# Что такое Entity Framework Core

Entity Framework представляет ORM-технологиюот компании Microsoft для доступа к данным. Entity Framework Core позволяет абстрагироваться от самой базы данных и ее таблиц и работать с данными как с объектами классом 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.

# Код и пояснение

## Класс User

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public int Age { get; set; }

}

Это обычный класс, который содержит несколько свойств. Каждое свойство будет сопоставляться с отдельным столбцом в таблице из бд.

Надо отметить, что Entity Framework требует определения ключа элемента для создания первичного ключа в таблице в бд. По умолчанию при генерации бд EF в качестве первичных ключей будет рассматривать свойства с именами Id или [Имя\_класса]Id (то есть UserId).

## Класс ApplicationContext

{

public class ApplicationContext : DbContext// определяет контекст данных, используемый для взаимодействия с базой данных

{

public DbSet<User> Users => Set<User>();//представляет набор объектов, которые хранятся в базе данных

public ApplicationContext() => Database.EnsureCreated();

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)//устанавливает параметры подключения

{

//DbContextOptionsBuilder с помощью метода UseSqlite позволяет настроить строку подключения для соединения с базой данных SQLite.

optionsBuilder.UseSqlite("Data Source=helloapp.db");

}

}}

## Класс Program

using (ApplicationContext db = new ApplicationContext())

{

// создаем два объекта User

User artem = new User { Name = "Artem", Age = 18 };

User georgi = new User { Name = "Georgi", Age = 19 };

// добавляем их в бд

db.Users.Add(artem);

db.Users.Add(jaan);

db.SaveChanges();

Console.WriteLine("Объекты успешно сохранены");

// 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}");

}

**}**

## Результат

## 

# Подключение к существующей базе данных

## Класс helloappContext

public partial class helloappContext : DbContext

{

public helloappContext()

{

}

public helloappContext(DbContextOptions<helloappContext> options)

: base(options)

{

}

public virtual DbSet<User> Users { get; set; } = null!;

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

if (!optionsBuilder.IsConfigured)

{

optionsBuilder.UseSqlite("Data Source=C:\\Users\\opilane.TTHK\\source\\repos\\ArtemKabilov2\_TARpv20\\EntityFramework\\EntityFramework\\bin\\Debug\\net6.0\\helloapp.db");

}

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

OnModelCreatingPartial(modelBuilder);

}

partial void OnModelCreatingPartial(ModelBuilder modelBuilder);

}

}

## Переработанный класс Program

using (helloappContext db = new helloappContext())

{

// 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}");

}

}

## Результат

## 

# Управление базой данны

Для управления базой данной в классе DbContext определено свойство Database, которое представляет тип Microsoft.EntityFrameworkCore.Infrastructure.DatabaseFacade и которое предоставляет некоторый функционал для управления базой данных.

## Database.EnsureCreated

Метод Database.EnsureCreated() и его асинхронная версия Database.EnsureCreatedAsync() гарантируют, что база данных будет создана.

## Database.EnsureDeleted

Метод Database.EnsureDeleted() и его асинхронная версия Database.EnsureDeletedAsync() гарантируют, что база данных будет удалена.

## Database.EnsureDeleted

Метод Database.EnsureDeleted() и его асинхронная версия Database.EnsureDeletedAsync() гарантируют, что база данных будет удалена.

## Database.CanConnect

Еще один метод, который стоит отметить, это Database.CanConnect() и его асинхронная версия Database.CanConnectAsync().

# Основные операции с данными. CRUD

В файле **Program.cs** определим все базовые операции с данными:

using (ApplicationContext db = new ApplicationContext())

{

User artem = new User { Name = "Artem", Age = 18 };

User georgi = new User { Name = "Georgi", Age = 19 };

// Добавление

db.Users.Add(artem);

db.Users.Add(georgi);

db.SaveChanges();

}

// получение

using (ApplicationContext db = new ApplicationContext())

{

// получаем объекты из бд и выводим на консоль

var users = db.Users.ToList();

Console.WriteLine("\nAndmed pärast lisamist:");

foreach (User u in users)

{

Console.WriteLine($"{u.Id}.{u.Name} - {u.Age}");

}

}

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("\nAndmed pärast redigeerimist:");

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("\nandmed pärast kustutamist:");

var users = db.Users.ToList();

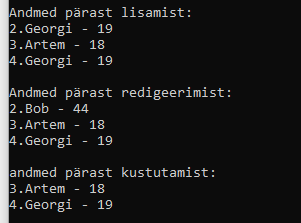
foreach (User u in users)

{

Console.WriteLine($"{u.Id}.{u.Name} - {u.Age}");

}

}



### Добавление

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

|  |  |
| --- | --- |
| 1  2 | db.Users.Add(artem);  db.SaveChanges(); |
|  |  |

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

|  |  |
| --- | --- |
| 1  2  3 | User tom = new User { Name = "Artem", Age = 18 };  User alice = new User { Name = "Georgi", Age = 19 };  db.Users.AddRange(artem,georgi); |

### Удаление

Удаление производится с помощью метода **Remove**:

|  |  |
| --- | --- |
| 1  2 | 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()**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | User? firstUser = db.Users.FirstOrDefault();  User? secondUser = db.Users.FirstOrDefault(u=>u.Id==2);  if (firstUser != null && secondUser != null)  {      db.Users.RemoveRange(firstUser, secondUser);      db.SaveChanges();  } |

### 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, которая определяет область действия объекта ApplicationContext. Но рассмотрим другой пример. Мы получаем объект в одном месте,а обновляем в другом. Например:

|  |  |
| --- | --- |
| 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 | 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. Потому что объект User больше не отслеживается контекстом данных. И в этом случае во второй конструкции using, где происходит редактирование, нам надо использовать метод Update:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | // 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()**:

|  |  |
| --- | --- |
| 1 | db.Users.UpdateRange(tom, alice); |

### Асинхронный API

Вместо метода **SaveChanges()** для асинхронного выполнения зароса к бд можно использовать его асинхронный двойник - **SaveChangesAsync()**. Также, для добавления данных определены асинхронные методы **AddAsync** и **AddRangeAsync**. Пример применения асинхронного API:

|  |  |
| --- | --- |
| 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 | using Microsoft.EntityFrameworkCore;    // для ToListAsync и FirstOrDefaultAsync  // Добавление  using (ApplicationContext db = new ApplicationContext())  {      User tom = new User { Name = "Tom", Age = 33 };      User alice = new User { Name = "Alice", Age = 26 };        // Добавление      await db.Users.AddRangeAsync(tom, alice);      await db.SaveChangesAsync();  }    // получение  using (ApplicationContext db = new ApplicationContext())  {      // получаем объекты из бд и выводим на консоль      var users = await db.Users.ToListAsync();      Console.WriteLine("Данные после добавления:");      foreach (User u in users)      {          Console.WriteLine($"{u.Id}.{u.Name} - {u.Age}");      }  }    // Редактирование  using (ApplicationContext db = new ApplicationContext())  {      // получаем первый объект      User? user = await db.Users.FirstOrDefaultAsync();      if (user != null)      {          user.Name = "Bob";          user.Age = 44;          //обновляем объект          await db.SaveChangesAsync();      }      // выводим данные после обновления      Console.WriteLine("\nДанные после редактирования:");      var users = await db.Users.ToListAsync();      foreach (User u in users)      {          Console.WriteLine($"{u.Id}.{u.Name} - {u.Age}");      }  }    // Удаление  using (ApplicationContext db = new ApplicationContext())  {      // получаем первый объект      User? user = await db.Users.FirstOrDefaultAsync();      if (user != null)      {          //удаляем объект          db.Users.Remove(user);          await db.SaveChangesAsync();      }      // выводим данные после обновления      Console.WriteLine("\nДанные после удаления:");      var users = await db.Users.ToListAsync();      foreach (User u in users)      {          Console.WriteLine($"{u.Id}.{u.Name} - {u.Age}");      }  } |

**Конфигурация подключения**

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

* Переопределение у класса контекста данных метода **OnConfiguring()**
* Передача конфигурации в конструктор базового класса **DbContext**

### Метод OnConfiguring

В предыдущих темах использовался первый метод. Например:

public class ApplicationContext : DbContext

{

    public DbSet<User> Users { get; set; } = null!;

    public ApplicationContext() => Database.EnsureCreated();

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

    {

        optionsBuilder.UseSqlite("Data Source=helloapp.db");

    }

}

В этот метод передается объект класса **DbContextOptionsBuilder**, который позволяет установить параметры подключения. Для их конфигурации параметров подключения у этого класса определено ряд методов в зависимости от того, какую именно систему баз данных мы собираемся использовать. Например, для установки подключения к SQLite вызывается метод UseSqlite(), в который передается строка подключения.

Это способ вполне рабочий и может использоваться. Единственно, что хочется отметить, что нам необязательно жестко определять строку подключения внутри контекста, мы можем получать ее извне:

public class ApplicationContext : DbContext

{

public DbSet<User> Users { get; set; } = null!;

public string connectionString;

public ApplicationContext(string connectionString)

{

this.connectionString = connectionString; // получаем извне строку подключения

Database.EnsureCreated();

}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseSqlite(connectionString);

}

}

Затем при создании объекта контекста передать строку подключения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | using (ApplicationContext db = new ApplicationContext("Data Source=helloapp.db"))  {      var users = db.Users.ToList();      Console.WriteLine("Пользователи:");      foreach (User user in users)      {          Console.WriteLine($"{user.Id}.{user.Name} - {user.Age}");      }  } |

### **Установка конфигурации в конструкторе**

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

public class ApplicationContext : DbContext

{

public DbSet<User> Users { get; set; } = null!;

public ApplicationContext(DbContextOptions<ApplicationContext> options)

: base(options)

{

Database.EnsureCreated();

}

}

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public int Age { get; set; }

}

using Microsoft.EntityFrameworkCore;

var optionsBuilder = new DbContextOptionsBuilder<ApplicationContext>();

var options = optionsBuilder.UseSqlite("Data Source=helloapp.db").Options;

using (ApplicationContext db = new ApplicationContext(options))

{

var users = db.Users.ToList();

foreach (User user in users)

Console.WriteLine($"{user.Id}.{user.Name} - {user.Age}");

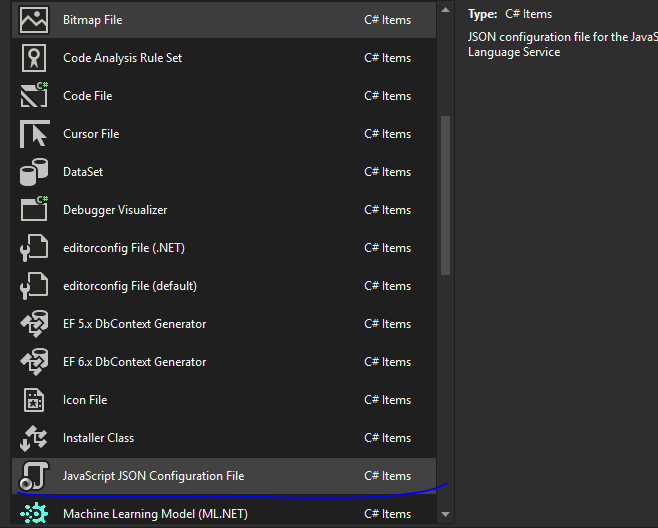
}

}

### Файл конфигурации

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 **JavaScript JSON Configuration File**, который назовем **appsettings.json**:



{

"ConnectionStrings": {

"DefaultConnection": "Data Source=helloapp.db"

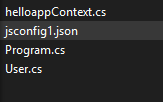
}

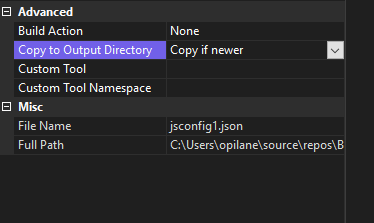
}

Здесь опять же определена наша строка подключения.

Чтобы ее использовать, нам надо добавить в проект через Nuget пакет: **Microsoft.Extensions.Configuration.Json**. Этот пакет специально предназначен для работы с конфигурацией в формате json.

После добавления файла в Visual Studio для его копирования в каталог приложения в окне свойств необходимо установить для опции **Copy to Output Directory** значение "Copy if newer" (или "Copy always")





Для работы с БД возьмем ранее определенный класс контекста:

public class ApplicationContext : DbContext

{

public DbSet<User> Users { get; set; } = null!;

public ApplicationContext(DbContextOptions<ApplicationContext> options)

: base(options)

{

Database.EnsureCreated();

}

}

Далее в файле **Program.cs** определим следующий код:

var builder = new ConfigurationBuilder();

// установка пути к текущему каталогу

builder.SetBasePath(Directory.GetCurrentDirectory());

// получаем конфигурацию из файла appsettings.json

builder.AddJsonFile("jsconfig1.json");

// создаем конфигурацию

var config = builder.Build();

// получаем строку подключения

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

var optionsBuilder = new DbContextOptionsBuilder<ApplicationContext>();

var options = optionsBuilder.UseSqlite(connectionString).Options;

using (ApplicationContext db = new ApplicationContext(options))

{

var users = db.Users.ToList();

foreach (User user in users)

Console.WriteLine($"{user.Id}.{user.Name} - {user.Age}");

}

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 jsconfig1.json.

В остальном работа с контекстом данных будет протекать также.

**Логгирование операций**

Для логгирования информации можно использовать метод **LogTo()**. Он применяется при конфигурации класса контекста данных.

Например, выведем всю информацию об операциях с базой данных на консоль. Допустим, у нас будут следующая модель:

public class User

{

public long Id { get; set; }

public string? Name { get; set; }

public long Age { get; set; }

}

И следующий класс контекста данных:

public class ApplicationContext : DbContext //определяет контекст данных, используемый для взаимодействия с базой данных

{

public DbSet<User> Users { get; set; } = null!;

public ApplicationContext()

{

Database.EnsureDeleted();

Database.EnsureCreated();

}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseSqlite("Data Source=helloapp.db");

optionsBuilder.LogTo(Console.WriteLine);

}

}

Для тестирования пусть у нас определена следующая программа:

using (ApplicationContext db = new ApplicationContext())

{

User user1 = new User { Name = "Artem", Age = 18 };

User user2 = new User { Name = "Georgi", Age = 19 };

db.Users.Add(user1);

db.Users.Add(user2);

db.SaveChanges();

var users = db.Users.ToList();

Console.WriteLine("Список пользователей:");

foreach (User u in users)

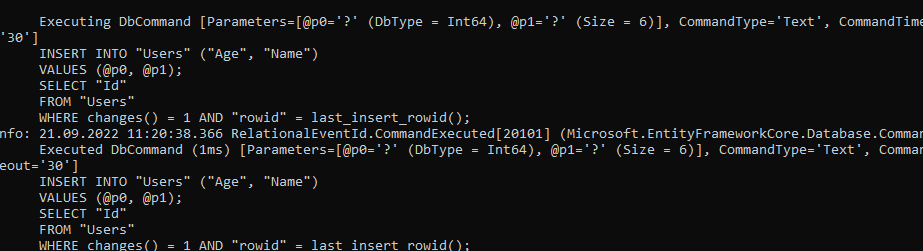
{

Console.WriteLine($"{u.Id}.{u.Name} - {u.Age}");

}

}

И при запуске приложения на консоль будет выведена детальная информация по всем операциям:



Подобным образом можно логгировать в другие места. Например, логгирование в окно Output, что производится с помощью метода **Debug.WriteLine()**:

optionsBuilder.LogTo(message => System.Diagnostics.Debug.WriteLine(message));

Другим распространенным способом логгирования является вывод в файл:

public class ApplicationContext : DbContext

{

readonly StreamWriter logStream = new StreamWriter("mylog.txt", true);

public DbSet<User> Users { get; set; } = null!;

public ApplicationContext()

{

Database.EnsureDeleted();

Database.EnsureCreated();

}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseSqlite("Data Source=helloapp.db");

optionsBuilder.LogTo(logStream.WriteLine);

}

public override void Dispose()

{

base.Dispose();

logStream.Dispose();

}

public override async ValueTask DisposeAsync()

{

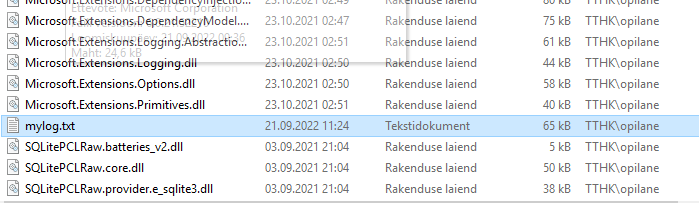
await base.DisposeAsync();

await logStream.DisposeAsync();

}

}

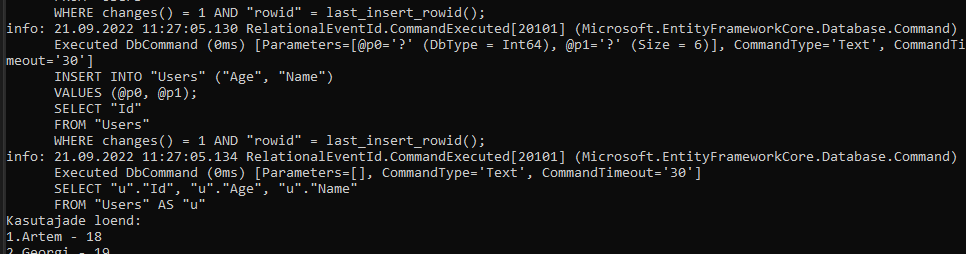
Собственно для записи в файл используется объект класса StreamWriter из пространства имен System.IO. Его метод logStream.WriteLine, который пищет в файл строку, передается в метод LogTo(). Для закрытия и утилизации файлового потока StreamWriter в классе контекста переопределены методы Dispose/DisposeAsync, в которых вызывается метод Dispose/DisposeAsync объекта StreamWriter. В итоге при выполнении программы в папке приложения появится файл лога mylog.txt.



По умолчанию EntityFramework Core использует значение Debug, но можно указать какое-нибудь другое значение:

|  |  |
| --- | --- |
| 1 | optionsBuilder.LogTo(Console.WriteLine, LogLevel.Information); |

оскольку каждый класс идентификатора имеет довольно много полей, которые представляют опеделенное сообщение, я не буду подробно расписывать все эти поля. Посмотрим на простом примере, как мы можем конкретизировать сообщения - например, нам надо вывести только выполняемые команды SQL. В этом случае мы можем воспользоваться RelationalEventId и его переменной CommandExecuted, которая представляет окончание выполнения команды:



#### Категории сообщений

Другим способом фильтрации сообщений представляет использование категорий, которые представлены классом **DbLoggerCategory** и который позволяет задать нужные категории логгирования:

Например, выведем в лог информацию только об исполняемых командах:

|  |
| --- |
| optionsBuilder.LogTo(Console.WriteLine, new[] { DbLoggerCategory.Database.Command.Name }); |

**Управление схемой БД и миграции**

Если мы меняем модели в Entity Framework, которые входят в контекст данных, например, добавляем в нее какие-то новые свойства или удаляем некоторые свойства, то необходимо, чтобы база данных также применяла эти изменения. Например, в прошлых темах был создан класс User, который описывал пользователя:

public class User

{

public long Id { get; set; }

public string? Name { get; set; }

public long Age { get; set; }

}

А для работы с базой данных использовался следующий контекст данных:

public class ApplicationContext : DbContext

{

public DbSet<User> Users { get; set; } = null!;

public ApplicationContext()

{

Database.EnsureCreated();

}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseSqlite("Data Source=helloapp.db");

}

}

Допустим, мы хотим добавить в класс User новое свойство, например:

public string? Position { get; set; }

И если у нас уже ранее была создана база данных, на которую указывает строка подключения в классе контекста, и мы попытаемся выполнить какие-нибудь операции с моделью User, например, получить данные этой модели

using (ApplicationContext db = new ApplicationContext())

{

var users = db.Users.ToList();

Console.WriteLine("Kasutajade loend:");

foreach (User u in users)

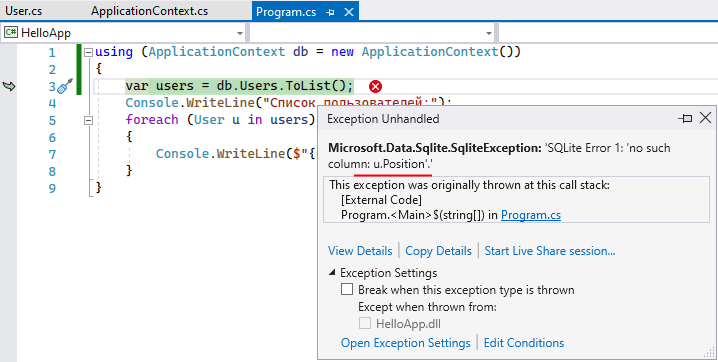
{

Console.WriteLine($"{u.Id}.{u.Name} - {u.Age}");

}

}

то мы столкнемся с ошибкой



**Ручное изменение базы данных**

В самых простых случаях мы можем написать sql-скрипт для добавления столбцов или таблиц, либо же даже можем изменить таблицы вручную с помощью различных программ, которые позволяют в режиме дизайнера редактировать таблицы.

Например, в примере выше применялась база данных SQLite. Для ее редактирования мы можем использовать программу [DB Browser for SQLite](https://sqlitebrowser.org/). Так, откроем базу данных в этой программе. Нажмем на таблицу Users правой кнопкой мыши и в появившемся контекстном меню выберем Modify Table: