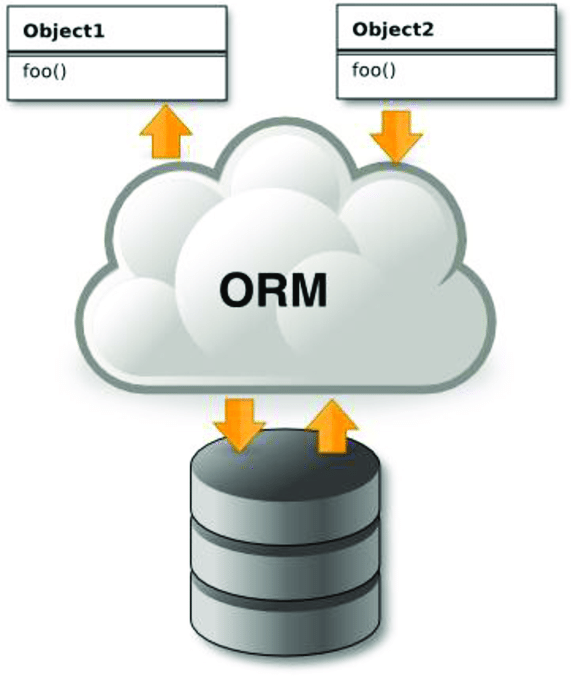
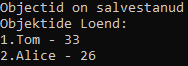
# Глава 1. Введение в Entity Framework Core

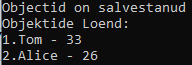


ORM-технология (object-relational mapping - отображения данных на реальные объекты) от компании Microsoft для доступа к данным. Entity Framework Core позволяет абстрагироваться от самой базы данных и ее таблиц и работать с данными как с объектами классом независимо от типа хранилища. Если на физическом уровне мы оперируем таблицами, индексами, первичными и внешними ключами, но на концептуальном уровне, который нам предлагает Entity Framework, мы уже работаем с объектами.

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 выполнение запроса транслирует выражения LINQ в выражения, понятные для конкретной СУБД (как правило, в выражения SQL).

## Первое приложение на EF Core

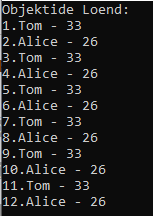
Оригинал (public DbSet<User> Users => Set<User>();) Вывод:

Альтернатива (public DbSet<User> Users { get; set; } = null!;) Вывод:

## Подключение к существующей базе данных

Scaffold-DbContext "Data Source=D:\\helloapp.db" Microsoft.EntityFrameworkCore.Sqlite

При выполнения этой команды в проект будет добавлены класс User,класс контекста данных, который будет называться по имени базы данных плюс суффикс "Context"

Вывод нового проекта DBFirstApp:

## Управление базой данных

### Database.EnsureCreated

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

Если база данных отсутствует, то данный метод создает ее.

Если база данных имеется, но она НЕ имеет таблиц, то этот метод создает таблицы, которые соответствуют схеме данных.

Если база данных имеется и она имеет таблицы, то этот метод не оказывает никакого влияния.

### Database.EnsureDeleted

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

Если база данных имеется, то она удаляется. Если база данных отсутствует, то этот метод не оказывает никакого влияния.

### Database.CanConnect

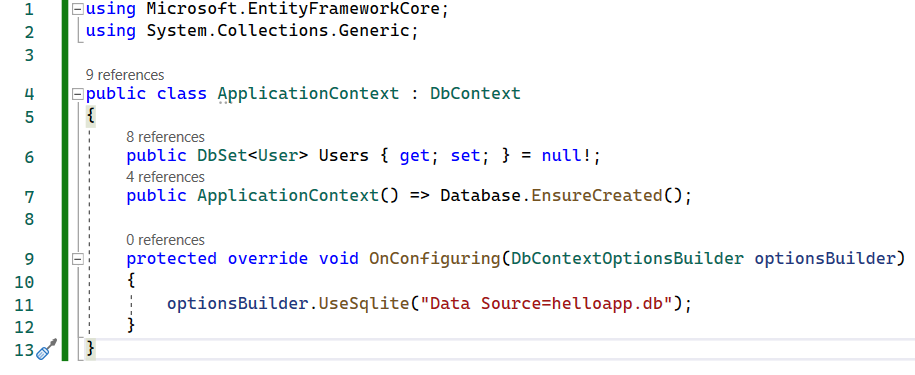
Еще один метод, который стоит отметить, это Database.CanConnect() и его асинхронная версия Database.CanConnectAsync(). Данный метод возвращает true, если бд доступна, и false, если бд не доступна:

## Основные операции с данными. CRUD

#### Класс User:

## 

#### Класс ApplicationContext:



#### Класс Program:

// Добавление

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("Andmed pärast lisamist:");

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("\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}");

}

## }

## Вывод исполненного кода:

### Удаление

Удаление производится с помощью метода Remove:

db.Users.Remove(user);

db.SaveChanges();

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();

}

### 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.

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

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:

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("Andmed pärast lisamist:");

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("\nAndmed pärast redigeerimist:");

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("\nAndmed pärast kustutamist:");

var users = await db.Users.ToListAsync();

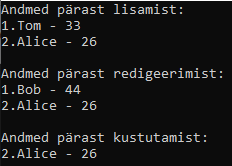
foreach (User u in users)

{

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

}

}

Вывод кода с использованием Асинхронного API:

## 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

using Microsoft.EntityFrameworkCore;

using System.Collections.Generic;

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.

### 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:

using Microsoft.EntityFrameworkCore;

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; }

}

Тогда мы могли бы использовать класс контекста следующим образом:

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}");

}

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

### 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

#### ApplicationContext:

using Microsoft.EntityFrameworkCore;

public class ApplicationContext : DbContext

{

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

public ApplicationContext(DbContextOptions<ApplicationContext> options)

: base(options)

{

Database.EnsureCreated();

}

}

#### Program.cs:

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}");

}

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");

## Логгирование операций

Логгирование позволяет нам получить информацию о выполняемых в Entity Framework операциях. Причем использовать как встроенные возможности, так и создать и встроить свою инфраструктуру логгирования.

### Метод LogTo

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

#### User.cs

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public int Age { get; set; }

}

#### ApplicationContext.cs

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)

{

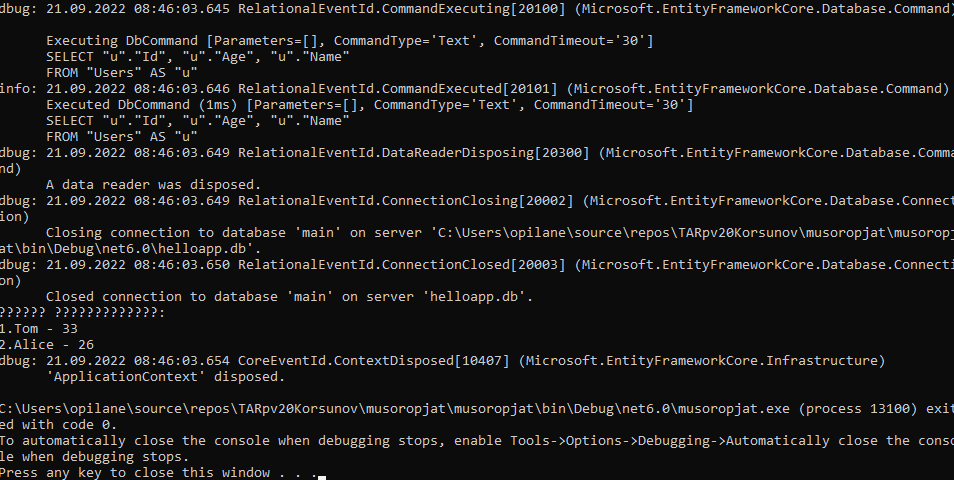
optionsBuilder.UseSqlite("Data Source=helloapp.db");

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.



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));

## Настройка логгирования

### Уровень логгирования

Метод LogTo() имеет ряд перегруженных версий, которые принимают разное количество параметров. Так, мы можем передать в LogTo уровень логгирования в виде одного из значений перечисления LogLevel:

* Trace: используется для вывода наиболее детализированных сообщений. Подобные сообщения могут нести важную информацию о приложении и его строении, поэтому данный уровень лучше использовать при разработке, но никак не при публикации
* Debug: для вывода информации, которая может быть полезной в процессе разработки и отладки приложения
* Information: уровень сообщений, позволяющий просто отследить поток выполнения приложения
* Warning: используется для вывода сообщений о неожиданных событиях, например, ошибках, которые не влияют не останавливают выполнение приложения, но в то же время должны быть иследованы
* Error: для вывода информации об ошибках и исключениях, которые возникли при текущей операции и которые не могут быть обработаны
* Critical: уровень критических ошибок, которые требуют немедленной реакции - ошибками операционной системы, потерей данных в бд, переполнение памяти диска и т.д.
* None: вывод информации в лог не применяется

### Конкретизация сообщений

Каждое сообщение в логе ассоциировано с определенным идентификатором события. По сути идентификаторы представляют тип возникающих событий

* SqlServerEventId: описывает сообщения, специфические для провайдера для MS SQL Server
* CoreEventId: описывает сообщения, общие для всех провайдеров Entity Framework Core
* RelationalEventId: описывает сообщения, общие для всех провайдеров для реляционных баз данных

Поскольку 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:

using Microsoft.EntityFrameworkCore;

using Microsoft.EntityFrameworkCore.Diagnostics;

public class ApplicationContext : DbContext

{

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

public ApplicationContext()

{

Database.EnsureDeleted();

Database.EnsureCreated();

}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

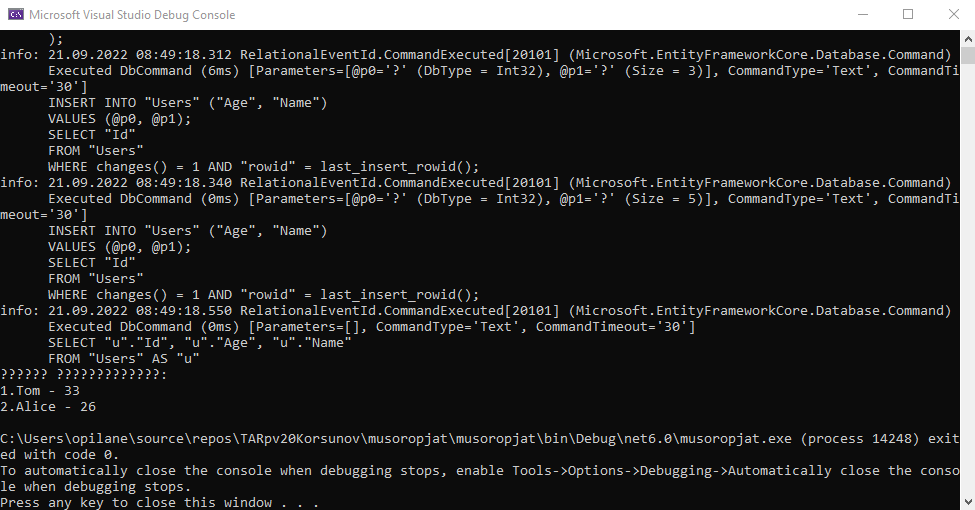
{

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

optionsBuilder.LogTo(Console.WriteLine, new[] { RelationalEventId.CommandExecuted });

}

}



## Категории сообщений

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

* Database.Command: категория для выполняемых команд, позволяет получить выполняемый код SQL
* Database.Connection : категория для операций подключения к БД
* Database.Transaction : категория для транзакций с бд
* Migration: категория для миграций
* Model: категория для действий, совершаемых при привязке модели
* Query: категория для запросов за исключением тех, что генерируют исполняемый код SQL
* Scaffolding: категория для действий, выполняемых в поцессе обратного инжиниринга (то есть когда по базе данных генерируются классы и класс контекста)
* Update: категория для сообщений вызова DbContext.SaveChanges()
* Infrastructure: категория для всех остальных сообщений

## 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.

### 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.

Теоретически и практически так можно делать. Стоит отметить, что при этом мы максимально контроллируем процесс изменения базы данных. Все данные, которые у меня были в таблице, так там и остались.

