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();

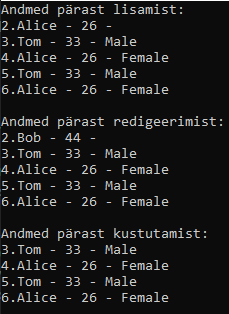
foreach (User u in users)

{

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);

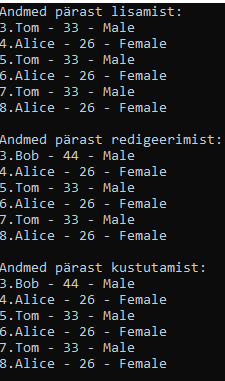
if (firstUser != null && secondUser != null)

{

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)

{

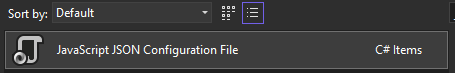
Console.WriteLine($"{user.Id}.{user.Name} - {user.Age}");

}

}

### 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))

{

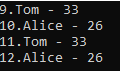
var users = db.Users.ToList();

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:");

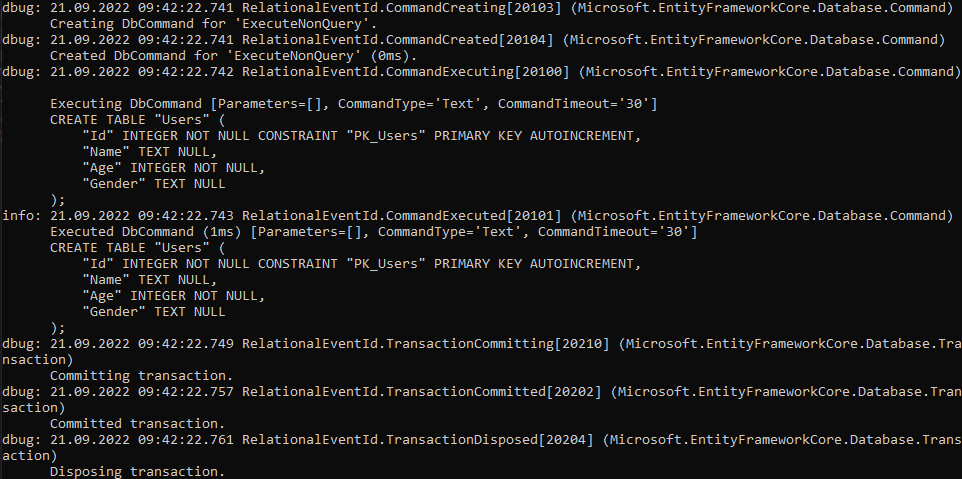
foreach (User u in users)

{

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.

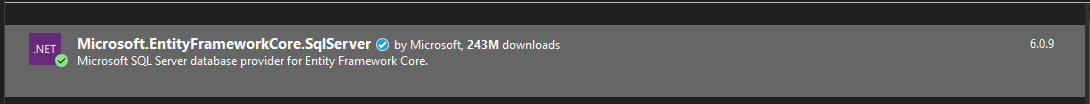
Теперь для создания и выполнения миграции перейдем в Visual Studio к окну **Package Manager Console**. Вначале введем команду:

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:");

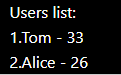
foreach (User u in users)

{

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

}

}



## MySQL

Определим контекст данных - класс ApplicationContext:

public class ApplicationContext : DbContext

{

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

public ApplicationContext()

{

Database.EnsureCreated();

}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseMySql("server=localhost;user=root;password=123456789;database=usersdb;",

new MySqlServerVersion(new Version(8, 0, 25)));

}

}

Для работы с MySQL вызывается метод **UseMySql()**, в который передается строка подключения. В строке подключения указываются адрес сервера (параметр server), имя пользователя в субд (User), его пароль (Password) и имя базы данных (Database).

В качестве второго параметра передается номер версии MySQL в виде объекта **MySqlServerVersion** - в его конструктор передается объект Version, который собственно содержит номер установленной версии MySQL. Например, в моем случае это версия **8.0.25**, соответственно я передаю объект new MySqlServerVersion(new Version(8, 0, 25)). Версию MySQL можно узнать, например, через MySQL Workbench

И для тестирования определим в файле **Program.cs** добавление и вывод данных:

using HelloApp;

using (ApplicationContext db = new ApplicationContext())

{

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

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

db.Users.AddRange(user1, user2);

db.SaveChanges();

}

// получение данных

using (ApplicationContext db = new ApplicationContext())

{

var users = db.Users.ToList();

Console.WriteLine("Список объектов:");

foreach (User u in users)

{

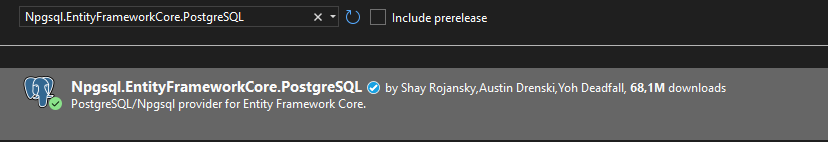
Console.WriteLine($"{u.Id}.{u.Name} - {u.Age}");

}

}

## PostgreSQL

Для работы с базой данных PostgreSQL в проект необходимо добавить через Nuget пакет **Npgsql.EntityFrameworkCore.PostgreSQL**:



Определим класс контекста данных:

using HelloApp;

using Microsoft.EntityFrameworkCore;

public class ApplicationContext : DbContext

{

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

public ApplicationContext()

{

Database.EnsureCreated();

}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseNpgsql("Host=localhost;Port=5432;Database=usersdb;Username=postgres;Password=пароль\_от\_postgres");

}

}

Для установки подключения к базе данных в методе OnConfiguring вызывается метод **UseNpgsql()**, в который передается строка подключения. Строка подключения содержит адрес сервера (параметр Host), порт (Port), название базы данных на сервере (Database), имя пользователя в рамках сервера PostgreSQL (Username) и его пароль (Password). В зависимости от настроек сервера PostgreSQL параметры могут отличаться.

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

// добавление данных

using HelloApp;

using (ApplicationContext db = new ApplicationContext())

{

// 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}");

}

}



### Миграции

Выше для создания базы данных использовался метод Database.EnsureCreated. Теперь изменим класс контекста данных - уберем вызов Database.EnsureCreated и изменим название база данных:

public class ApplicationContext : DbContext

{

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

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseNpgsql("Host=localhost;Port=5432;Database=usersdb2;Username=postgres;Password=123456789");

}

}

Посмотрим теперь, как использовать миграции. Прежде всего нам надо добавить в проект через Nuget пакет **Microsoft.EntityFrameworkCore.Tools** для поддержки миграций.

Для создания базы данных создадим и выполним миграции. Для этого в окне **Package Manager Console** введем команду:

Add-Migration Initial

После генерации файла миграции для создания базы данных выполним команду:

Update-Database

**Подключение к существующей базе данных**

Для подключения к существующей базе данных в PostgreSQL необходимо в окне **Package Manager Console** выполнить команду Scaffold-DbContext, которой передается строка подключения и название провайдера, то есть **Npgsql.EntityFrameworkCore.PostgreSQL** (для выполнения этой команды тоже необходим пакет **Microsoft.EntityFrameworkCore.Tools**). Сначала вводится команда Scaffold-DbContext и строка подключения:

|  |  |
| --- | --- |
|  | Scaffold-DbContext "Host=localhost;Port=5432;Database=usersdb;Username=postgres;Password=123456789" |

Затем в консоли появится слово **Provider**, после которого надо будет ввести название провайдера, то есть

|  |  |
| --- | --- |
|  | Npgsql.EntityFrameworkCore.PostgreSQL |

# Создание модели в Entity Framework Core

## Fluent API и аннотации данных

**Модель** в Entity Framework представляет набор всех сущностей и связей между ними, которыми управляет контекст данных. Все сущности, с которыми работает Entity Framework Core и которые хранятся в базе данных, определяются в C# в виде классов. При этом Entity Framework применяет ряд условностей для сопоставления классов с таблицами. Например, названия столбцов должны соответствовать названиям свойств и т.д. В этом случае Entity Framework сможет сопоставить столбцы таблицы и свойства классов.

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

### Fluent API

**Fluent API** представляет набор методов, которые определяют сопоставление между классами и их свойствами и таблицами и их столбцами. Для использования функционала Fluent API переопределяется метод **OnModelCreating()**:

public class ApplicationContext : DbContext

{

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");

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

// использование Fluent API

base.OnModelCreating(modelBuilder);

}

}

### Аннотации

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

using System.ComponentModel.DataAnnotations.Schema;

public class User

{

[Column("user\_id")]

public int Id { get; set; }

public string? Name { get; set; }

public int Age { get; set; }

public string Gender { get; set; }

}

## Определение моделей

### Включение сущностей в модель

По умолчанию все типы сущностей, для которых определены в контексте данных наборы **DbSet**, включаются в модель и в дальнейшем сопоставляются с таблицами в базе данных. Например:

public class ApplicationContext : DbContext

{

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");

}

}

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public int Age { get; set; }

public string Gender { get; set; }

}

### Включение сущностей в модель без DbSet

Но кроме того, в модель также включаются типы, на которые есть ссылки в сущностях, которые уже включены в модель, например, через свойства DbSet.

Например, пусть у нас определены следующие сущности:

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public int Age { get; set; }

public string Gender { get; set; }

// навигационное свойство

public Company? Company { get; set; }

}

public class Company

{

public int Id { get; set; }

public string? Name { get; set; }

}

public class Country

{

public int Id { get; set; }

public string? Name { get; set; }

}

И пусть у нас будет класс контекста данных:

using Microsoft.EntityFrameworkCore;

public class ApplicationContext : DbContext

{

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

public ApplicationContext()

{

Database.EnsureDeleted();

Database.EnsureCreated();

}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

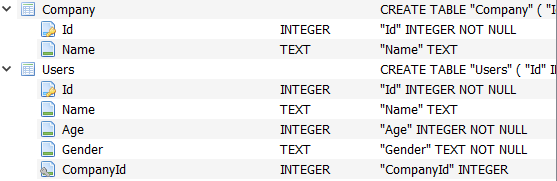
{

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

}

}

После создания базы данных в ней будут созданы две таблицы: Users и Company. А третий класс - Country никак не используется в сущностях User и Company, для Country нет свойства DbSet в классе контекста, поэтому она не будет включена в контекст и для нее не будет создаваться таблица в бд.



Поскольку для типа User определен набор DbSet, то для имени таблицы будет применяться имя этого набора, а для второй таблицы будет использоваться имя класса Company.

Еще один способ включения сущности в модель представляет вызов **Entity()** объекта **ModelBuilder** в методе **OnModelCreating()**:

using Microsoft.EntityFrameworkCore;

public class ApplicationContext : DbContext

{

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");

}

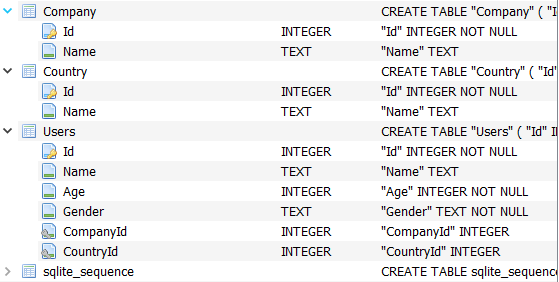
protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Country>();

}

}



## Свойства сущности

Класс User имеет четыре публичных свойства, поэтому при чтении или записи в базу данных Entity Framework будет автоматически сопоставлять столбцы из таблицы с этими свойствами по имени. Но такое поведение не всегда необходимо. Иногда требуется, наоборот, исключить определенное свойство, чтобы для него не создавался столбец в таблице.

Исключение с помощью Fluent API производится через метод **Ignore()**:

using Microsoft.EntityFrameworkCore;

public class ApplicationContext : DbContext

{

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");

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<User>().Ignore(u => u.Address);

}

}

Здесь свойство Address исключается из модели, и для него не будет создаваться столбец в таблице Users.

Исключение с помощью аннотаций данных:

using System.ComponentModel.DataAnnotations.Schema;

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

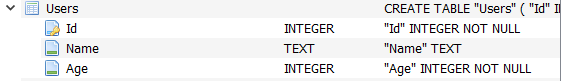
public int Age { get; set; }

[NotMapped]

public string? Address { get; set; }

}

В обоих случаях при миграции будет создана таблица Users, которая не будет содержать столбца для свойства Address, и оно не будет участвовать в сопоставлениях при операциях с бд:



### Использование полей класса

В примере выше применялись автосвойства, которые представляют сокращенную версию свойств без полноценных блоков get и set. Однако свойства не обязательно должны представлять именно автосвойства. Для хранения значений они могут использовать поля класса и иметь полноценные блоки get и set. Например:

public class User

{

string name;

public int Id { get; set; }

public string Name

{

get { return name; }

set { name = value; }

}

public int Age { get; set; }

public string Gender { get; set; }

}

## Конструкторы сущностей

Когда EF Core создает объект сущности, например, при после получения данных из БД, он вначале вызывае конструктор по умолчанию, который не имеет параметров, и затем передает каждому свойству полученные из бд значения.

Если EF Core находит конструктор с параметрами, где названия и типы параметров соответствуют устанавливаемым свойствам, то вместо установки свойств EF передает полученные из БД значения параметрам конструктора. При этом между параметрами и свойствами должно быть соответствие по типу и имени за тем исключением, что названия могут отличаться по регистру, например, свойство Name и параметр name. Рассмотрим на примере. Допустим, у нас есть следующая сущность User:

public class User

{

public int Id { get; set; }

public string Name { get; set; }

public int Age { get; set; }

public User(string name, int age)

{

Name = name;

Age = age;

Console.WriteLine($"Вызов конструктора для объекта {name}");

}

}

Класс User имеет три свойства и через конструктор устанавливает два из них.

Пусть у нас будет стандартный контекст данных:

using Microsoft.EntityFrameworkCore;

public class ApplicationContext : DbContext

{

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

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

}

}

И, допустим, в программе создаем несколько объектов User, добавляем их в БД и получаем обратно из БД:

using (ApplicationContext db = new ApplicationContext())

{

db.Database.EnsureDeleted();

db.Database.EnsureCreated();

User tom = new User("Tom", 37);

User bob = new User("Bob", 41);

db.Users.Add(tom);

db.Users.Add(bob);

db.SaveChanges();

}

using (ApplicationContext db = new ApplicationContext())

{

Console.WriteLine("Andmete hankimine andmebaasist:");

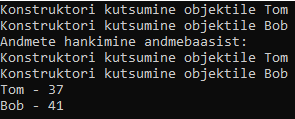
var users = db.Users.ToList();

foreach (var user in users)

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

}

Здесь при получении данных при выполнении метода db.Users.ToList() EF Core будет вызывать для каждой полученной строки из таблицы объект User, вызывая его конструктор с двумя параметрами. Для наглядности в примере выше разделы операции добавления и получения по разным объектам контекста. В итоге мы получим следующий консольный вывод:



Здесь надо учитывать несколько моментов:

* Необязательно для всех свойств определять в конструкторе свои параметры. Например, свойство Id не устанавливается в конструкторе. Те свойства, для которых в конструкторе не определено параметров, устанавливаются напрямую, как в общем случае.
* Параметры и свойства должны соответствовать по имени и типу за исключением регистра имени.
* Конструкторы могут иметь любой модификатор доступа, в том числе, private.
* EF Core НЕ устанавливает таким обазом навигационные свойства, которые представляют другие сущности и имеют конструктор:

public class User

{

public int Id { get; set; }

public string Name { get; set; }

public int Age { get; set; }

public Company? Company { get; set; }

public User(string name, int age)

{

Name = name;

Age = age;

}

}

public class Company

{

public int Id { get; set; }

public string Name { get; set; }

public ICollection<User> Users { get; set; } = new List<User>();

public Company(string name) => Name = name;

}

При этом класс может определять несколько конструкторов с разным количеством параметров:

public class User

{

public int Id { get; set; }

public string Name { get; set; }

public int Age { get; set; }

public User(string name)

{

Name = name;

Age = 18;

}

public User(string name, int age)

{

Name = name;

Age = age;

}

}

## Сопоставление таблиц и столбцов

### Сопоставление таблиц

Каждая сущность по умолчанию сопоставляется с таблицей, которая называется по имени свойства **DbSet<T>** в контексте данных, представляющего данную сущность. Если в контексте данных подобного свойства не определено, то для названия таблицы используется имя класса сущности.

#### Атрибут Table

Атрибут Table позволяет переопределить сопоставление с таблицей по имени:

[Table("People")]

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

}

Теперь сущность User будет сопоставляться с таблицей "People".

#### Метод ToTable

Аналогичное переопределение можно произвести через Fluent API с помощью метода **ToTable()**:

public class ApplicationContext : DbContext

{

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");

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<User>().ToTable("People");

}

}

### Сопоставление столбцов

По умолчанию каждое свойство сопоставляется с одноименным столбцом.

#### Атрибут Column

Атрибут **Column** переопределяет сопоставление:

public class User

{

[Column("user\_id")]

public int Id { get; set; }

public string? Name { get; set; }

public string? Gender { get; set; }

}

Теперь свойство Id будет сопоставляться со столбцом "user\_id".

#### Метод HasColumnName

Также сопоставление можно переопределить в Fluent API с помощью метода **HasColumnName**:

using Microsoft.EntityFrameworkCore;

public class ApplicationContext : DbContext

{

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");

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<User>().Property(u => u.Id).HasColumnName("user\_id");

}

}

## Обязательные и необязательные свойства

### Обязательные свойства

По умолчанию свойство является необязательным к установке, если оно допускает значение **null**. Это свойства, которые представляют nullable-типы, например, **string?**, **int?** и т.д. Хотя мы также можем настроить эти свойства как обязательные.

Свойство является обязательным, если оно не допускает значение **null**.

Например, возьмем следующую модель:

public class User

{

public int Id { get; set; }

public string Name { get; set; } = "";

public string? Company { get; set; }

public int Age { get; set; }

public string Gender { get; set; }

}