Hello App

Entity Framework представляет ORM-технологию (object-relational mapping - отображения данных на реальные объекты) от 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.

https://metanit.com/sharp/efcore/1.5.php

Итак, нам надо определить модель, которая будет описывать данные. Пусть наше приложение будет посвящено работе с пользователями. Поэтому добавим в проект новый класс ***User***:

**Код и обьяснение:**

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public int Age { get; set; }

public string? Gender { get; set; }

}

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

Взаимодействие с базой данных в Entity Framework Core происходит посредством специального класса - контекста данных. Поэтому добавим в наш проект новый класс, который назовем ApplicationContext и который будет иметь следующий код:

namespace HelloApp

{

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

{

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

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

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

{

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

}

}

}

Теперь определим сам код программы, который будет взаимодействовать с созданной БД. Для этого изменим файл **Program.cs** следующим образом:

using HelloApp;

using (ApplicationContext db = new ApplicationContext())

{

// создаем два объекта User

User tom = new User { Name = "Tom", Age = 33, Gender = "Male" };

User alice = new User { Name = "Alice", Age = 26, Gender = "Female" };

// добавляем их в бд

db.Users.Add(tom);

db.Users.Add(alice);

db.SaveChanges();

Console.WriteLine("Elements were successfuly saved");

// получаем объекты из бд и выводим на консоль

var users = db.Users.ToList();

Console.WriteLine("Elements list:");

foreach (User u in users)

{

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

}

}

Для примера проект консольного приложения на C# и для работы с бд SQLite через Entity Framework добавим в него NuGet-пакет **Microsoft.EntityFrameworkCore.Sqlite**.

В классе User всё тоже самое что и в предыдущих примерах.

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

using (ApplicationContext db = new ApplicationContext())

{

User tom = new User { Name = "Tom", Age = 33 };

User alice = new User { Name = "Alice", Age = 26 };

// Добавление

db.Users.Add(tom);

db.Users.Add(alice);

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();

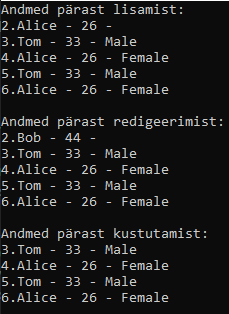
foreach (User u in users)

{

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

}

}



**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? firstUser = db.Users.FirstOrDefault();

User? secondUser = db.Users.FirstOrDefault(u => u.Id == 2);

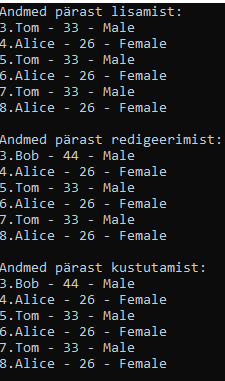
if (firstUser != null && secondUser != null)

{

db.Users.RemoveRange(firstUser, secondUser);

db.SaveChanges();

}



Данные под индексом 2 были удалены.

### 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:

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}");

}

}

Вывод в консоль с помощью API идентичен обыкновенному способу выведения данных из базы.

## Конфигурация подключения

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

{

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:

using (ApplicationContext db = new ApplicationContext("Data Source=helloapp.db"))

{

var users = db.Users.ToList();

Console.WriteLine("Пользователи:");

foreach (User user in users)

{

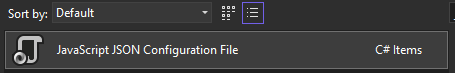
Console.WriteLine($"{user.Id}.{user.Name} - {user.Age}");

}

}

### Файл конфигурации

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



Определим в этом файле следующий код:

{

"ConnectionStrings": {

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

}

}

Для работы с БД возьмем ранее определенный класс контекста:

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

{

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

public ApplicationContext(DbContextOptions<ApplicationContext> options)

: base(options)

{

Database.EnsureCreated();

}

}

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

using Microsoft.EntityFrameworkCore;

using Microsoft.Extensions.Configuration;

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.UseSqlite(connectionString).Options;

using (ApplicationContext db = new ApplicationContext(options))

{

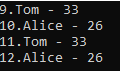
var users = db.Users.ToList();

foreach (User user in users)

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

}

Вывод в консоль:



### Метод LogTo

И следующий класс контекста 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);

}

В методе OnConfiguring() у передаваемого в качестве параметра объекта DbContextOptionsBuilder вызывается метод **LogTo()**, в который передается делегат Action<string> - то есть некоторое действие, которое принимает один параметр типа string и и ничего не возвращает. Именно такое действие представляет традиционный метод Console.WriteLine(), который выводит строку на консоль.

Для тестирования пусть у нас определена следующая программа:

using (ApplicationContext db = new ApplicationContext())

{

User user1 = new User { Name = "Tom", Age = 33 };

User user2 = new User { Name = "Alice", Age = 26 };

db.Users.Add(user1);

db.Users.Add(user2);

db.SaveChanges();

var users = db.Users.ToList();

Console.WriteLine("Список пользователей:");

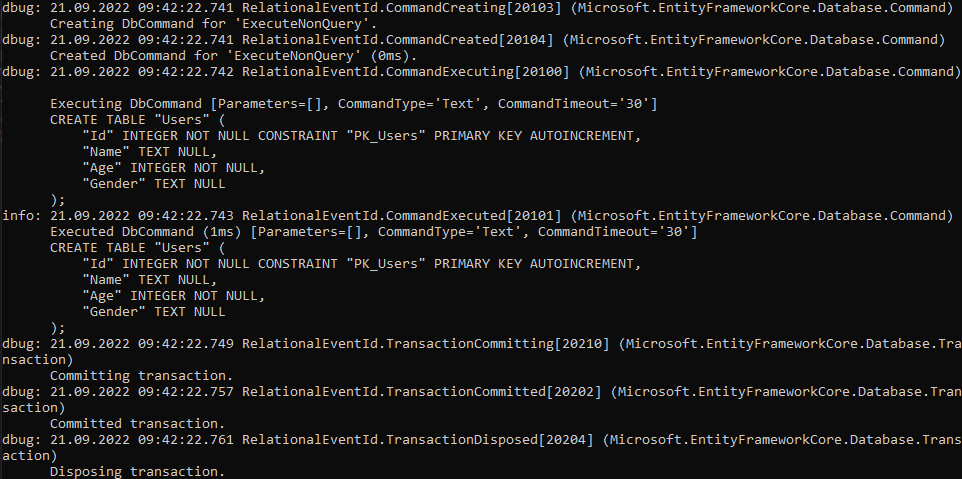
foreach (User u in users)

{

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

}

}



## Управление схемой БД и миграции

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

public string? Position { get; set; } // Новое свойство - должность пользователя

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("Список пользователей:");

foreach (User u in users)

{

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

}

}

### Database.EnsureCreated и Database.EnsureDeleted

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

public ApplicationContext()

{

Database.EnsureDeleted(); // удаляем бд со старой схемой

Database.EnsureCreated(); // создаем бд с новой схемой

}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

}

Для использования миграций в Visual Stuido необходимо добавить в проект через менеджер Nuget пакет **Microsoft.EntityFrameworkCore.Tools**.

Например, определим модели и контекст следующим образом:

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

public ApplicationContext()

{

// Database.EnsureCreated();

}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

}

Обратите внимание, что в конструкторе контекста закомментирован метод Database.EnsureCreated(). В данном случае он не нужен. Более того при выполнении миграции этот метод вызывает ошибку. Этот момент следует учитывать.

Также стоит отметить, что при самом первом применении миграции по отношению к БД SQLite Entity Framework пытается создать ее заново, однако если создаваемые таблицы в ней уже есть, то мы столкнемся с ошибкой. Поэтому следует убедиться, что по используемому пути нет файла базы данных с подобным именем. При последующих применениях миграции EF будет использовать бд, созданную при первой миграции.

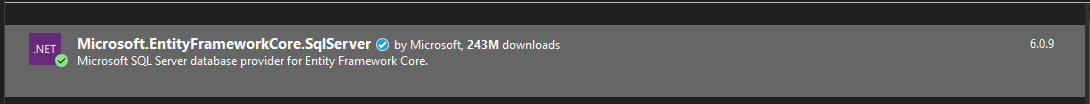
Теперь для создания и выполнения миграции перейдем в Visual Studio к окну **Package Manager Console**. Вначале введем команду:

Add-Migration InitialCreate

# Провайдеры баз данных

## MS SQL Server

Для работы с базой данных MS SQL Server через Entity Framework Core в проект необходимо добавить Nuget-пакет **Microsoft.EntityFrameworkCore.SqlServer**:



После установки пакета также определим класс контекста данных:

public class ApplicationContext : DbContext

{

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

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

}

}

Для подключения к SQL Server у класса **DbContextOptionsBuilder** определен метод расширения **UseSqlServer**, в который передается строка подключения для соединения с MS SQL Server. 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 MS SQL Server Express этот параметр, как правило, имеет значение .\SQLEXPRESS
* **Database**: название базы данных
* **Trusted\_Connection**: устанавливает проверку подлинности

В данном случае мы определяем, что в качестве сервера будет использоваться движок **localdb**, который предназначен специально для разработки:("Server=(localdb)\mssqllocaldb"), а база данных будет называться helloappdb ("Database=helloappdb").

Подробно про элементы строки подключения к MS SQL Server можно найти здесь: [Строка подключения для MS SQL Server](https://metanit.com/sharp/adonetcore/2.1.php)

Теперь определим в файле **Program.cs** простейшую программу по добавлению и извлечению объектов из базы данных:

using HelloApp;

// добавление данных

using (ApplicationContext db = new ApplicationContext())

{

db.Database.EnsureDeleted();

db.Database.EnsureCreated();

// создаем два объекта User

User user1 = new User { Name = "Tom", Age = 33 };

User user2 = new User { Name = "Alice", Age = 26 };

// добавляем их в бд

db.Users.AddRange(user1, user2);

db.SaveChanges();

}

// получение данных

using (ApplicationContext db = new ApplicationContext())

{

// получаем объекты из бд и выводим на консоль

var users = db.Users.ToList();

Console.WriteLine("Users list:");

foreach (User u in users)

{

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

}

}