Тем не менее этот подход имеет много недостатков. В частности, менее искушенные программисты могут не знать, как сопоставляются типы между SQL и C#. При указании данных столбцов и/или таблиц мы можем допустить ошибку - например, вместо "Position" написать "Positon". В конце концов такой подход может занять много времени, особенно когда речь идет о куда больших изменениях схемы БД.

### Database.EnsureCreated и Database.EnsureDeleted

Если нам не важны данные в БД и мы хотим ее просто пересоздать для соответствия новой структуре классов, то через контекст данных можно вызывать метод Database.EnsureDeleted для удаления и затем метод Database.EnsureCreated для создания бд. Например, в коде самого контекста данных (обычно в конструкторе):

using Microsoft.EntityFrameworkCore;

public class ApplicationContext : DbContext

{

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");

}

}

В то же время при удалении происходит полное удаление данных, что в ряде случаев может быть нежелательным. И в этом случае лучше использовать миграции.

### Миграция

Миграция по сути предствляет план перехода базы данных от старой схемы к новой. Как использовать миграции?

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

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:

Update-Database

Если планируется использовать миграции, то лучше их использовать сразу при создании базы данных.

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 class User

{

public int Id { get; set; }

public string? Name { get; set; }

public int Age { get; set; }

}

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=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.

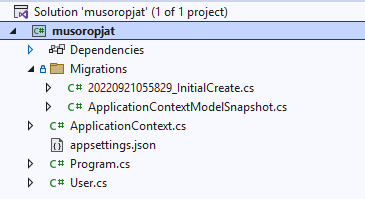
Также стоит отметить, что при самом первом применении миграции по отношению к БД SQLite Entity Framework пытается создать ее заново, однако если создаваемые таблицы в ней уже есть, то мы столкнемся с ошибкой. Поэтому следует убедиться, что по используемому пути нет файла базы данных с подобным именем.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

Название миграции произвольное. В данном случае это InitialCreate. Нажмем на Enter для создания миграции.

После этого в проект будет добавлена папка Migrations с классом миграции:



Папка содержит два файла:

* XXXXXXXXXXXXXX\_InitialCreate.cs: основной файл миграции, который содержит все применяемые действия
* [Имя\_контекста\_данных]ModelSnapshot.cs: содержит текущее состояние модели, используется при создании следующей миграции

В частности, мы можем открыть файл, который в названии содержит имя миграции, и посмотреть те действия, которые будет применяться:

using Microsoft.EntityFrameworkCore.Migrations;

#nullable disable

namespace musoropjat.Migrations

{

public partial class InitialCreate : Migration

{

protected override void Up(MigrationBuilder migrationBuilder)

{

migrationBuilder.CreateTable(

name: "Users",

columns: table => new

{

Id = table.Column<int>(type: "INTEGER", nullable: false)

.Annotation("Sqlite:Autoincrement", true),

Name = table.Column<string>(type: "TEXT", nullable: true),

Age = table.Column<int>(type: "INTEGER", nullable: false)

},

constraints: table =>

{

table.PrimaryKey("PK\_Users", x => x.Id);

});

}

protected override void Down(MigrationBuilder migrationBuilder)

{

migrationBuilder.DropTable(

name: "Users");

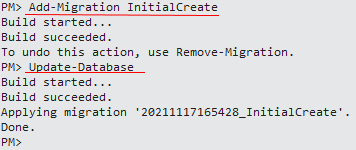
}

}

}

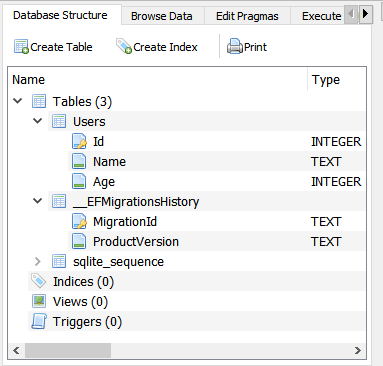
В миграции определяются два метода: Up() и Down(). В методе Up с помощью вызова метода CreateTable добавляется новое определение таблиц.

И в завершении чтобы выполнить миграцию, применим этот класс, набрав в той же консоли команду:



После выполнения миграции по указанному в методе optionsBuilder.UseSqlite("Data Source=D:\\helloapp.db") пути будет сгенерированная база данных. В случае с бд SQLite, для которой указан относительный путь (например, "Data Source=helloapp.db"), файл бд генерируется в папке проекта. Для других провайдеров - MS SQL Server, MySQL и т.д. бд генерируется на сервере бд в соответствии со строкой подключения.

Следует отметить, что кроме основных таблиц (в случае выше таблицы Users) база данных также будет содержать дополнительную таблицу \_EFMigrationsHystory, которая будет хранить информацию о миграциях.



Если мы изменим модель, например, добавим в класс User новое свойство:

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public int Age { get; set; }

public bool IsMarried { get; set; }

}

Чтобы база данных соответствовала измененной модели, также создадим новую миграцию и выполним ее:

Add-Migration IsMarriedToUserAdded

Update-Database

В данном случае будет создан класс миграции, который отражает добавление нового свойства в класс User:

using Microsoft.EntityFrameworkCore.Migrations;

#nullable disable

namespace musoropjat.Migrations

{

public partial class IsMarriedToUserAdded : Migration

{

protected override void Up(MigrationBuilder migrationBuilder)

{

migrationBuilder.AddColumn<bool>(

name: "IsMarried",

table: "Users",

type: "INTEGER",

nullable: false,

defaultValue: false);

}

protected override void Down(MigrationBuilder migrationBuilder)

{

migrationBuilder.DropColumn(

name: "IsMarried",

table: "Users");

}

}

}

Метод AddColumn как раз добавляет новый столбец в таблицу.

### Метод Migrate

В некоторых случаях, например, в приложениях с локальной базой данных (SQLite в UWP), мы можем выполнять миграции в процессе выполнения приложения. Для этого определен метод Database.Migrate() или его асинхронный двойник - Database.MigrateAsync(), который можно вызвать через объект контекста:

using Microsoft.EntityFrameworkCore;

using (ApplicationContext db = new ApplicationContext())

{

db.Database.Migrate(); // миграция

await db.Database.MigrateAsync(); // асинхронный метод для миграции

}

Стоит учитывать, что перед вызовом этого метода не следует вызывать метод EnsureCreated, который обходит миграции при создании базы данных, что вызывает ошибку при выполнении метода Migrate.

Чтобы задействовать этот метод, необходимо подключить пространство имен Microsoft.EntityFrameworkCore

### Создания скрипта sql для миграции

Entity Framework также позволяет создать из файлов миграции скрипт sql, который потом можно запустить для создания или реорганизации базы данных. Для создания скрипта sql необходимо ввести в окне Package Manager Console команду

Script-Migration

CREATE TABLE IF NOT EXISTS "\_\_EFMigrationsHistory" (

"MigrationId" TEXT NOT NULL CONSTRAINT "PK\_\_\_EFMigrationsHistory" PRIMARY KEY,

"ProductVersion" TEXT NOT NULL

);

BEGIN TRANSACTION;

CREATE TABLE "Users" (

"Id" INTEGER NOT NULL CONSTRAINT "PK\_Users" PRIMARY KEY AUTOINCREMENT,

"Name" TEXT NULL,

"Age" INTEGER NOT NULL

);

INSERT INTO "\_\_EFMigrationsHistory" ("MigrationId", "ProductVersion")

VALUES ('20220921055829\_InitialCreate', '6.0.9');

COMMIT;

BEGIN TRANSACTION;

ALTER TABLE "Users" ADD "IsMarried" INTEGER NOT NULL DEFAULT 0;

INSERT INTO "\_\_EFMigrationsHistory" ("MigrationId", "ProductVersion")

VALUES ('20220921060134\_IsMarriedToUserAdded', '6.0.9');

COMMIT;

Также можно передать название миграции, по которой необходимо создать скрипт:

Script-Migration InitialCreate

### Миграции в консоли

Если разработка осуществляется не в Visual Studio, то для миграций мы можем выполнять соответствующие команды в консоли. Для создания миграции:

dotnet ef migrations add InitialCreate

Для выполнения миграции:

dotnet ef database update

Для создания скрипта sql по миграции применяется следующая команда:

dotnet ef migrations script

С передачей названия миграции:

dotnet ef migrations script InitialCreate

### Миграция, если конструктор контекста принимает параметр DbContextOptions

Выше была рассмотрена миграция для контекста данных, который имеет конструктор без параметров и устанавливает настройки подключения в методе OnConfiguring(). Однако мы можем также передавать параметры подключения в контекст данных извне через конструктор с параметром типа DbContextOptions:

using Microsoft.EntityFrameworkCore;

public class ApplicationContext : DbContext

{

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

public ApplicationContext(DbContextOptions<ApplicationContext> options)

: base(options) { }

}

Например, у нас в проекте есть файл конфигурации appsettings.json:

{

"ConnectionStrings": {

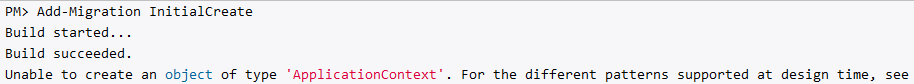
"DefaultConnection": "Data Source=helloapp.db"

}

}

Из которого в процессе выполнения приложения мы извлекаем строку подключения и передаем в контекст данных:

При создании миграции для такого контекста данных мы получим ошибку:

 Дело в том, что, если единственный конструктор класса контекста принимает параметр DbContext:

public ApplicationContext(DbContextOptions<ApplicationContext> options)

: base(options) { }

В этом случае при выполнении миграции инструментарий Entity Frameworkа ищет класс, который реализует интерфейс IDesignTimeDbContextFactory и который задает конфигурацию контекста.

Поэтому в этом случае нам необходимо добавить в проект подобный класс. Например:

using Microsoft.EntityFrameworkCore;

using Microsoft.EntityFrameworkCore.Design;

using Microsoft.Extensions.Configuration;

public class SampleContextFactory : IDesignTimeDbContextFactory<ApplicationContext>

{

public ApplicationContext CreateDbContext(string[] args)

{

var optionsBuilder = new DbContextOptionsBuilder<ApplicationContext>();

// получаем конфигурацию из файла appsettings.json

ConfigurationBuilder builder = new ConfigurationBuilder();

builder.SetBasePath(Directory.GetCurrentDirectory());

builder.AddJsonFile("appsettings.json");

IConfigurationRoot config = builder.Build();

// получаем строку подключения из файла appsettings.json

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

optionsBuilder.UseSqlite(connectionString);

return new ApplicationContext(optionsBuilder.Options);

}

}

Класс SampleContextFactory применяет интерфейс IDesignTimeDbContextFactory, который типизируется типом контекста данных - в данном случае класс ApplicationContext. Данный интерфейс содержит один метод CreateDbContext(), который должен возвращать созданный объект контекста данных.

В данном случае также получаем конфигурацию из файла appsettings.json и извлекаем из ее строку подключения и таким образом создаем контекст.

Хотя этот класс формально нигде не вызывается и никак не используется, фактически он вызывается инфраструктурой Entity Framework при создании миграции.

### Объединение миграций

Начиная с версии 6.0 Entity Framework позволяет создавать бандлы миграций - объединение миграций в виде исполняемого файла. Для создания бандла миграций надо в Visual Studio в окне Package Manager Console выполнить команду:

Bundle-Migration

А если использовуется .NET CLI, то в консоли надо перейти к папке проекта и выполнить команду

dotnet ef migrations bundle

После выполнения этих команд в папке решения будет сгенерирован файл efbundle (в Windows он будет иметь расширение exe). Запустим его.



И после запуска бандла будут последовательно применяться добавленные в бандл миграции.

# Провайдеры баз данных

## MS SQL Server

#### Program.cs

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public int Age { get; set; }

}

#### ApplicationContext.cs

using Microsoft.EntityFrameworkCore;

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")

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

#### Program.cs

// добавление данных

using (ApplicationContext db = new ApplicationContext())

{

Database.EnsureDeleted();

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}");

}

}

Вывод:



## MySQL

#### User.cs

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public int Age { get; set; }

}

#### ApplicationContext.cs

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

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

}

}

#### Program.cs

// добавление данных

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}");

}

}

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)).

Результат программы:



## PostgreSQL

#### User.cs

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public int Age { get; set; }

}

#### ApplicationContext.cs

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.

#### Program.cs

// добавление данных

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)

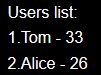
{

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

}

}

Консольный вывод:



### Миграции

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

using Microsoft.EntityFrameworkCore;

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");

}

}

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

После этого на сервере будет создана база данных, и мы сможем с ней взаимодействовать.

### Подключение к существующей базе данных

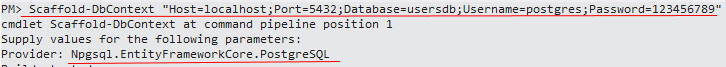
Для подключения к существующей базе данных в PostgreSQL необходимо в окне Package Manager Console выполнить команду Scaffold-DbContext, которой передается строка подключения и название провайдера, то есть Npgsql.EntityFrameworkCore.PostgreSQL (для выполнения этой команды тоже необходим пакет Microsoft.EntityFrameworkCore.Tools).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

Причем на данный момент название провайдера вводится вручную.



И если все прошло удачно, то EntityFramework Core автоматически сгенерирует все необходимые классы моделей и контекста.

# 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():

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)

{

// 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; }

}

В данном случае атрибут Column представляет аннотацию, которая указывает, что свойство Id будет сопоставляться со столбцом "user\_id" (а не Id, как бы было по умолчанию).

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

* Условности (conventions)
* Fluent API
* Аннотации данных

## 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:

#### ApplicationContext.cs

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");

}

}

#### User.cs

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public int Age { get; set; }

}

В данном случае, поскольку для класса User в классе контекста определено свойство типа DbSet

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

Таким образом, для сущности User будет создана таблица в бд.

### Ссылочные nullable-типы и DbSet

Класс DbSet, как и другие типы, является ссылочным. А, начиная с C# 10 и .NET 6 автоматически применяется функциональность ссылочных nullable-типов. И переменные/свойства тех типов, которые не являются nullable, следует инициализировать некотором значением перед их использованием.

То есть нам надо инициализировать свойство типа DbSet. Хотя в этом нет большого смысла, так как контструктор базового класса DbContext гарантирует, что все свойства типа DbSet будут инициализированы и соответственно в принципе не будут иметь значение null.

Тем не менее проблема остается, поскольку мы сталкиваемся с предупреждением. Чтобы выйти из этой ситуации мы можем инициализировать свойство с помощью выражения null!, которое говорит, что данное свойство в принципе не будет иметь значение null:

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

### 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:

#### User.cs

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public int Age { get; set; }

// навигационное свойство

public Company? Company { get; set; }

}

#### Company.cs

public class Company

{

public int Id { get; set; }

public string? Name { get; set; }

}

#### Country.cs

public class Country

{

public int Id { get; set; }

public string? Name { get; set; }

}

#### ApplicationContext.cs

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():

#### ApplicationContext.cs

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)

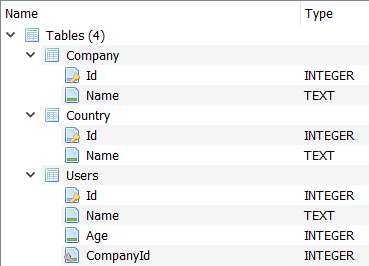
{

modelBuilder.Entity<Country>();

}

}

И если мы сейчас создадим и выполним миграции, то в базе данных будут уже три таблицы для сущностей:



### Исключение из модели

Иногда возникают ситуации, когда надо, наоборот, исключить сущность из модели. Например, в примере выше сущность Company ссылается на класс Company, и, допустим, мы не хотим, чтобы в базе данных была таблица Company. В этом случае мы можем использовать Fluent API или аннотации данных.

Применение Fluent API заключается в вызове метода Ignore():

#### ApplicationContext.cs

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.Ignore<Company>();

}

}

Аннотации данных предполагают установку над классом атрибута [NotMapped]:

#### Company.cs

using System.ComponentModel.DataAnnotations.Schema;

[NotMapped]

public class Company

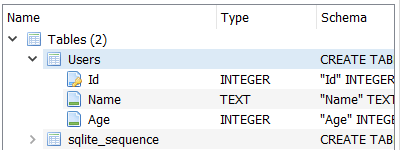
{

public int Id { get; set; }

public string Name { get; set; }

}

При исключении сущности Company в базе данных будет только одна таблица Users, причем она не будет содержать столбца, который бы сопоставлялся со свойством Company класса User:



## Свойства сущности

Класс User имеет три 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():

#### ApplicationContext.cs

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);

}

}

#### User.cs

using System.ComponentModel.DataAnnotations.Schema;

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

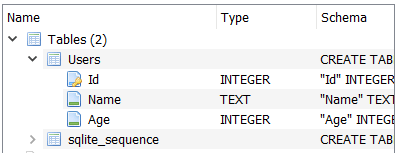
public int Age { get; set; }

[NotMapped]

public string? Address { get; set; }

}

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; }

}

## 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.

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.

Контекст данных:

using System.Collections.Generic;

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("Получение данных из БД");

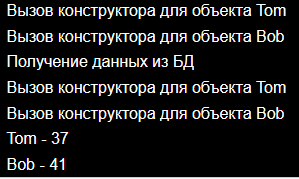
var users = db.Users.ToList();

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:



десь 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:

#### User.cs

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;

}

}

#### Company.cs

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;

}

}

## Использование полей сущности

Кроме свойств Entity Framework также может использовать поля класса (в том числе приватные) для сопоставления со столбцами. Например, возьмем следующую сущность:

#### User.cs

public class User

{

int id;

string name;

int age;

public int Id => id;

public int Age => age;

public User(string name, int age)

{

this.name = name;

this.age = age;

}

public void Print() => Console.WriteLine($"{id}. {name} - {age}");

}

Здесь в классе User определено три поля. Все они приватные, недоступные извне. Кроме того, есть два свойства для чтения, которые возвращают значения полей. Два поля - name и age устанавливаются только через конструктор. Третье поле - id, как мы ожидаем, будет устанавливаться при добавлении объекта сущности в базу данных.

Настроим класс контекста для применения этих полей:

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("Id").HasField("id");

modelBuilder.Entity<User>().Property("Age").HasField("age");

modelBuilder.Entity<User>().Property("name");

}

}

Для сопоставления полей со свойствами и столбцами применяется Fluent API. Чтобы использовать поле в качестве свойства при сопоставлении со столбцами применяется метод Property(), в который передается название поля: modelBuilder.Entity<User>().Property("name");

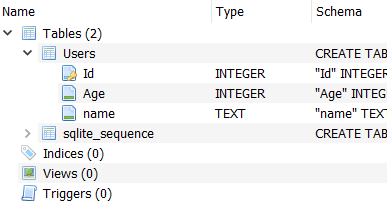
То есть в данном случае мы говорим, что мы хотим, чтобы поле "name" выступало в качестве свойства сущности и сопоставлялось со столбцом в таблице бд.

