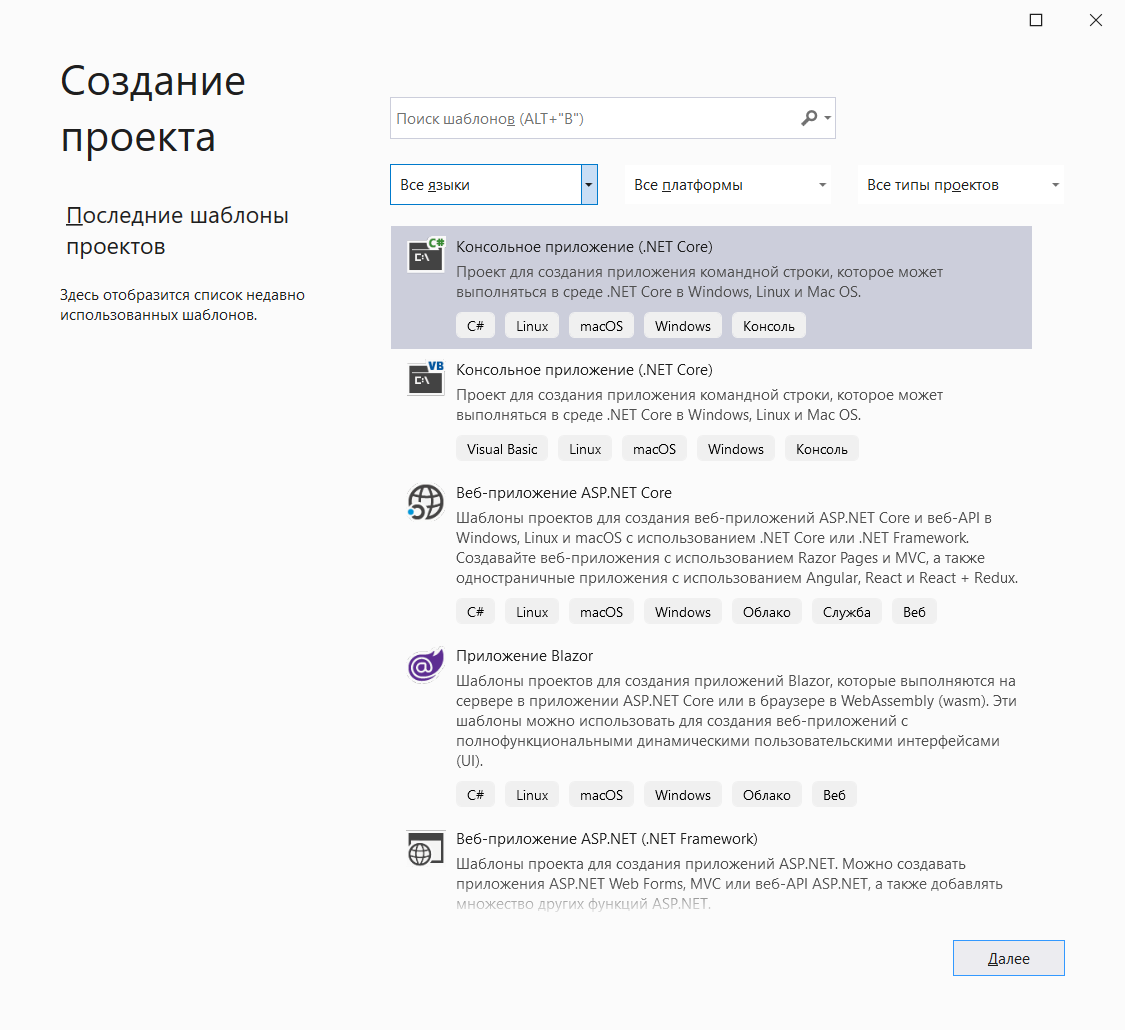
**Типы actions:**

* **Официальные actions**: GitHub предоставляет официальные actions для выполнения общих задач, таких как actions/checkout для клонирования репозитория или для настройки окружения.
* **Пользовательские actions**: Разработчики могут создавать свои собственные actions, которые могут быть повторно использованы в разных workflows. Эти actions могут быть написаны на различных языках программирования, таких как JavaScript или PowerShell, или упакованы в виде Docker контейнеров.

Эти компоненты взаимодействуют друг с другом, позволяя создавать мощные и гибкие автоматизированные процессы, которые помогают упростить и ускорить разработку и развертывание программного обеспечения.

**2. ПРИМЕР - создание консольного приложения для тестирования различных LINQ запросов к базе данных MS SQL Server**

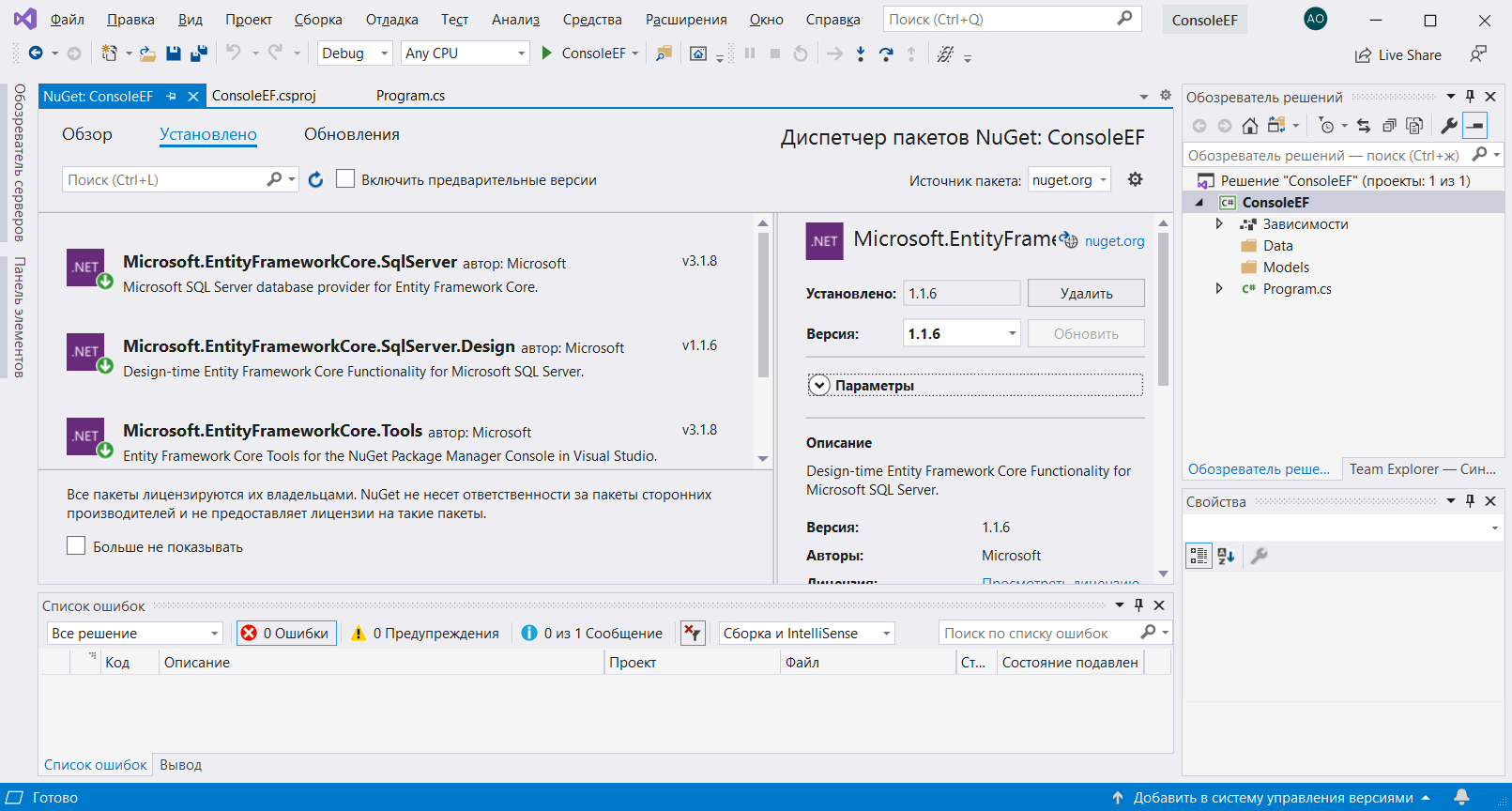
2.1. Создаем пустой проект консольного приложения по типу **Console App (.NET Core)**.



2.2. Перейдем в Visual Studio к окну Package Manager Console. Его открыть можно, перейдя в меню Tools –> NuGet Package Manager –> Package Manager Console

2.3. Для работы с существующей БД MS SQL Server нам надо добавить три пакета:

* **Microsoft.EntityFrameworkCore.SqlServer**
* **Microsoft.EntityFrameworkCore.Tools**
* **Microsoft.EntityFrameworkCore.SqlServer.Design**



2.4. В Package Manager Console выполним следующую команду:

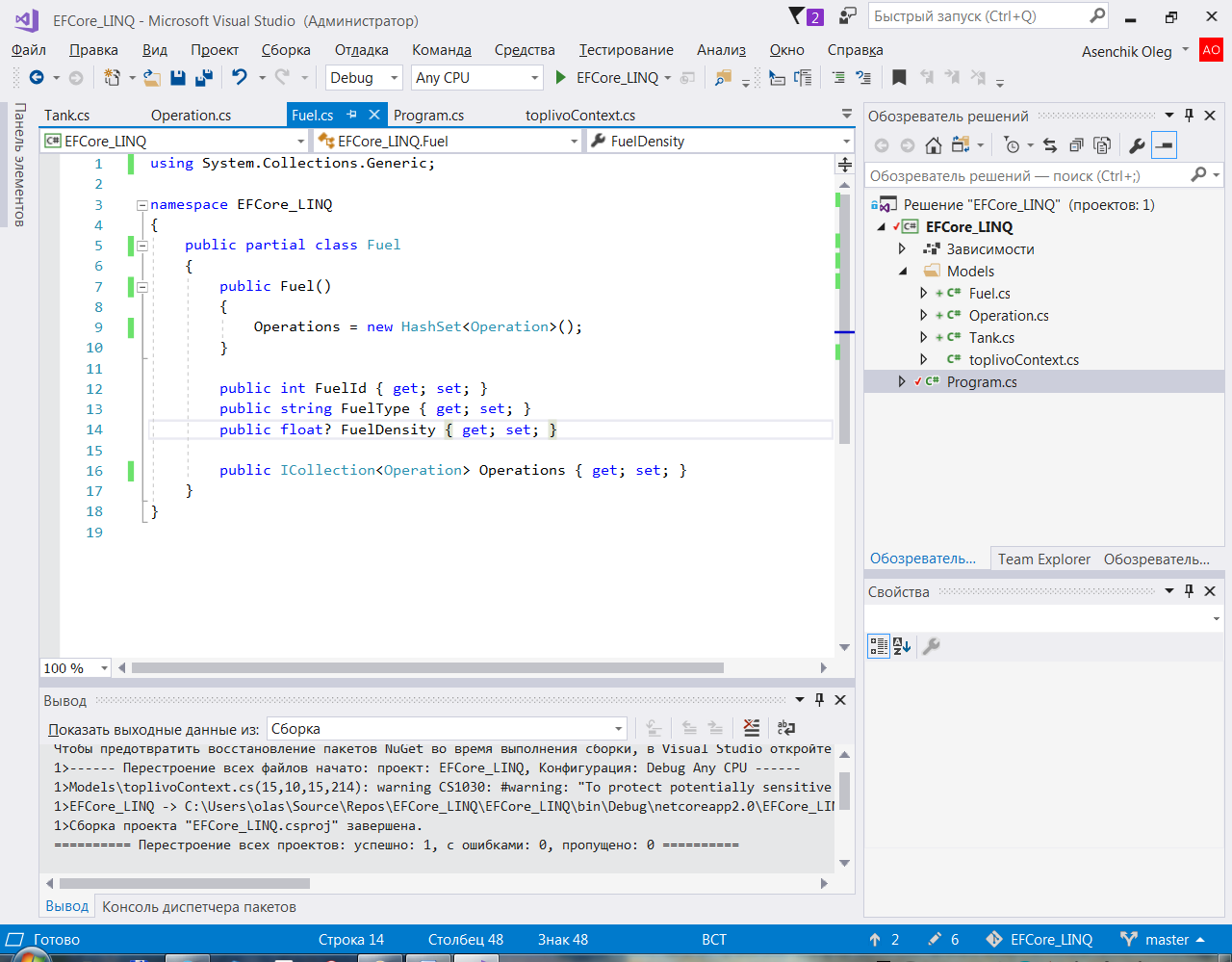
Scaffold-DbContext "Server=.\sqlexpress;Database=toplivo;Trusted\_Connection=True;" Microsoft.EntityFrameworkCore.SqlServer

или

Scaffold-DbContext "Server=(localdb)\mssqllocaldb;Database=toplivo; Trusted\_Connection =True;" Microsoft.EntityFrameworkCore.SqlServer

в зависимости от места нахождения базы данных.

2.5. Переименуем классы сущностей, используя форму единственного числа и переместим четыре сгенерированных класса в папку Models, создав ее предварительно:



Класс контекста toplivoContext

using Microsoft.EntityFrameworkCore;

namespace EFCore\_LINQ

{

public partial class toplivoContext: DbContext

{

public virtual DbSet<Fuel> Fuels { get; set; }

public virtual DbSet<Operation> Operations { get; set; }

public virtual DbSet<Tank> Tanks { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

if (!optionsBuilder.IsConfigured)

{

#warning To protect potentially sensitive information in your connection string, you should move it out of source code. See http://go.microsoft.com/fwlink/?LinkId=723263 for guidance on storing connection strings.

optionsBuilder.UseSqlServer(@"Server=.\sqlexpress;Database=toplivo;Trusted\_Connection=True;");

}

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Fuel>(entity =>

{

entity.HasKey(e => e.FuelId);

entity.Property(e => e.FuelId).HasColumnName("FuelID");

entity.Property(e => e.FuelType).HasMaxLength(50);

});

modelBuilder.Entity<Operation>(entity =>

{

entity.HasKey(e => e.OperationId);

entity.Property(e => e.OperationId).HasColumnName("OperationID");

entity.Property(e => e.Date).HasColumnType("date");

entity.Property(e => e.FuelId).HasColumnName("FuelID");

entity.Property(e => e.IncExp).HasColumnName("Inc\_Exp");

entity.Property(e => e.TankId).HasColumnName("TankID");

entity.HasOne(d => d.Fuel)

.WithMany(p => p.Operations)

.HasForeignKey(d => d.FuelId)

.OnDelete(DeleteBehavior.Cascade)

.HasConstraintName("FK\_Operations\_Fuels");

entity.HasOne(d => d.Tank)

.WithMany(p => p.Operations)

.HasForeignKey(d => d.TankId)

.OnDelete(DeleteBehavior.Cascade)

.HasConstraintName("FK\_Operations\_Tanks");

});

modelBuilder.Entity<Tank>(entity =>

{

entity.HasKey(e => e.TankId);

entity.Property(e => e.TankId).HasColumnName("TankID");

entity.Property(e => e.TankMaterial).HasMaxLength(20);

entity.Property(e => e.TankPicture).HasMaxLength(50);

entity.Property(e => e.TankType).HasMaxLength(20);

});

}

}

}

Класс Fuel одной из сущностей

using System.Collections.Generic;

namespace EFCore\_LINQ

{

public partial class Fuel

{

public Fuel()

{

Operations = new HashSet<Operation>();

}

public int FuelId { get; set; }

public string FuelType { get; set; }

public float? FuelDensity { get; set; }

public ICollection<Operation> Operations { get; set; }

}

}

2.4. В классе Program создадим необходимые методы для выполнения типовых действий:

2.4.1.Вывод на экран результатов выполнения LINQ запроса:

static void Print(string sqltext, IEnumerable items)

{

Console.WriteLine(sqltext);

Console.WriteLine("Записи: ");

foreach (var item in items)

{

Console.WriteLine(item.ToString());

}

Console.WriteLine();

Console.ReadKey();

}

4.2. Выполнение операций выборки данных:

static void Select(toplivoContext db)

{

// Определение LINQ запроса 1

var queryLINQ1 = from f in db.Operations

join t in db.Fuels

on f.FuelId equals t.FuelId

where (f.IncExp > 0 && f.Date.Year == 2016)

orderby f.FuelId descending

select new

{

Код\_операции = f.OperationId,

Название\_топлива = t.FuelType,

Приход\_Расход = f.IncExp,

Месяц = f.Date.Month

};

//то же, используя методы расширений

//var queryLINQ1 = db.Operations.Where(f => (f.Inc\_Exp > 0 && f.Date.Value.Year == 2016))

//.OrderBy(f => f.FuelId)

//.Join(db.Fuels, f => f.FuelId, t => t.FuelId, (f, t) => new { f.OperationID, t.FuelType, f.Inc\_Exp, f.Date.Value.Month });

string comment = "1. Результат выполнения запроса на выборку отсортированных записей из двух таблиц, удовлетворяющих заданному условию : \r\n";

//для наглядности выводим не более 5 записей

Print(comment, queryLINQ1.Take(5).ToList());

// Определение LINQ запроса 2

var queryLINQ2 = from o in db.Operations

where (o.IncExp > 0 && o.Date.Year == 2016)

group o.IncExp by o.FuelId into gr

select new

{

Код\_топлива = gr.Key,

Количество\_топлива = gr.Sum()

};

//то же, используя методы расширений:

//var queryLINQ2 = db.Operations.Where(o => ((o.IncExp > (Single?)0) && (o.Date.Value.Year == 2016)))

// .GroupBy(o => o.FuelId, o => o.IncExp)

// .Select(gr => new

// {

// Код\_топлива = gr.Key,

// Количества\_топлива = gr.Sum()

// }

// );

comment = "2. Результат выполнения запроса на выборку сгруппированных записей из одной таблицы, удовлетворяющих заданному условию, с выполнением групповой операции суммирования : \r\n";

//для наглядности выводим не более 5 записей

Print(comment, queryLINQ2.Take(5).ToList());

// Определение LINQ запроса 3

var queryLINQ3 = from t in db.Tanks

orderby t.TankId descending

select new

{

Название\_Емкости = t.TankType,

Материал\_Емкости = t.TankMaterial,

Объем\_Емкости = t.TankVolume,

Вес = t.TankWeight

};

comment = "3. Результат выполнения запроса на выборку записей из одной таблицы с выводом определенных полей: \r\n";

//для наглядности выводим не более 5 записей

Print(comment, queryLINQ3.Take(5).ToList());

}

4.3. Выполнение операций вставки данных:

static void Insert(toplivoContext db)

{

// Создать новую емкость

Tank tank = new Tank

{

TankType = "Бочка",

TankMaterial = "Дерево",

TankVolume = 30,

TankWeight = 100

};

// Создать новый вид топлива

Fuel fuel = new Fuel

{

FuelType = "Нитроглицерин",

FuelDensity = 3

};

// Добавить в DbSet

db.Tanks.Add(tank);

db.Fuels.Add(fuel);

// Сохранить изменения в базе данных

db.SaveChanges();

// Создать новую операцию

Operation operation = new Operation

{

TankId = tank.TankId,

FuelId = fuel.FuelId,

IncExp = 1000,

Date = DateTime.Now

};

// Добавить в DbSet

db.Operations.Add(operation);

// Сохранить изменения в базе данных

db.SaveChanges();

}

4.4. Выполнение операций удаления данных:

static void Delete(toplivoContext db)

{

//подлежащие удалению записи в таблице Tanks

string nametank = "Бочка1";

var tank = db.Tanks.Where(c => c.TankType == nametank);

//подлежащие удалению записи в таблице Fuels

string namefuel = "Нитроглицерин1";

var fuel = db.Fuels

.Where(c => c.FuelType == namefuel);

//подлежащие удалению записи в связанной таблице Operations

var someOperations = db.Operations

.Include("Tank")

.Include("Fuel")

.Where(o => ((o.Tank.TankType == nametank)) && (o.Fuel.FuelType == namefuel));

//Удаление нескольких записей в таблице Operations

db.Operations.RemoveRange(someOperations);

// сохранить изменения в базе данных

db.SaveChanges();

//Удаление нескольких записей в таблице Tanks и в таблице Fuels

db.Tanks.RemoveRange(tank);

db.Fuels.RemoveRange(fuel);

// сохранить изменения в базе данных

db.SaveChanges();

}

4.5. Выполнение операций обновления данных:

static void Update(toplivoContext db)

{

//подлежащие обновлению записи в таблице Tanks

string nametank = "Бочка";

var tank = db.Tanks.Where(c => c.TankType == nametank).FirstOrDefault();

//обновление

if (tank != null)

{

tank.TankType = "Бочка1";

tank.TankMaterial = "Дерево1";

};

//подлежащие обновлению записи в таблице Fuels

string namefuel = "Нитроглицерин";

var fuel = db.Fuels.Where(c => c.FuelType == namefuel).FirstOrDefault();

//обновление

if (fuel != null)

{

fuel.FuelType = "Нитроглицерин1";

};

//подлежащие обновлению записи в связанной таблице Operations

var someOperations = db.Operations.Include("Tank").Include("Fuel")

.Where(o => ((o.Tank.TankType == nametank)) && (o.Fuel.FuelType == namefuel));

//обновление

if (someOperations != null)

{

foreach (var op in someOperations)

{

op.Inc\_Exp = 0;

};

}

// сохранить изменения в базе данных

db.SaveChanges();

}

4.6. Вызов созданных методов в некотором порядке:

using System;

using System.Linq;

using Microsoft.EntityFrameworkCore;

using System.Collections;

namespace EFCore\_LINQ

{

public class Program

{

public static void Main(string[] args)

{

using (toplivoContext db = new toplivoContext())

{

//Выполняем разные методы, содержащие операции выборки и изменения данных

Console.WriteLine("====== Будет выполнена выборка данных (нажмите любую клавишу) ========");

Console.ReadKey();

Select(db);

Console.WriteLine("====== Будет выполнена вставка данных (нажмите любую клавишу) ========");

Console.ReadKey();

Insert(db);

Console.WriteLine("====== Выборка после вставки ========");

Select(db);

Console.WriteLine("====== Будет выполнено обновление данных (нажмите любую клавишу) ========");

Console.ReadKey();

Update(db);

Console.WriteLine("====== Выборка после обновления ========");

Select(db);

Console.WriteLine("====== Будет выполнено удаление данных (нажмите любую клавишу) ========");

Console.ReadKey();

Delete(db);

Console.WriteLine("====== Выборка после удаления ========");

Select(db);

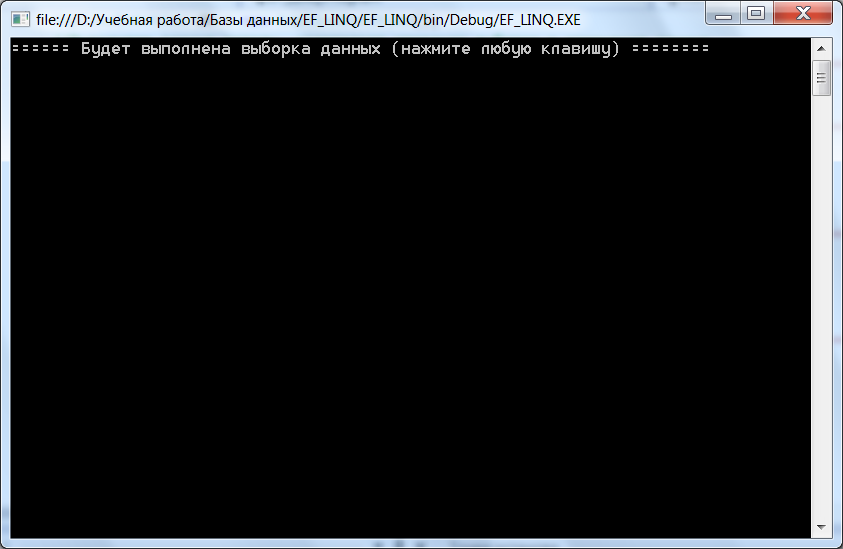
}

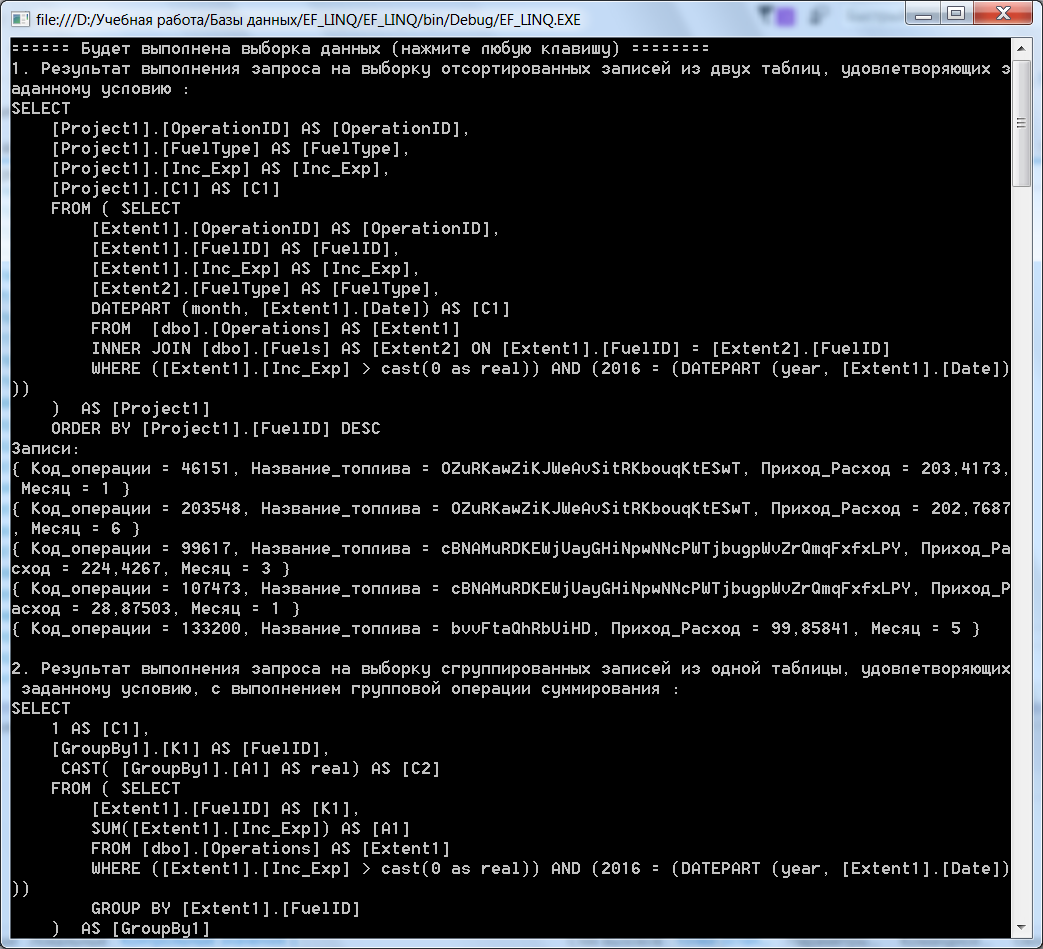
Console.Read();

}

}

5. Запускаем приложение и получаем результаты, периодически нажимая на клавиши для продолжения работы программы:





Например, в числе прочего, данный пример выбирает данные по операциям, произведенным в 2016 году, из таблицы Operations базы данных toplivo, с положительными значениями поля IncExp. LINQ запросы, которые это реализую приведены ниже:

// Определение LINQ запроса 1

var queryLINQ1 = from f in db.Operations

join t in db.Fuels

on f.FuelId equals t.FuelId

where (f.IncExp > 0 && f.Date.Year == 2016)

orderby f.FuelId descending

select new

{

Код\_операции = f.OperationId,

Название\_топлива = t.FuelType,

Приход\_Расход = f.IncExp,

Месяц = f.Date.Month

};

**3. ЗАДАНИЕ**

3.1. Создать, с использованием .NET Entity Framework Core консольное приложение, содержащее набор классов, моделирующих соответствующей своему варианту предметную область, на основе созданной в лабораторной работе № 1 базой данных MS SQL Server.

Для этого необходимо, используя подход Database First и средство Scaffolding (Reverse Engineering) Entity Framework Core, создать классы, моделирующие таблицы базы данных, и класс контекста данных.

3.2. Выполнить, используя объекты Entity Framework Core и LINQ:

1. Выборку всех данных из таблицы, стоящей в схеме базы данных нас стороне отношения «один» – 1 шт.
2. Выборку данных из таблицы, стоящей в схеме базы данных нас стороне отношения «один», отфильтрованные по определенному условию, налагающему ограничения на одно или несколько полей – 1 шт.
3. Выборку данных, сгруппированных по любому из полей данных с выводом какого-либо итогового результата (min, max, avg, сount или др.) по выбранному полю из таблицы, стоящей в схеме базы данных нас стороне отношения «многие» – 1 шт.
4. Выборку данных из двух полей двух таблиц, связанных между собой отношением «один-ко-многим» – 1 шт.
5. Выборку данных из двух таблиц, связанных между собой отношением «один-ко-многим» и отфильтрованным по некоторому условию, налагающему ограничения на значения одного или нескольких полей – 1 шт.
6. Вставку данных в таблицы, стоящей на стороне отношения «Один» – 1 шт.
7. Вставку данных в таблицы, стоящей на стороне отношения «Многие» – 1 шт.:
8. Удаление данных из таблицы, стоящей на стороне отношения «Один» – 1 шт.
9. Удаление данных из таблицы, стоящей на стороне отношения «Многие» – 1 шт.
10. Обновление удовлетворяющих определенному условию записей в любой из таблиц базы данных – 1 шт.

3.3. Разместить выполненный проект в репозитории на GitHub, создать README.md файл.

3.4. Используя средство GitHub Actions, написать рабочий процесс, который будет осуществлять компиляцию проекта под две разные платформы при любом изменении в репозитории.

Отредактировать README.md файл опубликованного проекта, вставив в него код для создания эмблемы состояния рабочего процесса (status badge), показывающей, чем в данный момент завершился рабочий процесс.

**4. ОФОРМЛЕНИЕ ОТЧЕТА**

Отчет должен содержать:

* Ссылку на GitHub репозиторий.
* Исходные коды классов моделей и класса контекста данных.
* Строку подключения к базе данных из конфигурационного json- файла.
* Исходные коды методов, реализующих требуемые по заданию LINQ запросы.
* Результаты выполнения LINQ запросов.
* Содержимое yml файла рабочего процесса.

**5. КОНТРОЛЬНЫЕ ВОПРОСЫ**

1. Общая характеристика фреймворка Entity Framework.
2. Общая характеристика объектов **DbContext,** D**bSet и DbQuery** Entity Framework.
3. Требования к свойствам и методам классам, соответствующих таблицам базы данных.
4. Класс контекста данных в Entity Framework, его методы, подключение к источнику данных.
5. Как и для чего делается миграция.
6. Как сформировать набор классов моделей и контекст данных для работы с существующей базой данных.
7. Реализация взаимодействия между классами моделей для отображения один-ко-многим в схеме базы данных.
8. Реализация взаимодействия между классами моделей для отображения один-к-одному в схеме базы данных.
9. Сопоставление типов данных полей таблиц СУБД MS SQL Server и свойств классов C#, моделирующих эти таблицы.
10. Класс DbSet, его свойства и методы.
11. Общая характеристика технологии LINQ и интерфейсов для работы с различными источниками данных.
12. Операция запроса в LINQ.
13. Способы записи LINQ запроса (синтаксисы записи). Выполнение запроса.
14. Запросы LINQ для осуществления фильтрации. Примеры для различных способов записи.
15. Запросы LINQ для осуществления сортировки и группировки. Примеры для различных способов записи.
16. Запросы LINQ для обращения к двум и более последовательностям. Соединение. Примеры для различных способов записи.
17. Правила построения выражений запросов в LINQ. Аналогии с SQL. Пример сложного выражения.
18. Изменение данных в таблице базы данных с использованием средств Entity Framework.
19. Вставка данных в таблицы базы данных с использованием средств Entity Framework.
20. Удаление данных из таблиц базы данных с использованием средств Entity Framework.
21. Обновление записей в таблицах базы данных с использованием средств Entity Framework.

**Литература для подготовки**

1. Руководство по Entity Framework Core - <https://metanit.com/sharp/efcore/> .
2. Руководство по Entity Framework Core от Microsoft <https://learn.microsoft.com/ru-ru/ef/core/>
3. Документация по GitHub Actions - <https://docs.github.com/ru/actions>