С помощью метода HasField() устанавливается поле, которое используется для свойства. Так, в выражении

modelBuilder.Entity<User>().Property("Id").HasField("id");

Для свойства Id будет использоваться поле id

В итоге Entity Framework создаст следующую таблицу:



В программе мы можем создать объект User и добавить в бд:

using (ApplicationContext db = new ApplicationContext())

{

User bob = new User("Bob", 30);

User kate = new User("Kate", 29);

db.Users.Add(bob);

db.Users.Add(kate);

db.SaveChanges();

var users = db.Users.ToList();

foreach (User user in users)

{

user.Print();

}

}

Консольный вывод программы:



## 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:

#### User.cs

using System.ComponentModel.DataAnnotations.Schema;

[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():

#### ApplicationContext.cs

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>().ToTable("People");

}

}

#### User.cs

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

}

С помощью дополнительного параметра schema можно определить схему, к которой будет принадлежать таблица:

modelBuilder.Entity<User>().ToTable("People", schema: "userstore");

### Сопоставление столбцов

По умолчанию каждое свойство сопоставляется с одноименным столбцом.

#### Атрибут Column

Атрибут Column переопределяет сопоставление:

#### User.cs

public class User

{

[Column("user\_id")]

public int Id { get; set; }

public string? Name { get; set; }

}

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

### Метод HasColumnName

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

#### ApplicationContext.cs

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");

}

}

#### User.cs

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

}

## 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:

#### User.cs

public class User

{

public int Id { get; set; }

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

public string? Company { get; set; }

public int Age { get; set; }

}

В данном случае свойство Name не представляет nullable-тип, поэтому оно рассматривается как обязательное (как и свойство Age). А свойство Company представляет nullable-тип - string?, соответственно является необязательным. Поэтому для этой сущности в SQLite будет сгенерирована следующая таблица:

CREATE TABLE "Users" (

"Id" INTEGER NOT NULL,

"Name" TEXT NOT NULL,

"Company" TEXT,

"Age" INTEGER NOT NULL,

CONSTRAINT "PK\_Users" PRIMARY KEY("Id" AUTOINCREMENT)

);

Здесь мы видим, что столбец Company допускает значение NULL, а столбец Name - не допускает благодаря установке атрибута NOT NULL. Хотя здесь приведен пример бд SQLite, но для других систем баз данных будет действовать аналогичная логика.

### Атрибут Required

Атрибут Required указывает, что данное свойство обязательно для установки, то есть будет иметь определение NOT NULL в БД, даже если оно представляет nullable-тип:

using System.ComponentModel.DataAnnotations;

public class User

{

public int Id { get; set; }

[Required]

public string? Name { get; set; }

}

А столбец Name в базе данных будет определен как NOT NULL.

Если мы не установим свойство Name у объекта User и попытаемся добавить этот объект в бд, то получим во время выполнения исключение типа Microsoft.EntityFrameworkCore.DbUpdateException:

#### Program.cs

using (ApplicationContext db = new ApplicationContext())

{

User tom = new User();

db.Users.Add(tom); // ! ошибка Microsoft.EntityFrameworkCore.DbUpdateException

db.SaveChanges();

}

### Метод IsRequired

То же самое можно сделать и через Fluent API с помощью метода IsRequired():

#### ApplicationContext.cs

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(b => b.Name).IsRequired();

}

}

#### User.cs

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

}

## Настройка ключей

По умолчанию в качестве ключа используется свойство, которое называется Id или [имя\_класса]Id. Например:

public class User

{

public int Id { get; set; }

//.........................

}

Или

public class User

{

public int UserId { get; set; }

//.........................

}

Для установки свойства в качестве первичного ключа с помощью аннотаций применяется атрибут [Key]:

using System.ComponentModel.DataAnnotations;

public class User

{

[Key]

public int Ident { get; set; }

public string? Name { get; set; }

}

Для конфигурации ключа с Fluent API применяется метод HasKey():

#### ApplicationContext.cs

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>().HasKey(u => u.Ident);

}

}

#### User.cs

public class User

{

public int Ident { get; set; }

public string? Name { get; set; }

}

Дополнительно с помощью Fluent API можно настроить имя ограничения, которое задается для первичного ключа. Для этого применяется метод HasName():

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<User>().HasKey(u => u.Ident);

}

### Составные ключи

С помощью Fluent API можно создать составной ключ из нескольких свойств:

#### ApplicationContext.cs

public class ApplicationContext : DbContext

{

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

public ApplicationContext()

{

Database.EnsureDeleted();

Database.EnsureCreated();

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<User>().HasKey(u => new { u.PassportSeria, u.PassportNumber });

}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

}

}

#### User.cs

public class User

{

public string? PassportNumber { get; set; }

public string? PassportSeria { get; set; }

public string? Name { get; set; }

}

Составной ключ можно создать только с помощью Fluent API. Применение подобного ключа:

#### Program.cs

using (ApplicationContext db = new ApplicationContext())

{

db.Users.Add(new User { PassportSeria = "1234", PassportNumber = "345678", Name = "Tom" });

db.Users.Add(new User { PassportSeria = "1234", PassportNumber = "345679", Name = "Bob" });

db.SaveChanges();

var users = db.Users.ToList();

foreach (var u in users)

Console.WriteLine($"{u.Name} : {u.PassportSeria} {u.PassportNumber}");

}

На уровне базы данных в случае с SQLite будет создаваться следующая таблица:

CREATE TABLE "Users" (

"PassportNumber" TEXT NOT NULL,

"PassportSeria" TEXT NOT NULL,

"Name" TEXT,

CONSTRAINT "PK\_Users" PRIMARY KEY("PassportSeria","PassportNumber")

);

### Альтернативные ключи

Альтернативные ключи представляют свойства, которые также, как и первичный ключ, должны иметь уникальное значение. В то же время альтернативные ключи не являются первичными. На уровне базы данных это выражается в установке для соответствующих столбцов ограничения на уникальность.

Для установки альтернативного ключа используется метод HasAlternateKey():

#### ApplicationContext.cs

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>().HasAlternateKey(u => u.Passport);

}

}

#### User.cs

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public string? Passport { get; set; }

}

В данном случае свойство Passport (серия и номер паспорта) будет альтернативным ключом. Созданная таблица Users в случае SQLite будет описываться следующим SQL-скриптом:

CREATE TABLE "Users" (

"Id" INTEGER NOT NULL,

"Name" TEXT,

"Passport" TEXT NOT NULL,

CONSTRAINT "AK\_Users\_Passport" UNIQUE("Passport"),

CONSTRAINT "PK\_Users" PRIMARY KEY("Id" AUTOINCREMENT)

);

Альтернативные ключи также могут быть составными:

#### ApplicationContext.cs

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>().HasAlternateKey(u => new { u.Passport, u.PhoneNumber });

}

}

#### User.cs

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public string? Passport { get; set; }

public string? PhoneNumber { get; set; }

}

В этом случае в SQLite будет создаваться следующая таблица:

CREATE TABLE "Users" (

"Id" INTEGER NOT NULL,

"Name" TEXT,

"Passport" TEXT NOT NULL,

"PhoneNumber" TEXT NOT NULL,

CONSTRAINT "AK\_Users\_Passport\_PhoneNumber" UNIQUE("Passport","PhoneNumber"),

CONSTRAINT "PK\_Users" PRIMARY KEY("Id" AUTOINCREMENT)

);

## Настройка индексов

Для увеличения производительности поиска в базе данных применяются индексы. По умолчанию индекс создается для каждого свойства, которое используется в качестве внешнего ключа. Однако Entity Framework также позволяет создавать свои индексы.

### Настройка индексов с помощью атрибутов

Для создания индекса можно использовать атрибут [Index]. Например:

#### User.cs

using Microsoft.EntityFrameworkCore;

[Index("PhoneNumber")]

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public string? Passport { get; set; }

public string? PhoneNumber { get; set; }

}

Первый и обязательный параметр атрибута указывает на свойство (или набор свойств), с которым будет ассоциирован индекс. В данном случае это свойство PhoneNumber.

Но также он может принимать набор свойств, для которых создается индекс. В этом случае названия свойств просто перечисляются через запятую:

using Microsoft.EntityFrameworkCore;

[Index("PhoneNumber", "Passport")]

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public string? Passport { get; set; }

public string? PhoneNumber { get; set; }

}

С помощью дополнительных параметров можно настроить уникальность и имя индекса:

using Microsoft.EntityFrameworkCore;

[Index("PhoneNumber", IsUnique = true, Name = "Phone\_Index")]

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public string? Passport { get; set; }

public string? PhoneNumber { get; set; }

}

В данном случае индекс будет называться Phone\_Index, а значение IsUnique = true указывает, что индекс должен быть уникальным.

### Настройка индексов с помощью Fluent API

Для создания индекса через Fluent API применяется метод HasIndex():

#### ApplicationContext.cs

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>().HasIndex(u => u.Passport);

}

}

#### User.cs

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public string? Passport { get; set; }

public string? PhoneNumber { get; set; }

}

### Уникальность индексов

С помощью дополнительного метода IsUnique() можно указать, что индекс должен иметь уникальное значение. Тем самым мы гарантируем, что в базе данных может быть только один объект с определенным значением для свойства-индекса:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<User>().HasIndex(u => u.Passport);

}

### Составные индексы

Также можно определить индексы сразу для нескольких свойств:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<User>().HasIndex(u => new { u.Passport, u.PhoneNumber });

}

### Имя индекса

Для установки имени индекса применяется метод HasDatabaseName(), в который передается имя индекса:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<User>()

.HasIndex(u => u.PhoneNumber)

.HasDatabaseName("PhoneIndex");

}

В данном случае для индекса будет использоваться свойство PhoneNumber, а называться он будет "PhoneIndex".

### Фильтры индексов

Некоторые системы управления базами данных позволяют определять индексы с фильрами или частичные индексы, которые позволяют выполнять индексацию только по ограниченному набору значений, что увеличивает производительность и уменьшает использование дискового простанства. И EntityFramework Core также позволяет создавать подобные индексы. Для этого применяется метод HasFilter(), в который передается sql-выражение, которое определяет условие фильтра. Например:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<User>()

.HasIndex(u => u.PhoneNumber)

.HasFilter("[PhoneNumber] IS NOT NULL");

}

В данном случае в качестве индекса будет использоваться столбец PhoneNumber. Причем только для тех строк, у которых в столбце PhoneNumber значение не равно NULL.

## Генерация значений свойств и столбцов

Если при добавлении или обновлении нового объекта у него уже установлено значение для свойства, Entity Framework использует это значение при вставке или обновлении в таблицу. Если для свойства явным образом не установлено значение, то для свойства устанавливается значение по умолчанию (null для nullable-типов, 0 для int, Guid.Empty для Guid и т.д.).

В зависимости от используемого провайдера базы данных, значения для свойств могут генерироваться на стороне клиента с помощью EF, либо же генерироваться уже на стороне базы данных при добавлении. Если значение генерируется базой данных, тогда при добавлении объекта в контекст EF может назначить временное значение. Это временное значение будет заменено значением, сгенерированным базой данных при вызове метода SaveChanges().

### Генерация ключей

По умолчанию для свойств первичных ключей, которые представляют типы int или GUID и которые имеют значение по умолчанию, генерируется значение при вставке в базу данных. Для всех остальных свойств значения по умолчанию не генерируется.

Например, пусть у нас имеет следующая модель:

#### ApplicationContext.cs

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");

}

}

#### User.cs

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

}

То после добавления в базу данных мы сможем получить сгенерированный Id:

using (ApplicationContext db = new ApplicationContext())

{

User user = new User { Name = "Tom" };

Console.WriteLine($"Id pered dobavleniem v kontekst {user.Id}"); // Id = 0

db.Users.Add(user);

db.SaveChanges();

Console.WriteLine($"Id posle dobalvenija v bazu dannyx {user.Id}"); // Id = 1

}

Вывод:



### Атрибут DatabaseGeneratedAttribute

Атрибут DatabaseGeneratedAttribute представляет аннотацию, которая позволяет изменить поведение базы данных при добавлении или изменении.

Например, мы хотим отключить автогенерацию значения при добавлении:

#### User.cs

using System.ComponentModel.DataAnnotations.Schema;

public class User

{

[DatabaseGenerated(DatabaseGeneratedOption.None)]

public int Id { get; set; }

public string? Name { get; set; }

}

И если теперь мы попробуем добавить объект без установленного Id, то EF в качесте временного значения будет использовать значение по умолчанию, то есть Id=0. В итоге при добавление более одного объекта в бд мы получим ошибку:

using (ApplicationContext db = new ApplicationContext())

{

db.Users.Add(new User { Name = "Tom" });

db.Users.Add(new User { Name = "Alice" }); // Ошибка

db.SaveChanges();

var users = db.Users.ToList();

foreach (var user in users)

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

}

В этом случае нам надо будет устанавливать Id:

#### ApplcationContext.cs

using (ApplicationContext db = new ApplicationContext())

{

db.Users.Add(new User { Id = 11, Name = "Tom" });

db.Users.Add(new User { Id = 23, Name = "Alice" });

db.SaveChanges();

var users = db.Users.ToList();

foreach (var user in users)

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

}

Если мы хотим, чтобы база данных, наоборот, сама генерировала значение, то в атрибут надо передавать значение DatabaseGeneratedOption.Identity:

#### User.cs

public class User

{

[DatabaseGenerated(DatabaseGeneratedOption.Identity)]

public int Id { get; set; }

public string Name { get; set; }

}

Но в данном случае для свойства Id это значение избыточно, так как значение генерируется по умолчанию.

### Fluent API

Отключение автогенерации значения для свойства с помощью Fluent API:

#### ApplicationContext.cs

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(b => b.Id).ValueGeneratedNever();

}

}

User.cs

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

}

### Значения по умолчанию

Для свойств, которые не представляют ключи и для которых не устанавливается значения, используются значения по умолчанию. Например, для свойств типа int это значение 0. С помощью метода HasDefaultValue() можно переопределить значение по умолчанию, которое будет применяться после добавления объекта в базу данных:

#### ApplicationContext.cs

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.Age).HasDefaultValue(18);

}

}

#### User.cs

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public int Age { get; set; }

}

В этом случае, если мы не укажем значение для свойства Age, то ему будет присвоено значение 18:

#### Program.cs

using (ApplicationContext db = new ApplicationContext())

{

User user1 = new User() { Name = "Tom" };

Console.WriteLine($"Age: {user1.Age}"); // 0

db.Users.Add(user1);

db.SaveChanges();

Console.WriteLine($"Age: {user1.Age}"); // 18

}

На уровне базы данных это будет проявляться в установке параметра DEFAULT:

CREATE TABLE "Users" (

"Id" INTEGER NOT NULL,

"Name" TEXT,

"Age" INTEGER NOT NULL DEFAULT 18,

CONSTRAINT "PK\_Users" PRIMARY KEY("Id" AUTOINCREMENT)

);

### HasDefaultValueSql

Метод HasDefaultValueSql() также определяет генерацию значения по умолчанию, только само значение устанавливается на основе кода SQL, который передается в этот метод.

Например, пусть в классе пользователя будет свойство CreatedAt, которое представляет дату занесения пользователя в базу данных:

#### User.cs

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public int Age { get; set; }

public DateTime CreatedAt { get; set; }

}

Для генерации значения этого свойства в базе данных можно вызывать специальные функции, которые применяются в той или иной СУБД. Например, в MS SQL Server/T-SQL это функция GETDATE(), в SQLite это функции DATETIME()/DATE() и т.д. Например:

#### ApplicationContext.cs

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.CreatedAt)

.HasDefaultValueSql("DATETIME('now')");

}

}

В метод HasDefaultValueSql() передается SQL-выражение, которые вызывается при добавлении объекта User в базу данных. Поскольку в данном случае используется база данных SQLite, то в качестве SQL-выражения передается вызов функции DATETIME('now') - "now" здесь указывает, что мы хотим получить текущую дату.

### Вычисляемые столбцы

Столбцы могут иметь значение, которое вычисляется на основании остальных столбцов. Например, пусть модель User имеет свойства для хранения имени и фамилии:

public class User

{

public int Id { get; set; }

public string? Name { get; }

public string? FirstName { get; set; }

public string? LastName { get; set; }

public int Age { get; set; }

}

А свойство Name должно представлять объединение свойств FirstName и LastName. И через Fluent API с помощью метода HasComputedColumnSql() можно установить в бд SQL-выражение, которое будет устанавливать значение столбца Name на основании столбцов FirstName и LastName:

#### ApplicationContext.cs

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.Name)

.HasComputedColumnSql("FirstName || ' ' || LastName");

}

}

Применение:

using (ApplicationContext db = new ApplicationContext())

{

User user1 = new User() { FirstName = "Tom", LastName = "Smith", Age = 36 };

Console.WriteLine(user1.Name); // до добавления Name имеет значение по умолчанию

db.Users.Add(user1);

db.SaveChanges();

Console.WriteLine(user1.Name); // Tom Smith

}

Полученный результат:



## Ограничения свойств

### Установка ограничений

С помощью метода HasCheckConstraint() можно установить ограничение для столбца. На уровне базы данных это приведет к установке для столбца атрибута CHECK, который задает ограничение.

В метод HasCheckConstraint() передается название столбца и sql-выражение, которое выполняет проверку корректности передаваемого значения. Например, у пользователя есть возраст, который должен укладываться в некоторые допустимые рамки. К примеру, возраст не может быть меньше 0 и больше 120:

#### ApplicationContext.cs

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>().HasCheckConstraint("Age", "Age > 0 AND Age < 120");

}

}

#### User.cs

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public int Age { get; set; }

}

В выражении

HasCheckConstraint("Age", "Age > 0 AND Age < 120");

в качестве первого параметра передается название столбца - в данном случае столбец "Age", а в качестве второго параметра - выражение SQL, которое будет использоваться в качестве ограничения.

В этом случае на уровне базы данных SQLite для сущности User будет создаваться следующая таблица:

CREATE TABLE "Users" (

"Id" INTEGER NOT NULL,

"Name" TEXT,

"Age" INTEGER NOT NULL,

CONSTRAINT "PK\_Users" PRIMARY KEY("Id" AUTOINCREMENT),

CONSTRAINT "CK\_Users\_Age" CHECK("Age" > 0 AND "Age" < 120)

);

Соответственно мы не сможем добавить в бд объект User, у которого значение свойства Age будет меньше 0 или больше 120.

using (ApplicationContext db = new ApplicationContext())

{

User bob = new User() { Name = "Bob", Age = 123 };

db.Users.Add(bob);

db.SaveChanges(); // ! Исключение

}

В качестве третьего параметра в HasCheckConstraint, передается делегат Action, который принимает объект CheckConstraintBuilder для настройки ограничия. В частности, мы можем задать имя ограничения:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<User>()

.HasCheckConstraint("Age", "Age > 0 AND Age < 120", c => c.HasName("CK\_User\_Age"));

}

### Ограничения по длине

Как правило, по умолчанию для строкового свойства в таблице создается столбец для хранения строки неограниченной длины. Используя аннотации данных или Fluent API, мы можем ограничить строку по длине.

Ограничение максимальной длины применяется только к строкам и к массивам, например, byte[].

### Атрибут MaxLength

В аннотациях данных ограничение по длине устанавливается с помощью атрибута MaxLength:

using System.ComponentModel.DataAnnotations;

public class User

{

public int Id { get; set; }

[MaxLength(50)]

public string? Name { get; set; }

}

Стоит отметить, что данное ограничение будет действовать только для тех систем баз данных, которые поддерживают данную возможность. Например, для бд SQLite это не будет иметь никакого значения. А в случае с бд MS SQL Server столбец Name в базе данных будет иметь тип nvarchar(50) и тем самым иметь ограничение по длине.

Надо отметить, что также есть атрибут [MinLength], который устанавливает минимальную длину, но он на определение таблицы не влияет.

#### Метод HasMaxLength

В Fluent API ограничение по длине устанавливается с помощью метода HasMaxLength():

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(b => b.Name).HasMaxLength(50);

}

}

## Конфигурация моделей

С помощью атрибутов и Fluent API для сущостей и их свойств можно установить многочисленные настройки. Однако, если настроек очень много, то они могут утяжелять класс контекста и сущностей. В этом случае Entity Framework Core позволяет вынести конфигурацию сущностей в отдельные классы.

Для вынесения конфигурации во вне необходимо создать класс конфигурации, реализующий интерфейс EntityTypeConfiguration<T>.

К примеру, пусть у нас есть следующий класс контекста и моделей:

ApplicationContext.cs

using Microsoft.EntityFrameworkCore;

public class ApplicationContext : DbContext

{

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

public DbSet<Company> Companies { 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").Property(p => p.Name).IsRequired();

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

modelBuilder.Entity<Company>().ToTable("Enterprises")

.Property(c => c.Name).IsRequired();

}

}

#### User.cs

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public int Age { get; set; }

}

#### Company.cs

public class Company

{

public int Id { get; set; }

public string? Name { get; set; }

}

Вся конфигурация здесь определена в методе OnModelCreating(). В принципе он не содержит много кода, однако при наличии гораздо большего количества сущностей и более изощренной их конфигурации с помощью Fluent API данный метод мог бы сильно раздуться в размерах. И теперь изменим определение контекста, применив конфигурации:

#### ApplicationContext.cs

using Microsoft.EntityFrameworkCore;

using Microsoft.EntityFrameworkCore.Metadata.Builders;

public class ApplicationContext : DbContext

{

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

public DbSet<Company> Companies { 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.ApplyConfiguration(new UserConfigure ());

modelBuilder.ApplyConfiguration(new CompanyConfigure());

}

}

#### UserConfigure.cs

using Microsoft.EntityFrameworkCore;

using Microsoft.EntityFrameworkCore.Metadata.Builders;

public class UserConfigure : IEntityTypeConfiguration<User>

{

public void Configure(EntityTypeBuilder<User> builder)

{

builder.ToTable("People").Property(p => p.Name).IsRequired();

builder.Property(p => p.Id).HasColumnName("user\_id");

}

}

#### CompanyConfigure.cs

using Microsoft.EntityFrameworkCore;

using Microsoft.EntityFrameworkCore.Metadata.Builders;

public class CompanyConfigure : IEntityTypeConfiguration<Company>

{

public void Configure(EntityTypeBuilder<Company> builder)

{

builder.ToTable("Enterprises").Property(c => c.Name).IsRequired();

}

}

#### User.cs

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public int Age { get; set; }

}

#### Company.cs

public class Company

{

public int Id { get; set; }

public string? Name { get; set; }

}

Теперь конфигурация моделей вынесена в отдельные классы. А для добавления конкретных конфигураций в контекст используется метод modelBuilder.ApplyConfiguration(), которому передается нужный объект конфигурации. В итоге по своему действию первый и второй варианты контекста будут идентичны.

В качестве альтернативы мы могли бы использовать еще один вариант. Вместо выделения отдельных классов конфигураций определить конфигурацию в виде отдельных методов в том же классе контекста данных:

#### ApplicationContext.cs

using Microsoft.EntityFrameworkCore;

using Microsoft.EntityFrameworkCore.Metadata.Builders;

public class ApplicationContext : DbContext

{

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

public DbSet<Company> Companies { 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>(UserConfigure);

modelBuilder.Entity<Company>(CompanyConfigure);

}

// конфигурация для типа User

public void UserConfigure(EntityTypeBuilder<User> builder)

{

builder.ToTable("People").Property(p => p.Name).IsRequired();

builder.Property(p => p.Id).HasColumnName("user\_id");

}

// конфигурация для типа Company

public void CompanyConfigure(EntityTypeBuilder<Company> builder)

{

builder.ToTable("Enterprises").Property(c => c.Name).IsRequired();

}

}

Здесь конфигурация определяется для каждого типа в отдельном методе, который в качестве параметра принимает объект EntityTypeBuilder<T>. Затем метод передается в вызов modelBuilder.Entity<T>() для соответствующей модели.

### Атрибут EntityTypeConfiguration

Еще один альтернативный вариант применения конфигураций представляет атрибут EntityTypeConfiguration, который применяется к сущности и который получает тип класса конфигурации:

#### ApplicationContext.cs

using Microsoft.EntityFrameworkCore;

using Microsoft.EntityFrameworkCore.Metadata.Builders;

public class ApplicationContext : DbContext

{

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

public DbSet<Company> Companies { get; set; } = null!;

public ApplicationContext()

{

Database.EnsureDeleted();

Database.EnsureCreated();

}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

}

}

#### UserConfiguration.cs

using Microsoft.EntityFrameworkCore.Metadata.Builders;

using Microsoft.EntityFrameworkCore;

public class UserConfiguration : IEntityTypeConfiguration<User>

{

public void Configure(EntityTypeBuilder<User> builder)

{

builder.ToTable("People").Property(p => p.Name).IsRequired();

builder.Property(p => p.Id).HasColumnName("user\_id");

}

}

#### CompanyConfiguration.cs

using Microsoft.EntityFrameworkCore.Metadata.Builders;

using Microsoft.EntityFrameworkCore;

public class CompanyConfiguration : IEntityTypeConfiguration<Company>

{

public void Configure(EntityTypeBuilder<Company> builder)

{

builder.ToTable("Enterprises").Property(c => c.Name).IsRequired();

}

}

#### User.cs

using Microsoft.EntityFrameworkCore;

[EntityTypeConfiguration(typeof(UserConfiguration))]

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public int Age { get; set; }

}

#### Company.cs

using Microsoft.EntityFrameworkCore;

[EntityTypeConfiguration(typeof(CompanyConfiguration))]

public class Company

{

public int Id { get; set; }

public string? Name { get; set; }

}

## Инициализация БД начальными данными

Иногда необходимо, чтобы при первом обращении база данных уже содержала некоторые данные. И Entity Framework Core позволяет инициализировать базу данных при ее создании некоторыми начальными данными. Благодаря этому к моменту первого использования базы данных она уже будет содержать начальные данные, которые мы сможем тут же использовать. И нам не потребуется вручную или программно добавлять в БД нужные нам данные.

Для инициализации БД при конфигурации определенной модели вызывается метод HasData(), в который передаются добавляемые данные:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<User>().HasData(new User { Id = 1, Name = "Tom", Age = 36 });

}

Например, инициализируем БД набором данных:

#### User.cs

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public int Age { get; set; }

}

#### ApplicationContext.cs

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>().HasData(

new User { Id = 1, Name = "Tom", Age = 23 },

new User { Id = 2, Name = "Alice", Age = 26 },

new User { Id = 3, Name = "Sam", Age = 28 }

);

}

}

У объекта ModelBuilder, который передается в OnModelCreating в качестве параметра, вызывается метод Entity<T>(). Этот метод типизируется типом, для которого будут добавляться начальные данные. То есть в данном случае данные будут добавляться в таблицу, где хранятся объекты User. Поэтому Entity типизируется типом User.

Далее по цепочке вызывается метод HasData(), который собственно и определяет начальные данные. В данном случае это набор из трех объектов User. При этом для каждого объекта необходимо установить значение первичного ключа - в данном случае значение свойства Id. Причем вне зависимости, генерирует ли база данных для данных автоматически индентификатор или нет, нам в любом случае его надо установить - это основное ограничение при инициализации БД начальными данными.

При этом следует учитывать, что инициализация начальными данными будет выполняться только в двух случаях:

* При выполнении миграции. (При создании миграции добавляемые данные автоматически включаются в скрипт миграции)
* При вызове метода Database.EnsureCreated(), который создает БД при ее отсутствии

В случае выше в конструкторе применяется метод Database.EnsureCreated(), поэтому при создании контекста данных

ApplicationContext db = new ApplicationContext();

будет автоматически производиться инициализация бд начальными данными.

Аналогично можно инициализировать данные нескольких сущностей, в том числе связанных между собой:

#### ApplicationContext.cs

using Microsoft.EntityFrameworkCore;

public class ApplicationContext : DbContext

{

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

public DbSet<Company> Companies { get; set; } = null!;

public ApplicationContext()

{

Database.EnsureDeleted();

Database.EnsureCreated();

}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseSqlite("Data Source=helloapp1.db");

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

// определяем компании

Company microsoft = new Company { Id = 1, Name = "Microsoft" };

Company google = new Company { Id = 2, Name = "Google" };

// определяем пользователей

User tom = new User { Id = 1, Name = "Tom", Age = 23, CompanyId = microsoft.Id };

User alice = new User { Id = 2, Name = "Alice", Age = 26, CompanyId = microsoft.Id };

User sam = new User { Id = 3, Name = "Sam", Age = 28, CompanyId = google.Id };

// добавляем данные для обеих сущностей

modelBuilder.Entity<Company>().HasData(microsoft, google);

modelBuilder.Entity<User>().HasData(tom, alice, sam);

}

}

#### User.cs

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public int Age { get; set; }

public Company? Company { get; set; }

public int? CompanyId { get; set; }

}

#### Company.cs

public class Company

{

public int Id { get; set; }

public string? Name { get; set; }

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

}

# Отношения между сущностями

## Внешние ключи и навигационные свойства

Для связей между сущностями в Entity Framework Core применяются внешние ключи и навигационные свойства. Так, возьмем к примеру следующие сущности:

#### Company.cs

public class Company

{

public int Id { get; set; }

public string? Name { get; set; } // название компании

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

}

#### User.cs

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public int CompanyId { get; set; } // внешний ключ

public Company? Company { get; set; } // навигационное свойство

}

И пусть у нас будет следующий контекст данных:

#### ApplicationContext.cs

using Microsoft.EntityFrameworkCore;

public class ApplicationContext : DbContext

{

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

public DbSet<Company> Companies { get; set; } = null!;

public ApplicationContext()

{

Database.EnsureDeleted();

Database.EnsureCreated();

}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

}

}

В данном случае сущность Company является главной сущностью, а класс User - зависимой, так как содержит ссылку на класс Company и зависит от этого класса.

Свойство CompanyId в классе User является внешним ключом, а свойство Company - навигационным свойством. По умолчанию название внешнего ключа должно принимать одно из следующих вариантов имени:

* Имя\_навигационного\_свойства+Имя ключа из связанной сущности - в нашем случае имя навигационного свойства Company, а ключа из модели Company - Id, поэтому в нашем случае нам надо обозвать свойство CompanyId, что собственно и было сделано в вышеприведенном коде.
* Имя\_класса\_связанной\_сущности+Имя ключа из связанной сущности - в нашем случае класс Company, а имя ключа из модели Company - Id, поэтому опять же в этом случае получается CompanyId

Свойство Users, представляющее список пользователей компании, в классе Company также является навигационным свойством.

В итоге после генерации базы данных в случае с SQLite таблицы будут иметь следующее определение:

CREATE TABLE "Users" (

"Id" INTEGER NOT NULL,

"Name" TEXT,

"CompanyId" INTEGER NOT NULL,

CONSTRAINT "FK\_Users\_Companies\_CompanyId" FOREIGN KEY("CompanyId") REFERENCES "Companies"("Id") ON DELETE CASCADE,

CONSTRAINT "PK\_Users" PRIMARY KEY("Id" AUTOINCREMENT)

);

CREATE TABLE "Companies" (

"Id" INTEGER NOT NULL,

"Name" TEXT,

CONSTRAINT "PK\_Companies" PRIMARY KEY("Id" AUTOINCREMENT)

);

### Установка главной сущности по навигационному свойству зависимой сущности

Причем при использовании классов нам достаточно установить либо одно навигационное свойство, либо свойство-внешний ключ. Например, укажем значение только для навигационного свойства:

### Program.cs

using (ApplicationContext db = new ApplicationContext())

{

Company company1 = new Company { Name = "Google" };

Company company2 = new Company { Name = "Microsoft" };

User user1 = new User { Name = "Tom", Company = company1 };

User user2 = new User { Name = "Bob", Company = company2 };

User user3 = new User { Name = "Sam", Company = company2 };

db.Companies.AddRange(company1, company2); // добавление компаний

db.Users.AddRange(user1, user2, user3); // добавление пользователей

db.SaveChanges();

foreach (var user in db.Users.ToList())

{

Console.WriteLine($"{user.Name} работает в {user.Company?.Name}");

}

}

Консольный вывод программы:



### Установка главной сущности по свойству-внешнему ключу зависимой сущности

Также можно использовать свойство-внешний ключ для установки связи:

using (ApplicationContext db = new ApplicationContext())

{

Company company1 = new Company { Name = "Google" };

Company company2 = new Company { Name = "Microsoft" };

db.Companies.AddRange(company1, company2); // добавление компаний

db.SaveChanges();

User user1 = new User { Name = "Tom", CompanyId = company1.Id };

User user2 = new User { Name = "Bob", CompanyId = company1.Id };

User user3 = new User { Name = "Sam", CompanyId = company2.Id };

db.Users.AddRange(user1, user2, user3); // добавление пользователей

db.SaveChanges();

foreach (var user in db.Users.ToList())

{

Console.WriteLine($"{user.Name} работает в {user.Company?.Name}");

}

}

Здесь надо отметить один момент: для устновки свойства внешнего ключа CompanyId нам необходимо знать его значение. Однако посколько оно связано со свойством Id класса Company, значение которого генерируется при добавление объекта в БД, соответственно в данном случае необходимо сначала добавить объект Company в базу данных.

### Установка зависимой сущности через навигационное свойство главной сущности

Выше для установки связи применялась зависимая сущность - User. Но мы также можем зайти с другой стороны и установить набор зависимых сущностей через навигационное свойство главной сущности:

#### Program.cs

using (ApplicationContext db = new ApplicationContext())

{

User user1 = new User { Name = "Tom" };

User user2 = new User { Name = "Bob" };

User user3 = new User { Name = "Sam" };

Company company1 = new Company { Name = "Google", Users = { user1, user2 } };

Company company2 = new Company { Name = "Microsoft", Users = { user3 } };

db.Companies.AddRange(company1, company2); // добавление компаний

db.Users.AddRange(user1, user2, user3); // добавление пользователей

db.SaveChanges();

foreach (var user in db.Users.ToList())

{

Console.WriteLine($"{user.Name} работает в {user.Company?.Name}");

}

}

### Отсутствие свойства внешнего ключа и навигационного свойства

Нам необязательно определять внешний ключ в зависимой сущности. Его можно опустить:

#### User.cs

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public Company? Company { get; set; } // навигационное свойство

}

В этом случае Entity Framework сам автоматически сгенерирует столбец для внешнего ключа в таблице Users.

CREATE TABLE "Users" (

"Id" INTEGER NOT NULL,

"Name" TEXT,

"CompanyId" INTEGER,

CONSTRAINT "PK\_Users" PRIMARY KEY("Id" AUTOINCREMENT),

CONSTRAINT "FK\_Users\_Companies\_CompanyId" FOREIGN KEY("CompanyId") REFERENCES "Companies"("Id")

);

Преимущество определения внешнего ключа в качестве свойства состоит в том, что в каких-то ситуациях нам может потребоваться только id связанной сущности. Тем более столбец для внешнего ключа в таблице в любом случае создается.

Более того, мы можем вовсе опустить навигационное свойство в классе User:

public class User

{

public int Id { get; set; }

public string Name { get; set; }

}

Но за счет того, что в классе Company также определено навигационное свойство Users все равно будет создаваться внешний ключ и связь таблицы Users и таблицы Companies. В частности, тогда в случае БД SQLite определение таблицы Users будет выглядеть следующим образом:

CREATE TABLE "Users" (

"Id" INTEGER NOT NULL,

"Name" TEXT,

"CompanyId" INTEGER,

CONSTRAINT "FK\_Users\_Companies\_CompanyId" FOREIGN KEY("CompanyId") REFERENCES "Companies"("Id"),

CONSTRAINT "PK\_Users" PRIMARY KEY("Id" AUTOINCREMENT)

);

В отличие от первой версии таблицы здесь не добавляется каскадное удаление.

## Настройка внешнего ключа через аннотации данных и Fluent API

### Настройка ключа с помощью аннотаций данных

В принципе название свойства - внешнего ключа необязательно должно следовать выше описанным условностям. Чтобы установить свойство в качестве внешнего ключа, применяется атрибут [ForeignKey]:

#### Company.cs

using System.ComponentModel.DataAnnotations.Schema;

public class Company

{

public int Id { get; set; }

public string? Name { get; set; } // название компании

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

}

#### User.cs

using System.ComponentModel.DataAnnotations.Schema;

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public int? CompanyInfoKey { get; set; }

[ForeignKey("CompanyInfoKey")]

public Company? Company { get; set; }

}

И пусть будет следующий контекст данных:

#### ApplicationContext.cs

using Microsoft.EntityFrameworkCore;

public class ApplicationContext : DbContext

{

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

public DbSet<Company> Companies { get; set; } = null!;

public ApplicationContext()

{

Database.EnsureDeleted();

Database.EnsureCreated();

}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

}

}

В случае БД SQLite будут генерироваться следующие таблицы:

CREATE TABLE "Users" (

"Id" INTEGER NOT NULL,

"Name" TEXT,

"CompanyInfoKey" INTEGER,

CONSTRAINT "PK\_Users" PRIMARY KEY("Id" AUTOINCREMENT),

CONSTRAINT "FK\_Users\_Companies\_CompanyInfoKey" FOREIGN KEY("CompanyInfoKey") REFERENCES "Companies"("Id")

);

CREATE TABLE "Companies" (

"Id" INTEGER NOT NULL,

"Name" TEXT,

CONSTRAINT "PK\_Companies" PRIMARY KEY("Id" AUTOINCREMENT)

);

### Настройка ключа с помощью Fluent API

Для настройки отношений между моделями с помощью Fluent API применяются специальные методы: HasOne / HasMany / WithOne / WithMany. Методы HasOne и HasMany устанавливают навигационное свойство для сущности, для которой производится конфигурация. Далее могут идти вызовы методов WithOne и WithMany, который идентифицируют навигационное свойство на стороне связанной сущности. Методы HasOne/WithOne применяются для обычного навигационного свойства, представляющего одиночный объект, а методы HasMany/WithMany используются для навигационных свойств, представляющих коллекции. Сам же внешний ключ устанавливается с помощью метода HasForeignKey:

#### ApplicationContext.cs

using Microsoft.EntityFrameworkCore;

public class ApplicationContext : DbContext

{

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

public DbSet<Company> Companies { get; set; } = null!;

public ApplicationContext()

{

Database.EnsureDeleted();

Database.EnsureCreated();

}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

}

}

#### User.cs

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public int? CompanyInfoKey { get; set; }

public Company? Company { get; set; }

}

#### Company.cs

public class Company

{

public int Id { get; set; }

public string? Name { get; set; } // название компании

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

}

В этом случе будут сгенерированы такие же таблицы, как и в предыдущем примере.

### Установка в качестве внешнего ключа произвольного свойства

Кроме того, с помощью Fluent API мы можем связь внешнего ключа не только с первичными ключами связанных сущностей, но и с другими свойствами. Например:

#### ApplicationContext.cs

using Microsoft.EntityFrameworkCore;

public class ApplicationContext : DbContext

{

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

public DbSet<Company> Companies { 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>()

.HasOne(u => u.Company)

.WithMany(c => c.Users)

.HasForeignKey(u => u.CompanyName)

.HasPrincipalKey(c => c.Name);

}

}

Метод HasPrincipalKey указывает на свойство связанной сущности, на которую будет ссылаться свойство-внешний ключ CompanyName. Кроме того, для свойства, указанного в HasPrincipalKey(), будет создавать альтернативный ключ.

Определение таблицы Users в SQLite будет выглядеть следующим образом:

CREATE TABLE "Users" (

"Id" INTEGER NOT NULL,

"Name" TEXT,

"CompanyName" TEXT,

CONSTRAINT "PK\_Users" PRIMARY KEY("Id" AUTOINCREMENT),

CONSTRAINT "FK\_Users\_Companies\_CompanyName" FOREIGN KEY("CompanyName") REFERENCES "Companies"("Name")

);

В программе при добавлении объектов в БД в этом случае можно установить как навигационное свойство, так и свойство внешнего ключа:

using (ApplicationContext db = new ApplicationContext())

{

Company company1 = new Company { Name = "Google" };

Company company2 = new Company { Name = "Microsoft" };

User user1 = new User { Name = "Tom", Company = company1 };

User user2 = new User { Name = "Bob", CompanyName = "Microsoft" };

User user3 = new User { Name = "Sam", CompanyName = company2.Name };

db.Companies.AddRange(company1, company2);

db.Users.AddRange(user1, user2, user3);

db.SaveChanges();

foreach (var user in db.Users.ToList())

{

Console.WriteLine($"{user.Name} работает в {user.Company?.Name}");

}

}

Результат работы будет тот же самый.

## Каскадное удаление

Каскадное удаление представляет автоматическое удаление зависимой сущности после удаления главной.

По умолчанию для сущностей применяется каскадное удаление, если наличие связанной сущности обязательно. Например:

#### Company.cs

using Microsoft.EntityFrameworkCore;

public class Company

{

public int Id { get; set; }

public string? Name { get; set; }

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

}

#### User.cs

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public int CompanyId { get; set; }

public Company? Company { get; set; }

}

#### ApplicationContext.cs

using Microsoft.EntityFrameworkCore;

public class ApplicationContext : DbContext

{

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

public DbSet<Company> Companies { get; set; } = null!;

public ApplicationContext()

{

Database.EnsureDeleted();

Database.EnsureCreated();

}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

}

}

Здесь свойство внешнего ключа имеет тип int, оно не допускает значения null и требует наличия конкретного значения - id связанного объекта Company (При этом то, что навигационное свойство Company допускает null, не имеет значения). То есть для объекта User обязательно необходимо наличия связанного объекта Company. Поэтому сгенерированная таблица Users будет иметь код:

CREATE TABLE "Users" (

"Id" INTEGER NOT NULL,

"Name" TEXT,

"CompanyId" INTEGER NOT NULL,

CONSTRAINT "PK\_Users" PRIMARY KEY("Id" AUTOINCREMENT),

CONSTRAINT "FK\_Users\_Companies\_CompanyId" FOREIGN KEY("CompanyId") REFERENCES "Companies"("Id") ON DELETE CASCADE

);

В определении внешнего ключа устанавливается каскадное удаление: ON DELETE CASCADE

Аналогичная связь будет устанавливаться, если свойство-внешний ключа отсутствует, а навигационное свойство НЕ представляет nullable-тип:

#### User.cs

public class User

{

Company? company;

public int Id { get; set; }

public string? Name { get; set; }

public Company Company

{

set => company = value;

get => company ?? throw new InvalidOperationException("Uninitialized property: Company");

}

}

Такая же таблица создается, если навигационное свойство представляет nullable-тип, но оно определено как обязательное, например, с помощью атрибута Required:

#### User.cs

using System.ComponentModel.DataAnnotations;

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

[Required]

public Company? Company { get; set; }

}

Например, добавим в базу данных 2 компании и 4 связанных с ними пользователей и затем удалим одну из компаний:

using (ApplicationContext db = new ApplicationContext())

{

// добавляем начальные данные

Company microsoft = new Company { Name = "Microsoft" };

Company google = new Company { Name = "Google" };

db.Companies.AddRange(microsoft, google);

db.SaveChanges();

User tom = new User { Name = "Tom", Company = microsoft };

User bob = new User { Name = "Bob", Company = google };

User alice = new User { Name = "Alice", Company = microsoft };

User kate = new User { Name = "Kate", Company = google };

db.Users.AddRange(tom, bob, alice, kate);

db.SaveChanges();

// получаем пользователей

var users = db.Users.ToList();

foreach (var user in users) Console.WriteLine(user.Name);

// Удаляем первую компанию

var comp = db.Companies.FirstOrDefault();

if (comp != null) db.Companies.Remove(comp);

db.SaveChanges();

Console.WriteLine("\nСписок пользователей после удаления компании");

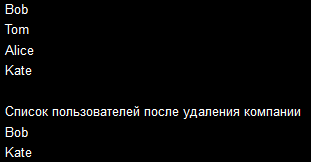
// снова получаем пользователей

users = db.Users.ToList();

foreach (var user in users) Console.WriteLine(user.Name);

}

Консольный вывод программы:



Удаление главной сущности - компании привело к удалению двух зависимых сущностей - пользователей.

Теперь изменим модели, указав необязательность наличия объекта Company:

#### Company.cs

public class Company

{

public int Id { get; set; }

public string? Name { get; set; }

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

}

#### User.cs

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public int? CompanyId { get; set; }

public Company? Company { get; set; }

}

Теперь внешний ключ имеет тип Nullable<int>, то есть он допускает значение null. Когда пользователь не будет принадлежать ни одной компании, это свойство будет иметь значение null. И в этом случае скрипт таблицы Users будет выглядеть следующим образом:

CREATE TABLE "Users" (

    "Id"    INTEGER NOT NULL,

    "Name"  TEXT,

    "CompanyId" INTEGER,

    CONSTRAINT "FK\_Users\_Companies\_CompanyId" FOREIGN KEY("CompanyId") REFERENCES "Companies"("Id"),

    CONSTRAINT "PK\_Users" PRIMARY KEY("Id" AUTOINCREMENT)

);

Аналогичная связь будет устанавливаться, если свойство-внешний ключа отсутствует, а навигационное свойство представляет nullable-тип:

public class User

{

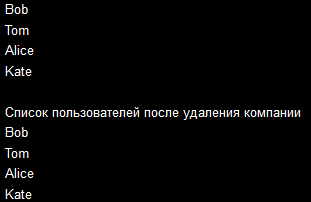
public int Id { get; set; }

public string? Name { get; set; }

public Company? Company { get; set; }

}

Если мы запустим ту же самую программу, то получим уже другой консольный вывод:



### Настройка каскадного удаления с помощью Fluent API

В Fluent API доступны три разных сценария, которые управляют поведением зависимой сущности в случае удаления главной сущности:

* Cascade: зависимая сущность удаляется вместе с главной
* SetNull: свойство-внешний ключ в зависимой сущности получает значение null
* Restrict: зависимая сущность никак не изменяется при удалении главной сущности

Например, установим каскадное удаление, даже если по умолчанию оно не предусматривается:

#### ApplicationContext.cs

using Microsoft.EntityFrameworkCore;

public class ApplicationContext : DbContext

{

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

public DbSet<Company> Companies { 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>()

.HasOne(u => u.Company)

.WithMany(c => c.Users)

.OnDelete(DeleteBehavior.Cascade);

}

}

#### Company.cs

public class Company

{

public int Id { get; set; }

public string? Name { get; set; }

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

}

#### User.cs

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public Company? Company { get; set; }

}

Соответственно чтобы отключить каскадное удаление, нам надо использовать вызов OnDelete(DeleteBehavior.SetNull).

## Загрузка связанных данных. Метод Include

Через навигационные свойства мы можем загружать связанные данные. И здесь у нас три стратегии загрузки:

* Eager loading (жадная загрузка)
* Explicit loading (явная загрузка)
* Lazy loading (ленивая загрузка)

В начале рассмотрим, что предствляет собой eager loading или жадная загрузка. Она позволяет загружать связанные данные с помощью метода Include(), в который передается навигационное свойство.

Например, пусть у нас есть следующие модели:

#### ApplicationContext.cs

using Microsoft.EntityFrameworkCore;

public class ApplicationContext : DbContext

{

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

public DbSet<Company> Companies { get; set; } = null!;

public ApplicationContext()

{

Database.EnsureDeleted();

Database.EnsureCreated();

}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

}

}

#### Company.cs

public class Company

{

public int Id { get; set; }

public string? Name { get; set; }

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

}

#### User.cs

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public int? CompanyId { get; set; }

public Company? Company { get; set; }

}

Добавим некоторые начальные данные и загрузим их из базы данных:

#### Program.cs

using Microsoft.EntityFrameworkCore;

using (ApplicationContext db = new ApplicationContext())

{

// пересоздадим базу данных

db.Database.EnsureDeleted();

db.Database.EnsureCreated();

// добавляем начальные данные

Company microsoft = new Company { Name = "Microsoft" };

Company google = new Company { Name = "Google" };

db.Companies.AddRange(microsoft, google);

User tom = new User { Name = "Tom", Company = microsoft };

User bob = new User { Name = "Bob", Company = google };

User alice = new User { Name = "Alice", Company = microsoft };

User kate = new User { Name = "Kate", Company = google };

db.Users.AddRange(tom, bob, alice, kate);

db.SaveChanges();

// получаем пользователей

var users = db.Users

.Include(u => u.Company) // подгружаем данные по компаниям

.ToList();

foreach (var user in users)

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

}

Для загрузки связанных данных используется метод Include:

var users = db.Users.Include(u => u.Company).ToList();

Поскольку свойство Company в классе User является навигационным свойством, через которое мы можем получить связанную с пользователем компанию, то мы можем использовать это свойство в методе Include. На уровне базы данных это выражение будет транслироваться в следующий SQL-запрос:

SELECT "u"."Id", "u"."CompanyId", "u"."Name", "c"."Id", "c"."Name"

FROM "Users" AS "u"

LEFT JOIN "Companies" AS "c" ON "u"."CompanyId" = "c"."Id"

То есть на уровне базы данных это будет означать использование выражения LEFT JOIN, который присоединяет данные из другой таблицы.

Консольный вывод программы:



Стоит отметить, что если данные уже ранее были загружены в контекст данных или просто ранее были в него добавлены, то можно не использовать метод Include для их получения, так как они уже в контексте. Например, возьмем выше приведенный пример:

#### Program.cs

using (ApplicationContext db = new ApplicationContext())

{

db.Database.EnsureDeleted();

db.Database.EnsureCreated();

Company microsoft = new Company { Name = "Microsoft" };

Company google = new Company { Name = "Google" };

db.Companies.AddRange(microsoft, google);

User tom = new User { Name = "Tom", Company = microsoft };

User bob = new User { Name = "Bob", Company = google };

User alice = new User { Name = "Alice", Company = microsoft };

User kate = new User { Name = "Kate", Company = google };

db.Users.AddRange(tom, bob, alice, kate);

db.SaveChanges();

var users = db.Users.ToList(); // метод Include не используется

foreach (var user in users)

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

}

Здесь не использован метод Include, но в итоге мы получим тот же самый результат. Почему? Потому что мы уже добавили все объекты в контекст при их создании с помощью методов db.Users.AddRange() и db.Companies.AddRange() и последующего сохранения с помощью вызова db.SaveChanges(). Объекты уже в контексте, нет смысла их притягивать с помощью метода Include. То же самое относится к ситуации, если ранее данные уже были загружены:

#### Program.cs

using (ApplicationContext db = new ApplicationContext())

{

var companies = db.Companies.ToList();

// получаем пользователей

var users = db.Users

//.Include(u => u.Company) // подгружаем данные по компаниям

.ToList();

foreach (var user in users)

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

}

Здесь к моменту получения пользователей компании уже загружены в констекст, поэтому нет смысла использоваться метод Include.

Теперь рассмотрим другую ситуацию:

using Microsoft.EntityFrameworkCore;

public static void Main(string[] args)

{

using (ApplicationContext db = new ApplicationContext())

{

// пересоздадим базу данных

db.Database.EnsureDeleted();

db.Database.EnsureCreated();

Company microsoft = new Company { Name = "Microsoft" };

Company google = new Company { Name = "Google" };

db.Companies.AddRange(microsoft, google);

User tom = new User { Name = "Tom", Company = microsoft };

User bob = new User { Name = "Bob", Company = google };

User alice = new User { Name = "Alice", Company = microsoft };

User kate = new User { Name = "Kate", Company = google };

db.Users.AddRange(tom, bob, alice, kate);

db.SaveChanges();

}

using (ApplicationContext db = new ApplicationContext())

{

var users = db.Users

.Include(u => u.Company) // добавляем данные по компаниям

.ToList();

foreach (var user in users)

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

}

}

Здесь программа логически разделена на две части: добавление объектов и их получение. Для каждой части создается свой объект ApplicationContext. В итоге при получении объект ApplicationContext не будет ничего знать об объектах, которые были добавлены в области действия другого объекта ApplicationContext. Поэтому в этом случае, если мы хотим получить связанные данные, нам необходимо использовать метод Include.

Подобным образом мы можем получить компании и подгрузить к ним связанных с ними пользователей через навигационное свойство Users в классе Company:

var companies = db.Companies

.Include(c => c.Users) // добавляем данные по пользователям

.ToList();

foreach (var company in companies)

{

Console.WriteLine(company.Name);

// выводим сотрудников компании

foreach (var user in company.Users)

Console.WriteLine(user.Name);

Console.WriteLine("----------------------"); // для красоты

}

Консольный вывод:



### Загрузка сущностей со сложной многоуровневой структурой

В примере выше структура моделей довольна простая - главная сущность связана с другой простой сущностью. Рассмотрим более сложную структуру моделей. Допустим, у каждой компании есть связанная сущность - страна, где находится компания: