# Глава 1

# Что такое Entity Framework Core

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

Как технология доступа к данным Entity Framework Core работает поверх платформы .NET и поэтому может использоваться на различных платформах стека .NET. Это и стандартные платформы типа Windows Forms, консольные приложения, WPF, UWP и ASP.NET Core. При этом кроссплатформенная природа EF Core позволяет задействовать ее не только на ОС Windows, но и на Linux и Mac OS X.

# Код и пояснение

## Класс User

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public int Age { get; set; }

}

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

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

## Класс ApplicationContext

{

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

{

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

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

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

{

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

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

}

}}

## Класс Program

using (ApplicationContext db = new ApplicationContext())

{

// создаем два объекта User

User artem = new User { Name = "Tom", ", Age = 19 };

User jaan = new User { Name = "Alice", Age = 22 };

// добавляем их в бд

db.Users.Add(artem);

db.Users.Add(jaan);

db.SaveChanges();

Console.WriteLine("Objektid on salvestatud");

// получаем объекты из бд и выводим на консоль

var users = db.Users.ToList();

Console.WriteLine("Objektide loend:");

foreach (User u in users)

{

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

}

**}**

## Результат

## 

# Подключение к существующей базе данных

## Класс helloappContext

public partial class helloappContext : DbContext

{

public helloappContext()

{

}

public helloappContext(DbContextOptions<helloappContext> options)

: base(options)

{

}

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

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

if (!optionsBuilder.IsConfigured)

{

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

}

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

OnModelCreatingPartial(modelBuilder);

}

partial void OnModelCreatingPartial(ModelBuilder modelBuilder);

}

}

## Переработанный класс Program

using (helloappContext db = new helloappContext())

{

// получаем объекты из бд и выводим на консоль

var users = db.Users.ToList();

Console.WriteLine("objektide loend:");

foreach (User u in users)

{

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

}

}

## Результат

## 

# Управление базой данны

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

## Database.EnsureCreated

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

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.

Данный метод можно вызвать в коде контекста данных, например, в конструкторе, чтобы гарантировать, что при вызове операций с данными БД уже создана:

public class ApplicationContext : DbContext

{

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

public ApplicationContext()

{

Database.EnsureCreated(); // гарантируем, что БД создана

}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

}

}

Либо можно вызывать вне класса контекста данных там, где нам доступен его объект

using (ApplicationContext db = new ApplicationContext())

{

db.Database.EnsureCreated();

// асинхронная версия

await db.Database.EnsureCreatedAsync();

}

## Database.EnsureDeleted

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

public class ApplicationContext : DbContext

{

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

public ApplicationContext()

{

Database.EnsureDeleted(); // гарантируем, что бд удалена

Database.EnsureCreated(); // гарантируем, что бд будет создана

}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

}

}

## Database.CanConnect

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

using (ApplicationContext db = new ApplicationContext())

{

bool isAvalaible = db.Database.CanConnect();

// bool isAvalaible2 = await db.Database.CanConnectAsync();

if (isAvalaible) Console.WriteLine(Andmebaas saadaval");

else Console.WriteLine("Andmebaas pole sadaval");

}

# Основные операции с данными. CRUD

## добавим класс контекста данных ApplicationContext

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 файле Program.cs определим все базовые операции с данными

/ Добавление

using (ApplicationContext db = new ApplicationContext())

{

User tom = new User { Name = "Tom", Age = 19 };

User alice = new User { Name = "Alice", Age = 22 };

// 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}");

}

}

## Результат

## 

# Конфигурация подключения

## Метод OnConfiguring

### Меням ApplicationContext.cs

public class ApplicationContext : DbContext

{

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

public string connectionString;

public ApplicationContext(string connectionString)

{

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

Database.EnsureCreated();

}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

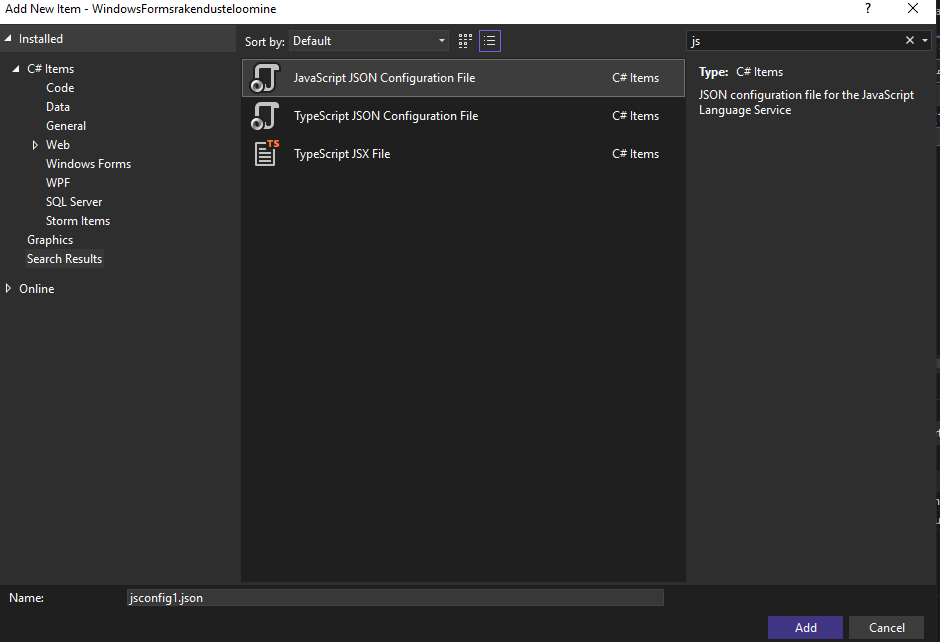
{

optionsBuilder.UseSqlite(connectionString);

}

}

### при создании объекта контекста передать строку подключения



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

{

User tom = new User { Name = "Tom", Age = 19 };

User alice = new User { Name = "Alice", Age = 22 };

// Добавление

db.Users.Add(tom);

db.Users.Add(alice);

db.SaveChanges();

}

## 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

public class ApplicationContext : DbContext

{

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

public ApplicationContext(DbContextOptions<ApplicationContext> options)

: base(options)

{

Database.EnsureCreated();

}

}

### Тогда мы могли бы использовать класс контекста следующим образом:

var optionsBuilder = new DbContextOptionsBuilder<ApplicationContext>();

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

// Добавление

using (ApplicationContext db = new ApplicationContext(options))

{

User tom = new User { Name = "Tom",", Age = 19 };

User alice = new User { Name = "Alice", Age = 22 };

// Добавление

db.Users.Add(tom);

db.Users.Add(alice);

db.SaveChanges();

}

## Файл конфигурации

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

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

Определим в этом файле следующий код:

{

"ConnectionStrings": {

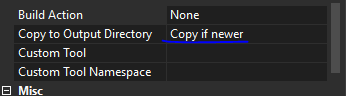
"DefaultConnection": "Data Source=helloapp.db"

}

}

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

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



### Для работы с БД возьмем ранее определенный класс контекста:

public class ApplicationContext : DbContext

{

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

public ApplicationContext(DbContextOptions<ApplicationContext> options)

: base(options)

{

Database.EnsureCreated();

}

}

### В файле Program.cs определим следующий код:

var builder = new ConfigurationBuilder();

// установка пути к текущему каталогу

builder.SetBasePath(Directory.GetCurrentDirectory());

// получаем конфигурацию из файла appsettings.json

builder.AddJsonFile("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;

// Добавление

using (ApplicationContext db = new ApplicationContext(options))

{

User tom = new User { Name = "Tom",Surname = "Tom", Age = 19 };

User alice = new User { Name = "Alice", Age = 22 };

// Добавление

db.Users.Add(tom);

db.Users.Add(alice);

db.SaveChanges();

}

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

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

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

В остальном работа с контекстом данных будет протекать также.

## Результат

## 

# 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. Рассмотрим оба варианта и начнем со встроенных возможностей.

## Метод LogTo

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

### Меняем ApplicationContext

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.

### Для тестирования пусть у нас определена следующая программа:

using (ApplicationContext db = new ApplicationContext())

{

User user1 = new User { Name = "Tom", Age = 19 };

User user2 = new User { Name = "Alice", Age = 22 };

db.Users.Add(user1);

db.Users.Add(user2);

db.SaveChanges();

var users = db.Users.ToList();

Console.WriteLine("kasutajate loend:");

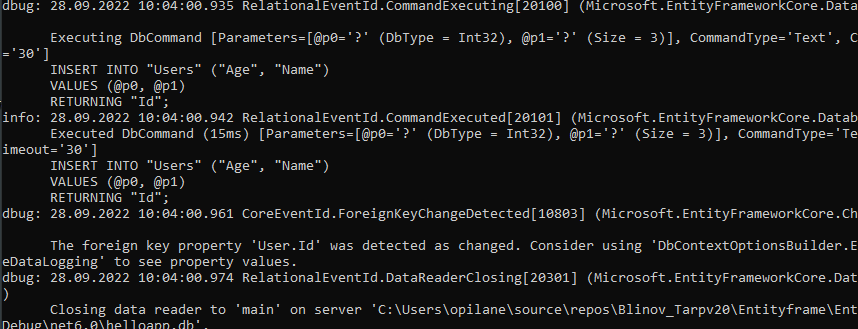
foreach (User u in users)

{

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

}

}



### Другим распространенным способом логгирования является вывод в файл:

public class ApplicationContext : DbContext

{

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

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

public ApplicationContext()

{

Database.EnsureDeleted();

Database.EnsureCreated();

}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

optionsBuilder.LogTo(logStream.WriteLine);

}

public override void Dispose()

{

base.Dispose();

logStream.Dispose();

}

public override async ValueTask DisposeAsync()

{

await base.DisposeAsync();

await logStream.DisposeAsync();

}

}

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



## 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:

public class ApplicationContext : DbContext

{

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

public ApplicationContext()

{

Database.EnsureDeleted();

Database.EnsureCreated();

}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

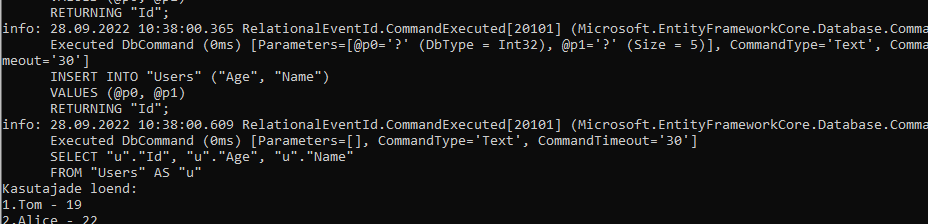
{

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.

### А для работы с базой данных использовался следующий контекст данных:

public class ApplicationContext : DbContext

{

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

public ApplicationContext()

{

Database.EnsureCreated();

}

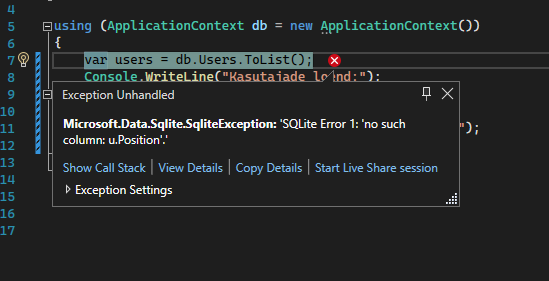
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

}

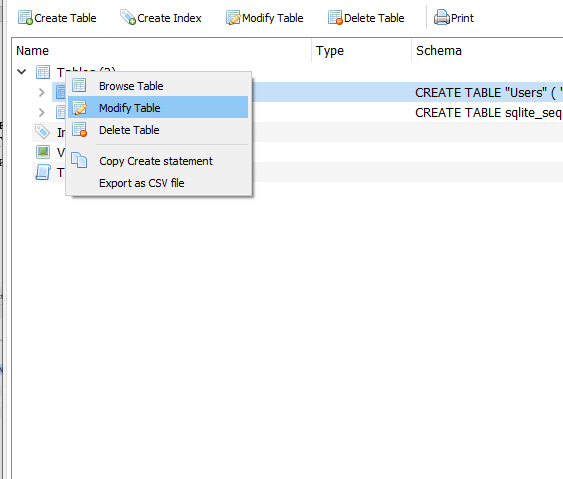
}



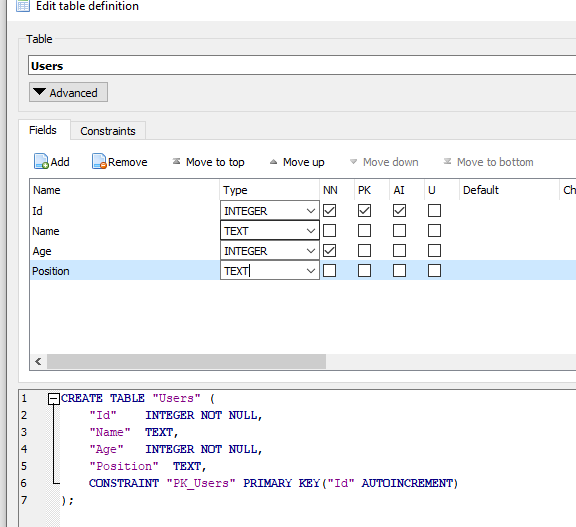
## Ручное изменение базы данных

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

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



В окне редактирования таблицы нажмем на кнопку Add для добавления нового столбца. После в определении таблицы добавится новая строка для определения нового столбца, где для названия столбца введем "Position", а в качестве типа столбца опеделим TEXT



Нажмем на кнопку OK, и в таблицу будет добавлен новый столбец Position, который будет соответствовать новуму свойству Position в классе User. И теперь таблица Users находится в соответствии с классом User. Больше никаких проблем при выполнении программы не возникнет.

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):

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.

Для создания миграции в окне Package Manager Console вводится команда

Add-Migration название миграции

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

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**.

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");

}

}

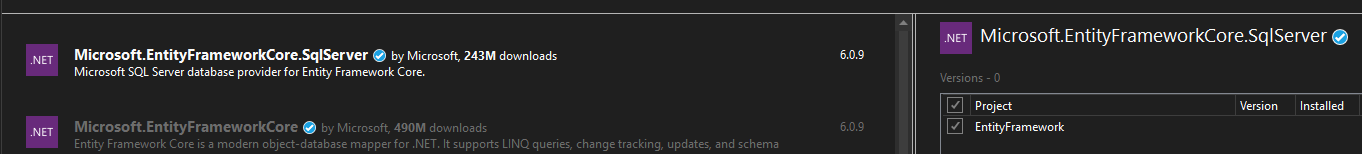
Обратите внимание, что в конструкторе контекста закомментирован метод Database.EnsureCreated(). В данном случае он не нужен. Более того при выполнении миграции этот метод вызывает ошибку. Этот момент следует учитывать.

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.

# Глава 2

# MS SQL Server

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



## Мнеяем ApplicationContext

public class ApplicationContext : DbContext

{

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

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

}

}

Для подключения к SQL Server у класса DbContextOptionsBuilder определен метод расширения UseSqlServer, в который передается строка подключения для соединения с MS SQL Server. Строка подключения разбивается на несколько частей:

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

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

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

using (ApplicationContext db = new ApplicationContext())

{

db.Database.EnsureDeleted();

db.Database.EnsureCreated();

// создаем два объекта User

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

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

// добавляем их в бд

db.Users.AddRange(user1, user2);

db.SaveChanges();

}

// получение данных

using (ApplicationContext db = new ApplicationContext())

{

// получаем объекты из бд и выводим на консоль

var users = db.Users.ToList();

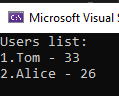
Console.WriteLine("Kasutajade loend:");

foreach (User u in users)

{

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

}}



## MySQL

## Для подключения к MySQL добавим через Nuget пакет **Pomelo.EntityFrameworkCore.MySql**:

## 

Стоит отметить, что также есть официальный провайдер от Oracle - **MySql.EntityFrameworkCore**, но он развивается довольно медленно, кроме того, не имеет поддержки ряда функционала. Поэтому предпочтительнее использовать пакет **Pomelo.EntityFrameworkCore.MySql**.

Для работы определим класс User:

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public int Age { get; set; }

}

И также определим контекст данных - класс ApplicationContext:

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

public ApplicationContext()

{

Database.EnsureCreated();

}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

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

}

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

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

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

using (ApplicationContext db = new ApplicationContext())

{

db.Database.EnsureDeleted();

db.Database.EnsureCreated();

// создаем два объекта User

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

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

// добавляем их в бд

db.Users.AddRange(user1, user2);

db.SaveChanges();

}

// получение данных

using (ApplicationContext db = new ApplicationContext())

{

// получаем объекты из бд и выводим на консоль

var users = db.Users.ToList();

Console.WriteLine("Users list:");

foreach (User u in users)

{

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

}

}

Результат программы:

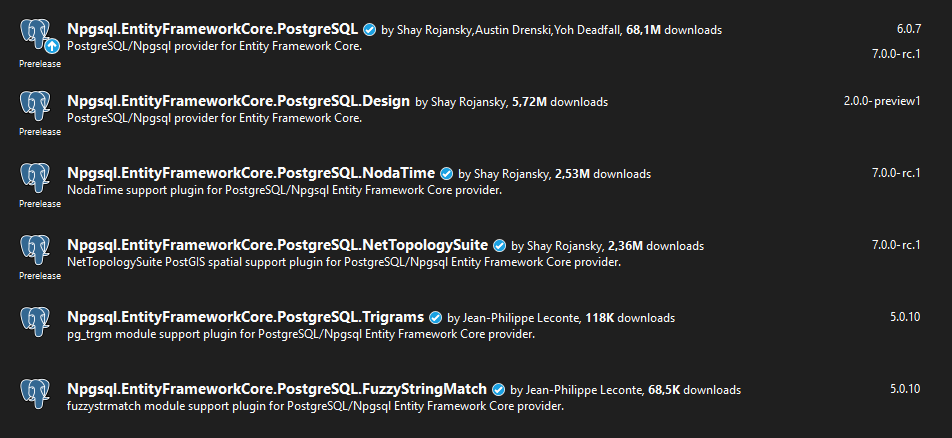
Objektid salvestatud

Objektide loend:

1.Tom - 33

2.Alice - 26

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



определим класс контекста данных:

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");

}

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



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

### Миграции

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

public class ApplicationContext : DbContext

{

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

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

}

}

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

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

Add-Migration Initial

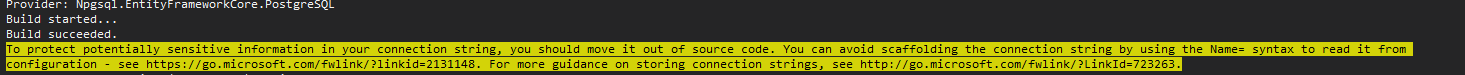
После генерации файла миграции для создания базы данных выполним команду:

Update-Database

### Подключение к существующей базе данных

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

Причем на данный момент название провайдера вводится вручную.



# ГЛАВА 3.

# Создание модели в Entity Framework Core

Fluent API и аннотации данных

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

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

### Fluent API

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

public 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:

public class User

{

[Column("user\_id")]

public long Id { get; set; }

public string? Name { get; set; }

public long 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).

### Включение сущностей в модель

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

public class ApplicationContext : DbContext

{

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

public ApplicationContext()

{

Database.EnsureDeleted();

Database.EnsureCreated();

}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

}

}

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public int Age { get; set; }

}

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

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

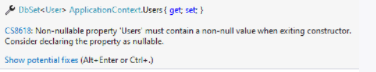
Таким образом, для сущности User будет создана таблица в бд.

### 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. Например, если мы напишем

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

То мы столкнемся с предупреждением:



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!;**

Другое решение - инициализировать свойство значением типа **Set<T>**:

public DbSet<User> Users => Set<User>();

### Включение сущностей в модель без DbSet

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

Например, пусть у нас определены следующие сущности:

public class Company

{

public int Id { get; set; }

public string? Name { get; set; }

}

public class Country

{

public int Id { get; set; }

public string? Name { get; set; }

}

И пусть у нас будет класс контекста данных:

public class ApplicationContext : DbContext

{

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

public ApplicationContext()

{

Database.EnsureDeleted();

Database.EnsureCreated();

}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

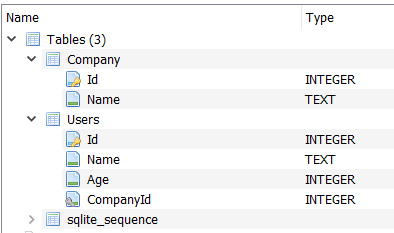
{

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()**:

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)

{

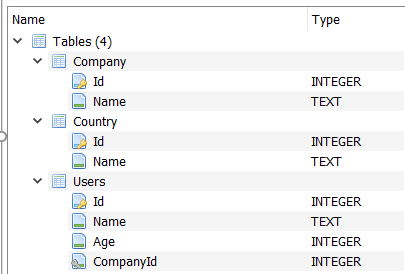
modelBuilder.Entity<Country>();

}

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()**: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>();

}

Aннотации данных предполагают установку над классом атрибута **[NotMapped]**:

[NotMapped]

public class Company

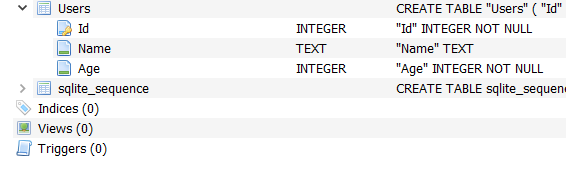
{

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

По умолчанию модель включает все свойства сущности, которые определены как публичные и которые открыты для записи и чтения. Например,

public class User

{

public long Id { get; set; }

public string? Name { get; set; }

public long Age { get; set; }

}

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()**:

public class ApplicationContext : DbContext

{

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

public ApplicationContext()

{

Database.EnsureDeleted();

Database.EnsureCreated();

}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

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

}

}

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

Исключение с помощью аннотаций данных:

public class User

{

public long Id { get; set; }

public string? Name { get; set; }

public long Age { get; set; }

[NotMapped]

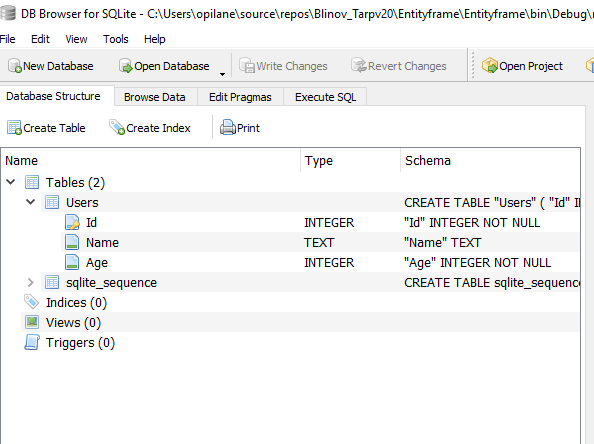
public string? Address { get; set; }

}

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

### Использование полей класса

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



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.

Рассмотрим на примере. Допустим, у нас есть следующая сущность User:

public int Id { get; set; }

public string Name { get; set; }

public int Age { get; set; }

public User(string name, int age)

{

Name = name;

Age = age;

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

}

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

Пусть у нас будет стандартный контекст данных:

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");

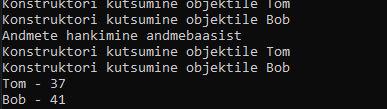
var users = db.Users.ToList();

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:



Здесь надо учитывать несколько моментов:

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

public class User

{

public int Id { get; set; }

public string Name { get; set; }

public int Age { get; set; }

public Company? Company { get; set; }

public User(string name, int age)

{

Name = name;

Age = age;

}

}

public class Company

{

public int Id { get; set; }

public string Name { get; set; }

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

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

}

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

## Использование полей сущности

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

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:

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");

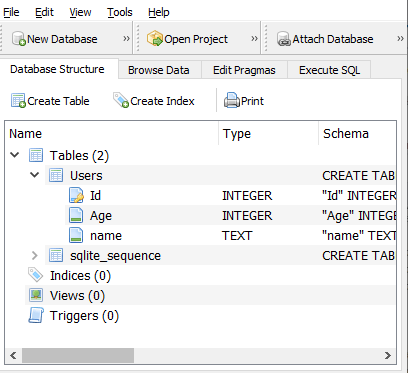
То есть в данном случае мы говорим, что мы хотим, чтобы поле "name" выступало в качестве свойства сущности и сопоставлялось со столбцом в таблице бд.

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

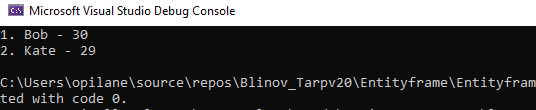
modelBuilder.Entity<User>().Property("Id").HasField("id");

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:



## Сопоставление таблиц и столбцов

#### Атрибут Table

Атрибут Table позволяет переопределить сопоставление с таблицей по имени:

[Table("People")]

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

}

Теперь сущность User будет сопоставляться с таблицей "People".

#### Метод ToTable

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

public 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");

}

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:

public class User

{

[Column("user\_id")]

public int Id { get; set; }

public string? Name { get; set; }

}

#### Метод HasColumnName

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

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

public ApplicationContext()

{

Database.EnsureDeleted();

Database.EnsureCreated();

}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

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

}

### Обязательные свойства

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

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

Например, возьмем следующую модель:

public class User

{

public int Id { get; set; }

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

public string? Company { get; set; }

public int Age { get; set; }

}

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:

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**:

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()**:

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();

}

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

public class User

{

public int Id { get; set; }

}

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

public class User

{

[Key]

public int Ident { get; set; }

public string? Name { get; set; }

}

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

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);

}

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).HasName("UsersPrimaryKey");

}

### Составные ключи

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

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

public ApplicationContext()

{

Database.EnsureDeleted();

Database.EnsureCreated();

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

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

}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

}

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

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()**:

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);

}

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:

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 });

}

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:

[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:

[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:

[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()**:

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);

}

#### 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).IsUnique();

}

#### 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**

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:

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

public ApplicationContext()

{

Database.EnsureDeleted();

Database.EnsureCreated();

}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

}

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

}

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 перед добавлением в контекст {user.Id}"); // Id = 0

db.Users.Add(user);

db.SaveChanges();

Console.WriteLine($"Id после добавления в базу данных {user.Id}"); // Id = 1

}

### Атрибут DatabaseGeneratedAttribute

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

Например, мы хотим отключить автогенерацию значения при добавлении:

[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:

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**:

[DatabaseGenerated(DatabaseGeneratedOption.Identity)]

public int Id { get; set; }

public string Name { get; set; }

### **Fluent API**

Отключение автогенерации значения для свойства с помощью Fluent API:

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();

}

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:

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);

}

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:

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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | 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:

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:

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 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:

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

}

## **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:

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");

}

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**:

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()**:

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:

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public int Age { get; set; }

}

public class Company

{

public int Id { get; set; }

public string? Name { get; set; }

}

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();

}

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:

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 UserConfiguration());

modelBuilder.ApplyConfiguration(new CompanyConfiguration());

}

}

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");

}

}

public class CompanyConfiguration : IEntityTypeConfiguration<Company>

{

public void Configure(EntityTypeBuilder<Company> builder)

{

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

}

}

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public int Age { get; set; }

}

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:

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:

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");

}

}

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");

}

}

public class CompanyConfiguration : IEntityTypeConfiguration<Company>

{

public void Configure(EntityTypeBuilder<Company> builder)

{

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

}

}

[EntityTypeConfiguration(typeof(UserConfiguration))]

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public int Age { get; set; }

}

[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:

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 }

);

}

}

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:

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);

}

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; }

}

public class Company

{

public int Id { get; set; }

public string? Name { get; set; }

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

}

# ГЛАВА 4.

# 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:

public class Company

{

public int Id { get; set; }

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

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

}

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:

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:

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();

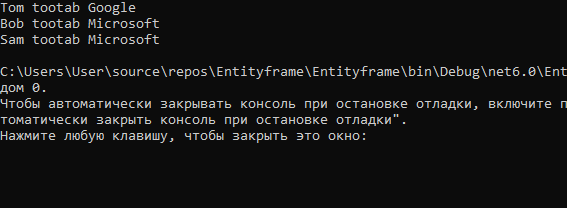
foreach (var user in db.Users.ToList())

{

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

}

}



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

Также можно использовать свойство-внешний ключ для установки связи:

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:

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:

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public Company? Company { get; set; }

}

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | 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; }

}

о 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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | 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]**:

public class Company

{

public int Id { get; set; }

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

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

}

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:

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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | 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**. 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**:

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.CompanyInfoKey);

}

}

public class Company

{

public int Id { get; set; }

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

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

}

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public int? CompanyInfoKey { get; set; }

public Company? Company { get; set; }

}

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:

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);

}

public class Company

{

public int Id { get; set; }

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

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

}

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public string? CompanyName { get; set; }

public Company? Company { get; set; }

}

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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | 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();

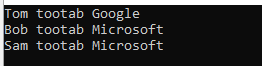
foreach (var user in db.Users.ToList())

{

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

}

}



## **Каскадное удаление**

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

public class Company

{

public int Id { get; set; }

public string? Name { get; set; }

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

}

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public int CompanyId { get; set; }

public Company? Company { get; set; }

## }

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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | 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:

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**:

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("\nKasutajate loend pärast ettevõtte kustutamist");

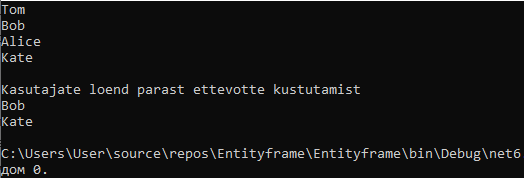
// снова получаем пользователей

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:

public class Company

{

public int Id { get; set; }

public string? Name { get; set; }

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

}

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public int? CompanyId { get; set; }

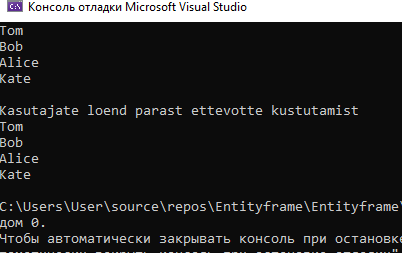
public Company? Company { get; set; }

}

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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | 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)  ); |

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:

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);

}

}

public class Company

{

public int Id { get; set; }

public string? Name { get; set; }

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

}

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public Company? Company { get; set; }

}

## **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:

public class Company

{

public int Id { get; set; }

public string? Name { get; set; }

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

}

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public int? CompanyId { get; set; }

public Company? Company { get; set; }

}

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");

}

Добавим некоторые начальные данные и загрузим их из базы данных:

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**:

|  |  |
| --- | --- |
| 1 | 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:

|  |  |
| --- | --- |
| 1  2  3 | 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:

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}");

}

Объекты уже в контексте, нет смысла их притягивать с помощью метода Include. То же самое относится к ситуации, если ранее данные уже были загружены:

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();

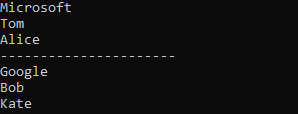
foreach (var user in users)

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:



### 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:

public class Country

{

public int Id { get; set; }

public string? Name { get; set; }

public List<Company> Companies { get; set; } = new();

}

public class Company

{

public int Id { get; set; }

public string? Name { get; set; }

public int CountryId { get; set; }

public Country? Country { get; set; }

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

}

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public int? CompanyId { get; set; }

public Company? Company { get; set; }

}

И пусть есть следующий контекст данных

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

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

public DbSet<Country> Countries { get; set; } = null!;

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

}

#### ThenInclude

Допустим, вместе с пользователями мы хотим загрузить и страны, в которых базируются компании пользователей. То есть получается, что нам нужно спуститься еще на уровень ниже: User - Company - Country. Для этого нам надо применить метод **ThenInclude()**, который работает похожим образом, что и Include:

using (ApplicationContext db = new ApplicationContext())

{

// пересоздадим базу данных

db.Database.EnsureDeleted();

db.Database.EnsureCreated();

Country usa = new Country { Name = "USA" };

Country japan = new Country { Name = "Japan" };

db.Countries.AddRange(usa, japan);

// добавляем начальные данные

Company microsoft = new Company { Name = "Microsoft", Country = usa };

Company sony = new Company { Name = "Sony", Country = japan };

db.Companies.AddRange(microsoft, sony);

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

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

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

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

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

db.SaveChanges();

}

// получение данных

using (ApplicationContext db = new ApplicationContext())

{

// получаем пользователей

var users = db.Users

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

.ThenInclude(c => c!.Country) // к компаниям подгружаем данные по странам

.ToList();

foreach (var user in users)

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

}

Вначале загружаются данные пользователям. Затем загружаются связанные данные по компании. И чтобы пойти дальше по цепочке навигационных свойств, надо использовать метод **ThenInclude()**, через который затем подгружаются страны компаний.

При загрузке связанных данных EF Core гарантирует, что если связанная сущность не установлена (например, свойство CompanyId в объекте User равно null) то данное навигационное свойство просто будет игнорироваться. Соответственно никакой ошибки в процессе получения данных не произойдет. Но поскольку компилятор не знает об этом, то он выдает предупреждение, например, в следующем случае:

|  |  |
| --- | --- |
| 1 | .ThenInclude(c => c.Country)    // 'c' may be null here |

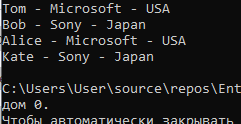
В этом случае мы можем использовать оператор **!** (null-forgiving оператор), чтобы указать, что значение null в данной ситуации невоможно.

|  |  |
| --- | --- |
| 1 | .ThenInclude(c => c!.Country)    // норм, компилятор доволен |

В итоге на уровне базы данных это выльется в следующий код SQL:

|  |  |
| --- | --- |
| 1  2  3  4 | SELECT "u"."Id", "u"."CompanyId", "u"."Name", "c"."Id", "c"."CountryId", "c"."Name", "c0"."Id", "c0"."Name"  FROM "Users" AS "u"  LEFT JOIN "Companies" AS "c" ON "u"."CompanyId" = "c"."Id"  LEFT JOIN "Countries" AS "c0" ON "c"."CountryId" = "c0"."Id" |

В итоге мы получим следующий консольный вывод:



#### Include

Также мы можем использовать тот же метод **Include** для загрузки данных далее по цепочке:

using (ApplicationContext db = new ApplicationContext())

{

// получаем пользователей

var users = db.Users

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

.ThenInclude(c => c!.Country) // к компаниям подгружаем данные по странам

.ToList();

foreach (var user in users)

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

}

В этом случае будет формироваться такой же sql-запрос, и мы получим аналогичный результат.

### Многоуровневая система данных

И в конце рассмотрим более сложную многоуровневую структуру сущностей:

public class City

{

public int Id { get; set; }

public string? Name { get; set; }

}

// страна компании

public class Country

{

public int Id { get; set; }

public string? Name { get; set; }

public int CapitalId { get; set; }

public City? Capital { get; set; } // столица страны

public List<Company> Companies { get; set; } = new();

}

public class Company

{

public int Id { get; set; }

public string? Name { get; set; }

public int CountryId { get; set; }

public Country? Country { get; set; }

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

}

// должность пользователя

public class Position

{

public int Id { get; set; }

public string? Name { get; set; }

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

}

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public int? CompanyId { get; set; }

public Company? Company { get; set; }

public int? PositionId { get; set; }

public Position? Position { get; set; }

}

Теперь у каждого пользователя также есть ссылка на должность, представленную классом Position. Компания хранит ссылку на страну Country, которая хранит ссылку на столицу в виде объекта City.

Для взаимодействия с бд определим следующий контекст данных:

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

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

public DbSet<City> Cities { get; set; } = null!;

public DbSet<Country> Countries { get; set; } = null!;

public DbSet<Position> Positions { get; set; } = null!;

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

}

Теперь добавим начальные и данные и загрузим пользователей с детальными данными:

using (ApplicationContext db = new ApplicationContext())

{

// пересоздадим базу данных

db.Database.EnsureDeleted();

db.Database.EnsureCreated();

Position manager = new Position { Name = "Manager" };

Position developer = new Position { Name = "Developer" };

db.Positions.AddRange(manager, developer);

City washington = new City { Name = "Washington" };

db.Cities.Add(washington);

Country usa = new Country { Name = "USA", Capital = washington };

db.Countries.Add(usa);

Company microsoft = new Company { Name = "Microsoft", Country = usa };

Company google = new Company { Name = "Google", Country = usa };

db.Companies.AddRange(microsoft, google);

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

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

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

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

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

db.SaveChanges();

}

using (ApplicationContext db = new ApplicationContext())

{

// получаем пользователей

var users = db.Users

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

.ThenInclude(comp => comp!.Country) // к компании добавляем страну

.ThenInclude(count => count!.Capital) // к стране добавляем столицу

.Include(u => u.Position) // добавляем данные по должностям

.ToList();

foreach (var user in users)

{

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

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

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

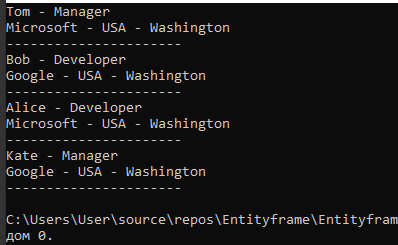
}

}

На уровне базы данных это будет транслироваться в следующий SQL-запрос:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | SELECT u.Id, u.CompanyId, u.Name, u.PositionId, c.Id, c.CountryId, c.Name, c0.Id, c0.CapitalId, c0.Name, c1.Id, c1.Name, p.Id, p.Name  FROM Users AS u  LEFT JOIN Companies AS c ON u.CompanyId == c.Id  LEFT JOIN Countries AS c0 ON c.CountryId == c0.Id  LEFT JOIN Cities AS c1 ON c0.CapitalId == c1.Id  LEFT JOIN Positions AS p ON u.PositionId == p.Id) |

В итоге мы получим следующий консольный вывод:



**Explicit loading**

Стратегия **Explicit loading** предполагает явную загрузку данных с помощью метода **Load()**. Допустим, у нас имеются следующие сущности и контекст данных

public class Company

{

public int Id { get; set; }

public string? Name { get; set; }

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

}

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public int? CompanyId { get; set; }

public Company? Company { get; set; }

}

public class ApplicationContext : DbContext

{

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

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

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

}

}

Загрузим данные по первой компании:

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();

}

using (ApplicationContext db = new ApplicationContext())

{

Company? company = db.Companies.FirstOrDefault();

if (company != null)

{

db.Users.Where(u => u.CompanyId == company.Id).Load();

Console.WriteLine($"Company: {company.Name}");

foreach (var u in company.Users)

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

}

}

Выражение db.Users.Where(p=>p.CompanyId==company.Id).Load() загружает пользователей в контекст. Подвыражение Where(p=>p.CompanyId==company.Id) означает, что загружаются только те пользователи, у которых свойство CompanyId соответствует свойству Id ранее полученной компании

Консольный вывод программы:



Для загрузки связанных данных мы также можем использовать методы **Collection()** и **Reference**. Метод **Collection** применяется, если навигационное свойство представляет коллекцию:

using (ApplicationContext db = new ApplicationContext())

{

Company? company = db.Companies.FirstOrDefault();

if (company != null)

{

db.Users.Where(u => u.CompanyId == company.Id).Load();

Console.WriteLine($"Company: {company.Name}");

foreach (var u in company.Users)

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

}

}

На уровне базы данных вызов db.Entry(company).Collection(t=>t.Users).Load() будет трансформироваться в следующую команду SQL:

|  |  |
| --- | --- |
| 1  2  3 | SELECT "u"."Id", "u"."CompanyId", "u"."Name"  FROM "Users" AS "u"  WHERE "u"."CompanyId" = @\_\_p\_0 |

@\_\_p\_0 в данном случае это параметр, который представляет id компании и который автоматически передается инфаструктурой EF Core.

Если навигационное свойство представляет одиночный объект, то можно применять метод **Reference**:

using (ApplicationContext db = new ApplicationContext())

{

User? user = db.Users.FirstOrDefault(); // получаем первого пользователя

if (user != null)

{

db.Entry(user).Reference(u => u.Company).Load();

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

}

}

На уровне базы данных этот запрос будет транслироваться в следующую команду sql:

|  |  |
| --- | --- |
| 1  2  3 | SELECT "c"."Id", "c"."Name"  FROM "Companies" AS "c"  WHERE "c"."Id" = @\_\_p\_0 |

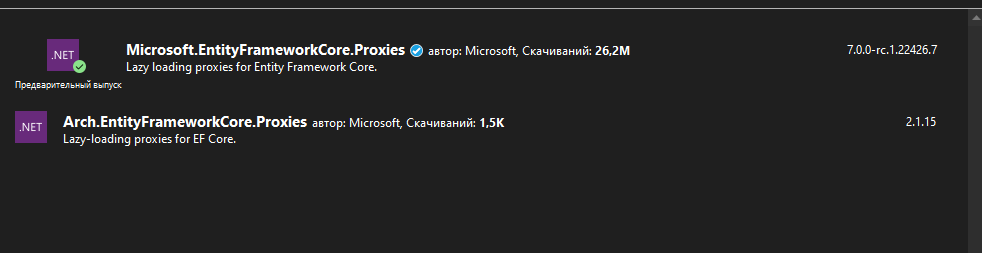
где @\_\_p\_0 в данном случае это автоматически передаваемый параметр, который представляет свойство CompanyId выбранного пользователя.

## **Lazy loading**

**Lazy loading** или ленивая загрузка предполагает неявную автоматическую загрузку связанных данных при обращении к навигационному свойству. Однако здесь есть ряд условий:

* При конфигурации контекста данных вызвать метод **UseLazyLoadingProxies()**
* Все навигационные свойства должны быть определены как виртуальные (то есть с модификатором **virtual**), при этом сами классы моделей должны быть открыты для наследования

Используем lazy loading. Прежде всего добавим в проект через nuget пакет **Microsoft.EntityFrameworkCore.Proxies**.



Далее определим следующие модели и контекст данных:

public class Company

{

public int Id { get; set; }

public string? Name { get; set; }

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

}

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public int? CompanyId { get; set; }

public virtual Company? Company { get; set; }

}

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

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

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder

.UseLazyLoadingProxies() // подключение lazy loading

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

}

Прежде всего в методе OnConfiguring у объекта DbContextOptionsBuilder вызывается метод **UseLazyLoadingProxies()**, который делает доступной ленивую загрузку.

Также навигационное свойство Users в классе Company и навигационное свойство Company в классе User определены как виртуальные, то есть имеют модификатор **virtual**.

После этого мы можем загрузить пользователей и связанные с ними компании следующим образом:

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();

}

using (ApplicationContext db = new ApplicationContext())

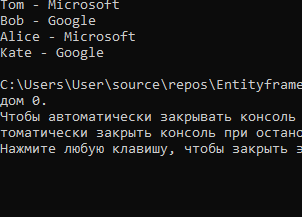
{

var users = db.Users.ToList();

foreach (User user in users)

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

}



Теперь посмотрим, что происходит на уровне базы данныx. Вначале получаем пользователей:

var users = db.Users.ToList();

В базе данных выполняется sql-команда:

|  |  |
| --- | --- |
| 1  2 | SELECT "u"."Id", "u"."CompanyId", "u"."Name"  FROM "Users" AS "u" |

Далее в цикле перебираем всех полученных пользователей и обращаемся к навигационному свойству Company для получения связанной компании:

foreach (User user in users)

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

Поскольку идет обращение к навигационному свойству Company, то автоматически подтягиваются связанные с ним данные - объекты Company. В данном случае у нас выше было добавлено только 2 компании и обе эти компании ссвязанные с перебираемыми пользователями: два пользователя связаны с одной компанией, а два других пользователя - с другой. Поэтому будут выполняться два запроса:

|  |  |
| --- | --- |
| 1  2  3 | SELECT "c"."Id", "c"."Name"  FROM "Companies" AS "c"  WHERE "c"."Id" = @\_\_p\_0 |

@\_\_p\_0 в данном случае это параметр, который хранит значения свойства CompanyId пользователя, для которого надо получить компанию.

Перебираются четыре пользователя, но выполняются только два запроса, так как после того как объект загружен в контекст, в дальнейшем он берется из контекста.

То есть если совместить консольный вывод и выполняемые выражения SQL, то получится следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | // получение всех пользователей  SELECT "u"."Id", "u"."CompanyId", "u"."Name"  FROM "Users" AS "u"  // идет обращение к свойству Company, его компании нет в контексте  // поэтому выполняется sql-запрос  SELECT "c"."Id", "c"."Name"  FROM "Companies" AS "c"  WHERE "c"."Id" = @\_\_p\_0  Tom - Microsoft  // компания этого пользователя уже в контексте, не надо выполнять новый запрос  Alice - Microsoft  // компании следующего пользователя нет в контексте  // поэтому выполняется sql-запрос  SELECT "c"."Id", "c"."Name"  FROM "Companies" AS "c"  WHERE "c"."Id" = @\_\_p\_0  Bob - Google  // компания этого пользователя уже в контексте, не надо выполнять новый запрос  Kate - Google |

Таким же образом можно загрузить компании и связанных с ними пользователей:

using (ApplicationContext db = new ApplicationContext())

{

var companies = db.Companies.ToList();

foreach (Company company in companies)

{

Console.Write($"{company.Name}:");

foreach (User user in company.Users)

Console.Write($"{user.Name} ");

Console.WriteLine();

}

}

## **Отношение один к одному**

Отношение один к одному предполагает, что главная сущность может ссылаться только на один объект зависимой сущности. В свою очередь, зависимая сущность может ссылаться только на один объект главной сущности.

Рассмотрим стандартный пример подобных отношений: есть класс пользователя User, который хранит логин и пароль, то есть данные учетных записей. А все данные профиля, такие как имя, возраст и так далее, выделяются в класс профиля UserProfile.

public class User

{

public int Id { get; set; }

public string? Login { get; set; }

public string? Password { get; set; }

public UserProfile? Profile { get; set; }

}

public class UserProfile

{

public int Id { get; set; }

public string? Name { get; set; }

public int Age { get; set; }

public int UserId { get; set; }

public User? User { get; set; }

## }

public class ApplicationContext : DbContext

{

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

public DbSet<UserProfile> UserProfiles { get; set; } = null!;

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

}

## }

 этой связи между классами сущность UserProfile является зависимой по отношению к сущности User. И чтобы установить связь один к одному, у зависимой сущности устанавливается свойство внешний ключ: public int UserId { get; set; }. Благодаря этому Entity Framework узнает, что UserProfile является зависимой сущностью. К примеру, в классе User также есть навигационное свойство - ссылка на объект UserProfile, но при этом внешний ключ отсутствует.

В итоге для сущности UserProfile в случае с SQLite будет создана следующая таблица в базе данных:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | CREATE TABLE "UserProfiles" (      "Id"    INTEGER NOT NULL,      "Name"  TEXT,      "Age"   INTEGER NOT NULL,      "UserId"    INTEGER NOT NULL,      CONSTRAINT "PK\_UserProfiles" PRIMARY KEY("Id" AUTOINCREMENT),      CONSTRAINT "FK\_UserProfiles\_Users\_UserId" FOREIGN KEY("UserId") REFERENCES "Users"("Id") ON DELETE CASCADE  ); |

Внешне эта таблица не отличается от таблицы, которая создается для зависимой сущности при связи один-ко-многим. Однако также стоит добавить, что для этой таблицы для столбца, который представляет внешний ключ (в данном случае UserId), создается уникальный индекс. И этот индекс гарантирует, что только одна зависимая сущность (здесь UserProfile) может быть связана с одной главной сущностью (здесь сущность User):

|  |  |
| --- | --- |
| 1 | CREATE UNIQUE INDEX "IX\_UserProfiles\_UserId" ON "UserProfiles" ("UserId") |

Посмотрим, как работать с моделями с такой связью. Добавление:

using(ApplicationContext db = new ApplicationContext())

{

// пересоздадим базу данных

db.Database.EnsureDeleted();

db.Database.EnsureCreated();

User user1 = new User { Login = "login1", Password = "pass1234" };

User user2 = new User { Login = "login2", Password = "5678word2" };

db.Users.AddRange(user1, user2);

UserProfile profile1 = new UserProfile { Age = 22, Name = "Tom", User = user1 };

UserProfile profile2 = new UserProfile { Age = 27, Name = "Alice", User = user2 };

db.UserProfiles.AddRange(profile1, profile2);

db.SaveChanges();

## }

## Получение данных:

using (ApplicationContext db = new ApplicationContext())

{

foreach (User user in db.Users.Include(u => u.Profile).ToList())

{

Console.WriteLine($"Name: {user.Profile?.Name} Age: {user.Profile?.Age}");

Console.WriteLine($"Login: { user.Login} Password: { user.Password} \n");

}

## }

## Редактирование:

using (ApplicationContext db = new ApplicationContext())

{

User? user = db.Users.FirstOrDefault();

// получаем первый объект User

if (user != null)

{

user.Password = "dsfvbggg";

db.SaveChanges();

}

// получаем объект UserProfile для пользователя с логином "login2"

UserProfile? profile = db.UserProfiles.FirstOrDefault(p => p.User.Login == "login2");

if (profile != null)

{

profile.Name = "Alice II";

db.SaveChanges();

}

}

При удалении надо учитывать следующее: так как объект UserProfile требует наличие объекта User и зависит от этого объекта, то при удалении связанного объекта User также будет удален и связанный с ним объект UserProfile. Если же будет удален объект UserProfile, на объект User это никак не повлияет:

using (ApplicationContext db = new ApplicationContext())

{

// удаляем первый объект User

User? user = db.Users.FirstOrDefault();

if (user != null)

{

db.Users.Remove(user);

db.SaveChanges();

}

// удаляем объект UserProfile c логином login2

UserProfile? profile = db.UserProfiles.FirstOrDefault(p => p.User.Login == "login2");

if (profile != null)

{

db.UserProfiles.Remove(profile);

db.SaveChanges();

}

}

### **Объединение таблиц**

Entity Framework Core позволяет хранить данные сущностей, которые связаны отношением один-к-одному, в одной таблице. Например, возьмем те же модели User и UserProfile и определим для них одну таблицу Users:

public class User

{

public int Id { get; set; }

public string? Login { get; set; }

public string? Password { get; set; }

public UserProfile? Profile { get; set; }

}

public class UserProfile

{

public int Id { get; set; }

public string? Name { get; set; }

public int Age { get; set; }

public User? User { get; set; }

}

public class ApplicationContext : DbContext

{

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

public DbSet<UserProfile> UserProfiles { get; set; } = null!;

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<User>()

.HasOne(u => u.Profile).WithOne(p => p.User)

.HasForeignKey<UserProfile>(up => up.Id);

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

modelBuilder.Entity<UserProfile>().ToTable("Users");

}

}

В этом случае в БД SQLite будет создаваться одна таблица Users:

CREATE TABLE "Users" (

    "Id"    INTEGER NOT NULL,

    "Login" TEXT,

    "Password"  TEXT,

    "Name"  TEXT,

    "Age"   INTEGER,

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

);

Например, добавление и получение обеих моделей

using (ApplicationContext db = new ApplicationContext())

{

// пересоздадим базу данных

db.Database.EnsureDeleted();

db.Database.EnsureCreated();

User user1 = new User { Login = "login1", Password = "pass1234" };

User user2 = new User { Login = "login2", Password = "5678word2" };

db.Users.AddRange(user1, user2);

UserProfile profile1 = new UserProfile { Age = 22, Name = "Tom", User = user1 };

UserProfile profile2 = new UserProfile { Age = 27, Name = "Alice", User = user2 };

db.UserProfiles.AddRange(profile1, profile2);

db.SaveChanges();

}

using (ApplicationContext db = new ApplicationContext())

{

// получим данные

foreach (var u in db.Users.Include(u => u.Profile).ToList())

{

Console.WriteLine($"Name: {u.Profile?.Name} Age: {u.Profile?.Age}");

Console.WriteLine($"Login: { u.Login} Password: { u.Password} \n");

}

}

## **Отношение один ко многим**

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public int CompanyId { get; set; }

public Company? Company { get; set; } // компания пользователя

}

public class Company

{

public int Id { get; set; }

public string? Name { get; set; }

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

}

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

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

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

}

Добавление данных:

using (ApplicationContext db = new ApplicationContext())

{

// пересоздадим базу данных

db.Database.EnsureDeleted();

db.Database.EnsureCreated();

// создание и добавление моделей

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 = microsoft };

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

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

db.SaveChanges();

}

Получение данных:

using (ApplicationContext db = new ApplicationContext())

{

// вывод пользователей

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

foreach (User user in users)

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

// вывод компаний

var companies = db.Companies.Include(c => c.Users).ToList();

foreach (Company comp in companies)

{

Console.WriteLine($"\n Компания: {comp.Name}");

foreach (User user in comp.Users)

{

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

}

}

}

Редактирование:

using (ApplicationContext db = new ApplicationContext())

{

// изменение имени пользователя

User? user1 = db.Users.FirstOrDefault(p => p.Name == "Tom");

if (user1 != null)

{

user1.Name = "Tomek";

db.SaveChanges();

}

// изменение названия компании

Company? comp = db.Companies.FirstOrDefault(p => p.Name == "Google");

if (comp != null)

{

comp.Name = "Alphabet";

db.SaveChanges();

}

// смена компании сотрудника

User? user2 = db.Users.FirstOrDefault(p => p.Name == "Bob");

if (user2 != null && comp != null)

{

user2.Company = comp;

db.SaveChanges();

}

}

Удаление:

using (ApplicationContext db = new ApplicationContext())

{

User? user = db.Users.FirstOrDefault(u => u.Name == "Bob");

if (user != null)

{

db.Users.Remove(user);

db.SaveChanges();

}

Company? comp = db.Companies.FirstOrDefault();

if (comp != null)

{

db.Companies.Remove(comp);

db.SaveChanges();

}

}

Следует учитывать, что если зависимая сущность (в данном случае User) требует обязательного наличия главной сущности (в данном случае Company), то на уровне базы данных при удалении главной сущности с помощью каскадного удаления будут удалены и связанные с ней зависимые сущности. Так, в данном случае для объекта User установлено обязательное наличие объекта Company:

public int CompanyId { get; set; }

public Company? Company { get; set; }

## **Отношение многие ко многим**

Отношение **многие-ко-многим** (many-to-many) представляет связь, при которой объект одной сущности может ссылаться на множество объектов второй сущности, а объект второй сущности, в свою очередь, может ссылаться на множество объектов первой сущности. Примером подобного отношения может служить посещение студентами университетских курсов.

Например, определим следующие классы моделей и контекста:

public class Course

{

public int Id { get; set; }

public string? Name { get; set; }

public List<Student> Students { get; set; } = new();

}

public class Student

{

public int Id { get; set; }

public string? Name { get; set; }

public List<Course> Courses { get; set; } = new();

}

public class ApplicationContext : DbContext

{

    public DbSet<Course> Courses { get; set; } = null!;

    public DbSet<Student> Students { get; set; } = null!;

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

    {

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

    }

}

В данном случае все студенты, поступившие на курс, будут помещаться в свойство Students класса Course. Аналогично, все курсы студента будут храниться в свойстве Courses класса Student. То есть стандартная связь многие ко многие. Однако, при создании базы данных в ней будет три таблицы. Например, в случае с SQLite они будут выглядеть следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | CREATE TABLE "Courses" (      "Id"    INTEGER NOT NULL,      "Name"  TEXT,      CONSTRAINT "PK\_Courses" PRIMARY KEY("Id" AUTOINCREMENT)  );    CREATE TABLE "Students" (      "Id"    INTEGER NOT NULL,      "Name"  TEXT,      CONSTRAINT "PK\_Students" PRIMARY KEY("Id" AUTOINCREMENT)  );    CREATE TABLE "CourseStudent" (      "CoursesId" INTEGER NOT NULL,      "StudentsId"    INTEGER NOT NULL,      CONSTRAINT "FK\_CourseStudent\_Courses\_CoursesId" FOREIGN KEY("CoursesId") REFERENCES "Courses"("Id") ON DELETE CASCADE,      CONSTRAINT "FK\_CourseStudent\_Students\_StudentsId" FOREIGN KEY("StudentsId") REFERENCES "Students"("Id") ON DELETE CASCADE,      CONSTRAINT "PK\_CourseStudent" PRIMARY KEY("CoursesId","StudentsId")  ); |

То есть в реальности на уровне базы данных создается промежуточная таблица, которая хранит связи между студентами и курсами. Тем не менее на уровне кода C# нам не надо создавать промежуточную сущность, Entity Framework Core начиная с версии 5.0 умеет управлять подобной связью.

Рассмотрим, как мы можем работать с моделями в связи многие ко многим. Добавление:

using (ApplicationContext db = new ApplicationContext())

{

// пересоздадим базу данных

db.Database.EnsureDeleted();

db.Database.EnsureCreated();

// создание и добавление моделей

Student tom = new Student { Name = "Tom" };

Student alice = new Student { Name = "Alice" };

Student bob = new Student { Name = "Bob" };

db.Students.AddRange(tom, alice, bob);

Course algorithms = new Course { Name = "Algoritmid" };

Course basics = new Course { Name = "Programmeerimise alused" };

db.Courses.AddRange(algorithms, basics);

// добавляем к студентам курсы

tom.Courses.Add(algorithms);

tom.Courses.Add(basics);

alice.Courses.Add(algorithms);

bob.Courses.Add(basics);

db.SaveChanges();

}

Стоит отметить, что здесь мы добавляем курсы к студентам, но также можем сделать и наоборот - добавить студентов к курсам:

algorithms.Students.AddRange(new List<Student>() { tom, bob });

Вывод данных:

using (ApplicationContext db = new ApplicationContext())

{

var courses = db.Courses.Include(c => c.Students).ToList();

// выводим все курсы

foreach (var c in courses)

{

Console.WriteLine($"Course: {c.Name}");

// выводим всех студентов для данного кура

foreach (Student s in c.Students)

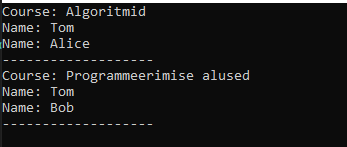
Console.WriteLine($"Name: {s.Name}");

Console.WriteLine("-------------------");

}

}

Консольный вывод:



Обновление данных (например, удалим у студента один курс и добавим другой)

using (ApplicationContext db = new ApplicationContext())

{

Student? alice = db.Students.Include(s => s.Courses).FirstOrDefault(s => s.Name == "Alice");

Course? algorithms = db.Courses.FirstOrDefault(c => c.Name == "Algoritmid");

Course? basics = db.Courses.FirstOrDefault(c => c.Name == "Programmerinise alused");

if (alice != null && algorithms != null && basics != null)

{

// удаление курса у студента

alice.Courses.Remove(algorithms);

// добавление нового курса студенту

alice.Courses.Add(basics);

db.SaveChanges();

}

}

Удаление же студента или курса из базы данных приведет к тому, что все строки из промежуточной таблицы, которые связаны с удаляемым объектом, также будут удалены:

Student? student = db.Students.FirstOrDefault();

db.Students.Remove(student);

db.SaveChanges();

### Конфигурация связи

EF Core позволяет сконфигурировать отношение многие ко многие. Обычно подобная конфигурация требуется для настройки промежуточной таблицы. Например, мы хотим переопределить название таблицы, которая создается для промежуточной сущности.

public class ApplicationContext : DbContext

{

public DbSet<Course> Courses { get; set; } = null!;

public DbSet<Student> Students { get; set; } = null!;

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Course>()

.HasMany(c => c.Students)

.WithMany(s => s.Courses)

.UsingEntity(j => j.ToTable("Enrollments"));

}

}

Последний метод в цепочке - **UsingEntity** позволяет настроить промежуточную таблицу. Фактически объект, представленный буквой **j** как раз представляет условную промежуточную сущность, для которой создается таблица. Так, в данном случае промежуточная таблица будет называться "Enrollments".

### Добавление столбцов в промежуточную таблицу

EF Core автоматически создает промежуточную таблицу с двумя столбцами, через которые она связана с двумя другими таблицами. Однако иногда может потребоваться добавить в промежуточную таблицу еще какие-то данные. Например, в случае со студентами и курсами мы бы могли хранить в промежуточной таблице также дату поступления студента на выбранный курс. В этом случае на уровне кода C# лучше создать помежуточную сущность, которая будет содержать описание данных, которые мы хотим определить. Например:

public class Course

{

public int Id { get; set; }

public string? Name { get; set; }

public List<Student> Students { get; set; } = new();

public List<Enrollment> Enrollments { get; set; } = new();

}

public class Student

{

public int Id { get; set; }

public string? Name { get; set; }

public List<Course> Courses { get; set; } = new();

public List<Enrollment> Enrollments { get; set; } = new();

}

public class Enrollment

{

public int StudentId { get; set; }

public Student? Student { get; set; }

public int CourseId { get; set; }

public Course? Course { get; set; }

public int Mark { get; set; } // оценка студента

}

Для определения данных, которые будут храниться в промежуточной таблице, здесь определена промежуточная сущность Enrollment, которая содержит навигационные свойства на сущности Student и Course, а также содержит дополнительное свойство Mark (оценка студента).

Для настройки связи определим следующий контекст данных:

public class ApplicationContext : DbContext

{

public DbSet<Course> Courses { get; set; } = null!;

public DbSet<Student> Students { get; set; } = null!;

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder

.Entity<Course>()

.HasMany(c => c.Students)

.WithMany(s => s.Courses)

.UsingEntity<Enrollment>(

j => j

.HasOne(pt => pt.Student)

.WithMany(t => t.Enrollments)

.HasForeignKey(pt => pt.StudentId),

j => j

.HasOne(pt => pt.Course)

.WithMany(p => p.Enrollments)

.HasForeignKey(pt => pt.CourseId),

j =>

{

j.Property(pt => pt.Mark).HasDefaultValue(3);

j.HasKey(t => new { t.CourseId, t.StudentId });

j.ToTable("Enrollments");

});

}

}

Для конфигурации промежуточной сущности Enrollment также используется метод UsingEntity>Enrollment<(). Вначале настраиваются внешние ключи таблиц:

j => j

.HasOne(pt => pt.Student)

.WithMany(t => t.Enrollments)

.HasForeignKey(pt => pt.StudentId),

j => j

.HasOne(pt => pt.Course)

.WithMany(p => p.Enrollments)

.HasForeignKey(pt => pt.CourseId),

j =>

В последней части настраиваем свойства сущности Enrollment, а также имя соответствующей таблицы и ее ключи:

j =>

{

j.Property(pt => pt.Mark).HasDefaultValue(3);

j.HasKey(t => new { t.CourseId, t.StudentId });

j.ToTable("Enrollments");

});

столбца Mark устанавливает значение по умолчанию - число 3.

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | CREATE TABLE "Enrollments" (      "StudentId" INTEGER NOT NULL,      "CourseId"  INTEGER NOT NULL,      "Mark"  INTEGER NOT NULL DEFAULT 3,      CONSTRAINT "FK\_Enrollments\_Courses\_CourseId" FOREIGN KEY("CourseId") REFERENCES "Courses"("Id") ON DELETE CASCADE,      CONSTRAINT "FK\_Enrollments\_Students\_StudentId" FOREIGN KEY("StudentId") REFERENCES "Students"("Id") ON DELETE CASCADE,      CONSTRAINT "PK\_Enrollments" PRIMARY KEY("CourseId","StudentId")  ); |

Теперь рассмотрим некоторые операции. Добавление

using (ApplicationContext db = new ApplicationContext())

{

// пересоздадим базу данных

db.Database.EnsureDeleted();

db.Database.EnsureCreated();

// создание и добавление моделей

Student tom = new Student { Name = "Tom" };

Student alice = new Student { Name = "Alice" };

Student bob = new Student { Name = "Bob" };

db.Students.AddRange(tom, alice, bob);

Course algorithms = new Course { Name = "ALgoritmid" };

Course basics = new Course { Name = "Programmerimise alused" };

db.Courses.AddRange(algorithms, basics);

// добавляем к студентам курсы

tom.Enrollments.Add(new Enrollment { Course = algorithms });

tom.Courses.Add(basics);

alice.Enrollments.Add(new Enrollment { Course = algorithms, Mark = 4 });

bob.Enrollments.Add(new Enrollment { Course = basics });

db.SaveChanges();

}

Особенностью работы с данными в данном случае будет то, что мы можем использовать два способа для установки связи одной сущности с другой:

|  |  |
| --- | --- |
| 1  2 | tom.Enrollments.Add(new Enrollment { Course = algorithms, EnrollmentDate = DateTime.Now });  tom.Courses.Add(basics); |

В первом случае устанавливаем связь студента с курсом через добавление объекта Enrollment, который обеспечивает эту связь. Во втором случае добавляем курс напрямую в коллекцию курсов студента, что в сою очередь также автоматически добавить данные в связующую таблицу Enrollments.

Вывод данных:

using (ApplicationContext db = new ApplicationContext())

{

var courses = db.Courses.Include(c => c.Students).ToList();

// выводим все курсы

foreach (var c in courses)

{

Console.WriteLine($"Course: {c.Name}");

// выводим всех студентов для данного кура

foreach (Student s in c.Students)

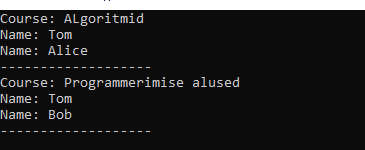
Console.WriteLine($"Name: {s.Name}");

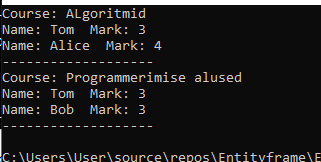
Console.WriteLine("-------------------");

}

}

Консольный вывод:





## **Комплексные типы**

Обычно Entity Framework Core сопоставляет каждый класс с отдельной таблицей. Однако бывают случаи, когда определенный класс существует не сам по себе, а несет некоторую дополнительную информацию по отношению к другой главной модели. Данная функциональность предполагает, что у нас есть несколько типов, но один из них является владельцем (owner). Другие же зависимые или собственные типы (owned entity types) являются частью типа-владельца и без него существовать не могут.

Для создания связки "главный тип - зависимые типы" можно использовать либо атрибут **OwnedAttribute**, либо настройки Fluent API.

### OwnedAttribute

Атрибут **OwnedAttribute** позволяет установить зависимый тип. Например:

public class User

{

public int Id { get; set; }

public string? Login { get; set; }

public string? Password { get; set; }

public UserProfile? Profile { get; set; }

}

[Owned]

public class UserProfile

{

public string? Name { get; set; }

public int Age { get; set; }

## }

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");

}

## }

В данном случае есть тип-владелец User и есть владеемый им тип UserProfile. Класс User определяет основные настройки учетной записи - логин и пароль, а UserProfile - всторостепенные данные - имя, возраст и т.д. Чтобы указать, что User владеет типом UserProfile, над UserProfile указывается атрибут OwnedAttribute. Кроме того, UserProfile не содержит ключа.

В итоге при создании базы данных данные обоих типов будут содержаться в одной таблице:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | CREATE TABLE "Users" (      "Id"    INTEGER NOT NULL,      "Login" TEXT,      "Password"  TEXT,      "Profile\_Name"  TEXT,      "Profile\_Age"   INTEGER,      CONSTRAINT "PK\_Users" PRIMARY KEY("Id" AUTOINCREMENT)  ); |

Использование типов:

using (ApplicationContext db = new ApplicationContext())

{

User user1 = new User

{

Login = "login1",

Password = "pass1234",

Profile = new UserProfile { Age = 23, Name = "Tom" }

};

User user2 = new User

{

Login = "login2",

Password = "5678word2",

Profile = new UserProfile { Age = 27, Name = "Alice" }

};

db.Users.AddRange(user1, user2);

db.SaveChanges();

// добавление данных

var users = db.Users.ToList();

foreach (User u in users)

Console.WriteLine($"Name: {u.Profile?.Name} Age: {u.Profile?.Age} Login: { u.Login} Password: { u.Password} ");

## }

### **Fluent API**

Также для настройки связи можно использовать Fluent API, в частности, метод **OwnsOne()**:

public class User

{

public int Id { get; set; }

public string? Login { get; set; }

public string? Password { get; set; }

public UserProfile? Profile { get; set; }

}

[Owned]

public class UserProfile

{

public string? Name { get; set; }

public int Age { get; set; }

}

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>().OwnsOne(u => u.Profile);

}

}

В методе OwnsOne() указывается навигационное свойство, которое представляет зависимый тип.

### **Приватные типы**

Вполне возможно, что мы захотим сделать навигационное свойство Profile приватным. В этом случае понадобится дополнительная настройка:

public class User

{

private User() { }

public User(string login, string password, UserProfile profile)

{

Login = login; Password = password; Profile = profile;

}

public int Id { get; set; }

public string? Login { get; set; }

public string? Password { get; set; }

private UserProfile? Profile { get; set; }

public override string ToString()

{

return $"Name: {Profile?.Name} Age: {Profile?.Age} Login: {Login} Password: {Password}";

}

}

public class UserProfile

{

public string? Name { get; set; }

public int Age { get; set; }

## }

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>().OwnsOne(typeof(UserProfile), "Profile");

}

## }

## В этом случае в метод OwnsOne передается тип навигационного свойства и само имя навигационного свойства. Применение:

using (ApplicationContext db = new ApplicationContext())

{

User user1 = new User("login1", "pass1234", new UserProfile { Age = 23, Name = "Tom" });

User user2 = new User("login2", "5678word2", new UserProfile { Age = 27, Name = "Alice" });

db.Users.AddRange(user1, user2);

db.SaveChanges();

var users = db.Users.ToList();

foreach (User user in users)

Console.WriteLine(user.ToString());

## }

### **Вложенные собственные типы**

Одни собственные типы могут иметь другие собственные типы, увеличивая сложность структуры классов. Например, определим следующие классы:

public class User

{

public int Id { get; set; }

public string? Login { get; set; }

public string? Password { get; set; }

[Required]

public UserProfile? Profile { get; set; }

}

public class UserProfile

{

public Claim? Name { get; set; }

public Claim? Age { get; set; }

}

public class Claim

{

public string? Key { get; set; }

public string? Value { get; set; }

}

Главной сущностью здесь является класс User, который содержит объект класса UserProfile. Класс UserProfile, в свою очередь, также содержит объекты еще одного класса - Claim, который представляет отдельные данные по принципу ключ-значение. Обратите внимание, что свойство Profile является **обязательным**.

Теперь определим контекст данных:

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>().OwnsOne(u => u.Profile, p =>

{

p.OwnsOne(c => c.Name);

p.OwnsOne(c => c.Age);

});

}

## }

С помощью метода **OwnsOne()** устанавливается включение через свойства собственных типов. И после создания базы данных мы получим следующую таблицу User, которая будет содержать данные, представленные собственными типами.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | CREATE TABLE "Users" (      "Id"    INTEGER NOT NULL,      "Login" TEXT,      "Password"  TEXT,      "Profile\_Name\_Key"  TEXT,      "Profile\_Name\_Value"    TEXT,      "Profile\_Age\_Key"   TEXT,      "Profile\_Age\_Value" TEXT,      CONSTRAINT "PK\_Users" PRIMARY KEY("Id" AUTOINCREMENT)  ); |

Для работы с собственными типами определим следующий код:

using (ApplicationContext db = new ApplicationContext())

{

User user1 = new User

{

Login = "login1",

Password = "pass1234",

Profile = new UserProfile

{

Age = new Claim { Key = "Age", Value = "23" },

Name = new Claim { Key = "Name", Value = "Tom" }

}

};

User user2 = new User

{

Login = "login2",

Password = "5678word2",

Profile = new UserProfile

{

Age = new Claim { Key = "Age", Value = "27" },

Name = new Claim { Key = "Name", Value = "Alice" }

}

};

db.Users.AddRange(user1, user2);

db.SaveChanges();

var users = db.Users.ToList();

foreach (User user in users)

Console.WriteLine($"Name: {user.Profile?.Name?.Value} Age: {user.Profile?.Age?.Value} Login: {user.Login} Password: {user.Password}");

## }

**Иерархические данные**

Некоторые данные представляют иерархическую структуру по типу деревьев, например, меню, где один пункт меню может иметь подменю. Или файловая система, где в каталоге могут быть другие каталоги. Рассмотрим, как в Entity Framework мы можем создать и использовать подобную иерархическую структуру.

Определим следующую сущность MenuItem:

public class MenuItem

{

public int Id { get; set; }

public string? Title { get; set; }

public int? ParentId { get; set; }

public MenuItem? Parent { get; set; }

public List<MenuItem> Children { get; set; } = new();

}

MenuItem представляет пункт меню, который может иметь подменю. Свойство Title представляет некоторый текст пункта меню. Свойство Parent хранит ссылку на родительский пункт меню при его наличии, а ParentId - id родительского пункта меню. Причем поскольку у пунктов меню верхнего уровня не может быть родительских меню, то свойства ParentId и Parent определены как необязательные. А свойство Children представляет пункты подменю.

И определим следующий контекст данных:

public DbSet<MenuItem> MenuItems { get; set; } = null!;

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

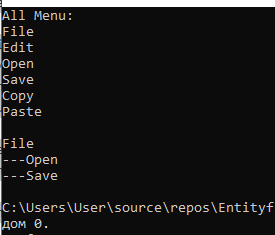
optionsBuilder.UseSqlite("Data Source=helloapp.db");

}

В итоге при создании базы данных будет генерироваться следующая таблица:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | CREATE TABLE "MenuItems" (      "Id"    INTEGER NOT NULL,      "Title" TEXT,      "ParentId"  INTEGER,      CONSTRAINT "FK\_MenuItems\_MenuItems\_ParentId" FOREIGN KEY("ParentId") REFERENCES "MenuItems"("Id"),      CONSTRAINT "PK\_MenuItems" PRIMARY KEY("Id" AUTOINCREMENT)  ); |

Использование MenuItem:



# **ГЛАВА 5**

# Наследование

## Подход TPH - Table Per Hierarchy

По умолчанию при работе с цепочками наследования классов Entity Framework Core использует подход TPH (Table Per Hierarchy / Таблица на одну иерархию классов). При использовании данного подхода TPH для всех классов из одной иерархии в базе данных создается одна таблица. А чтобы определить, к какому именно классу относится строка в таблице, в этой же таблице создается дополнительный столбец - дискриминатор.

Например, у нас есть следующая иерархия классов:

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

}

public class Employee : User

{

public int Salary { get; set; }

}

public class Manager : User

{

public string? Departament { get; set; }

}

Есть базовый класс User, который представляет пользователя и от которого наследуются класс Employee - класс работника и Manager - класс управляющего.

Определим контекст данных:

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

public DbSet<Employee> Employees { get; set; } = null!;

public DbSet<Manager> Managers { get; set; } = null!;

public ApplicationContext()

{

Database.EnsureDeleted();

Database.EnsureCreated();

}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

}

Чтобы включить все классы из иерархии наследования в базу данных, в контексте данных для каждого типа должен быть определен набор DbSet.

Сгенерированная база данных будет содержать для всех типов одну таблицу Users. Кроме всех свойств классов User, Employee и Manager здесь также появляется еще один столбец - Discriminator. Он имеет тип nvarchar (то есть строка), а в качестве значения он принимает название класса, к которому относится строка в таблице. В итоге в бд будет создаваться следующая таблица:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | CREATE TABLE "Users" (      "Id"    INTEGER NOT NULL,      "Name"  TEXT,      "Discriminator" TEXT NOT NULL,      "Salary"    INTEGER,      "Departament"   TEXT,      CONSTRAINT "PK\_Users" PRIMARY KEY("Id" AUTOINCREMENT)  ); |

Пример использования:

using (ApplicationContext db = new ApplicationContext())

{

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

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

db.Users.Add(user1);

db.Users.Add(user2);

Employee employee = new Employee { Name = "Sam", Salary = 500 };

db.Employees.Add(employee);

Manager manager = new Manager { Name = "Robert", Departament = "IT" };

db.Managers.Add(manager);

db.SaveChanges();

var users = db.Users.ToList();

Console.WriteLine("Koik kasutajad");

foreach (var user in users)

{

Console.WriteLine(user.Name);

}

Console.WriteLine("\nKoik tootajad");

foreach (var emp in db.Employees.ToList())

{

Console.WriteLine(emp.Name);

}

Console.WriteLine("\nKoid juhid");

foreach (var man in db.Managers.ToList())

{

Console.WriteLine(man.Name);

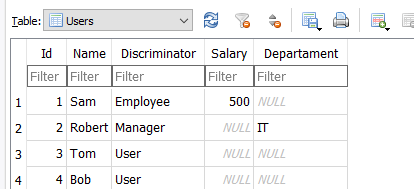
}

}

Консольный вывод:



И в итоге данные таблицы Users в бд будут выглядеть следующим образом:



При необходимости мы даже можем добавить в класс User свойство Discriminator:

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public string? Discriminator { get; set; }

}

## Подход TPT - Table Per Type

В прошлой статье был рассмотрен подход TPH - одна таблица для всей иерархии наследования классов. Но также EntityFramework Core позволяет использовать другой подход - TPT или Table Per Type, который предполагает создание отдельной таблицы для каждого класса из иерархии. Для реализации подхода TPT можно использовать два способа: атрибуты или Fluent API.

### Применение TPT на основе атрибутов

С помощью атрибута **[Table]** мы можем указать для каждого класса свою таблицу:

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

}

[Table("Employees")]

public class Employee : User

{

public int Salary { get; set; }

}

[Table("Managers")]

public class Manager : User

{

public string? Departament { get; set; }

}

public class ApplicationContext : DbContext

{

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

public DbSet<Employee> Employees { get; set; } = null!;

public DbSet<Manager> Managers { get; set; } = null!;

public ApplicationContext()

{

Database.EnsureDeleted();

Database.EnsureCreated();

}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

}

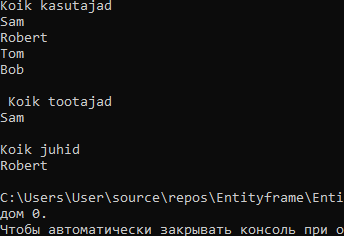
}

В этом случае в бд будут создаваться следующие три таблицы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | CREATE TABLE "Users" (      "Id"    INTEGER NOT NULL,      "Name"  TEXT,      CONSTRAINT "PK\_Users" PRIMARY KEY("Id" AUTOINCREMENT)  );  CREATE TABLE "Employees" (      "Id"    INTEGER NOT NULL,      "Salary"    INTEGER NOT NULL,      CONSTRAINT "FK\_Employees\_Users\_Id" FOREIGN KEY("Id") REFERENCES "Users"("Id") ON DELETE CASCADE,      CONSTRAINT "PK\_Employees" PRIMARY KEY("Id" AUTOINCREMENT)  );  CREATE TABLE "Managers" (      "Id"    INTEGER NOT NULL,      "Departament"   TEXT,      CONSTRAINT "FK\_Managers\_Users\_Id" FOREIGN KEY("Id") REFERENCES "Users"("Id") ON DELETE CASCADE,      CONSTRAINT "PK\_Managers" PRIMARY KEY("Id" AUTOINCREMENT)  ); |

Здесь мы видим, что все свойства базового класса User будут храниться в одной таблице, а те данные, которые относятся только к производным классам, хранятся в отдельных таблицах.

Пример использования:



### Применение TPT на основе Fluent API

Также мы можем настроить TPT с помощью метода **ToTable()** во Fluent API:

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

}

public class Employee : User

{

public int Salary { get; set; }

}

public class Manager : User

{

public string? Departament { get; set; }

}

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

public DbSet<Employee> Employees { get; set; } = null!;

public DbSet<Manager> Managers { 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<Employee>().ToTable("Employees");

modelBuilder.Entity<Manager>().ToTable("Managers");

}

# ГЛАВА 6

# Запросы LINQ to Entities

Введение в LINQ to Entities

Для извлечения данных из базы данных Entity Framework Core использует технологию **LINQ to Entities**. В основе данной технологии лежит язык интегрированных запросов LINQ (Language Integrated Query). LINQ предлагает простой и интуитивно понятный подход для получения данных с помощью выражений, которые по форме близки выражениям языка SQL.

Хотя при работе с базой данных мы оперируем запросами LINQ, но, реляционные базы данных как MS SQL Server или SQLite, понимают только запросы на языке SQL. Поэтому Entity Framework Core, используя выражения LINQ to Entities, транслирует их в определенные запросы, понятные для используемого источника данных.

### Построение запросов

Создавать запросы мы можем двумя способами: через операторы LINQ и через методы расширения. Пусть для рассмотрения основных запросов в LINQ to Entities у нас будут следующие модели со связью один-ко-многим:

public class Company

{

public int Id { get; set; }

public string? Name { get; set; }

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

}

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public int Age { get; set; }

public int CompanyId { get; set; }

public Company? Company { get; set; }

}

И пусть будет следующий контекст данных:

public class ApplicationContext : DbContext

{

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

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

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

}

}

Вначале используем некоторые операторы LINQ:

using (ApplicationContext db = new ApplicationContext())

{

// пересоздаем 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", Age = 36, Company = microsoft };

User bob = new User { Name = "Bob", Age = 39, Company = google };

User alice = new User { Name = "Alice", Age = 28, Company = microsoft };

User kate = new User { Name = "Kate", Age = 25, Company = google };

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

db.SaveChanges();

}

using (ApplicationContext db = new ApplicationContext())

{

var users = (from user in db.Users.Include(p => p.Company)

where user.CompanyId == 1

select user).ToList();

foreach (var user in users)

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

}

Вначале здесь происходит добавление данных, если они отсутствуют. Далее примяются Linq-операторы:

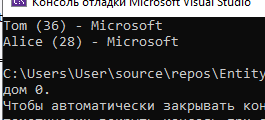
var users = (from user in db.Users.Include(p => p.Company)

where user.CompanyId == 1

select user).ToList();

Данный запрос говорит, что из набора db.Users надо выбрать каждый объект в переменную user. Далее с помощью оператора where проводится фильтрация объектов, и если объект соответствует критерию (в данном случае id компании должно равняться 1), то этот объект передается дальше. И в конце оператор select передает выбранные значения в результирующую выборку, которая возвращается LINQ-выражением.

В итоге мы получим следующий консольный вывод:



Теперь для получения данных используем методы расширения LINQ:

using (ApplicationContext db = new ApplicationContext())

{

var users = db.Users.Include(p => p.Company).Where(p => p.CompanyId == 1);

}

Оба запроса в итоге транслируются в одно и то же выражение sql:

|  |  |
| --- | --- |
| 1  2  3  4 | SELECT "u"."Id", "u"."Age", "u"."CompanyId", "u"."Name", "c"."Id", "c"."Name"  FROM "Users" AS "u"  INNER JOIN "Companies" AS "c" ON "u"."CompanyId" = "c"."Id"  WHERE "u"."CompanyId" = 1 |

Для большинства методов определены асинхронные версии, при необходимости получать данные в асинхронном режиме, мы можем их задействовать:

using (ApplicationContext db = new ApplicationContext())

{

var users = await db.Users

.Include(p => p.Company)

.Where(p => p.CompanyId == 1)

.ToListAsync(); // асинхронное получение данных

foreach (var user in users)

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

}

## **Выборка и фильтрация**

Рассмотрим, как в EF Core выполнять фильтрацию. Для этого используем модели из прошлой темы:

public class Company

{

public int Id { get; set; }

public string? Name { get; set; }

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

}

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public int Age { get; set; }

public int CompanyId { get; set; }

public Company? Company { get; set; }

}

### Where

Если необходимо отфильтровать получаемые данные, то для этого можно использовать метод **Where**. Например, выберем из бд всех пользователей, которые работают в компании "Google":

using (ApplicationContext db = new ApplicationContext())

{

var users = db.Users.Where(p => p.Company!.Name == "Google");

foreach (User user in users)

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

}

Аналогичный запос с помощью операторов LINQ:

using (ApplicationContext db = new ApplicationContext())

{

var users = (from user in db.Users

where user.Company!.Name == "Google"

select user).ToList();

foreach (User user in users)

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

}

### EF.Functions.Like

С помощью метода **EF.Functions.Like()** можно задать условие запроса, которое транслируется в Entity Framework Core в выражение с оператором LIKE. Метод принимает два параметра - оцениваемое выражение и шаблон, с которым сравнивается его значение. Например, найдем всех пользователей, в имени которых присутствует подстрока "Tom" (это могут быть "Tom", "Tomas", "Tomek", "Smith Tom"):

using (ApplicationContext db = new ApplicationContext())

{

var users = db.Users.Where(p => EF.Functions.Like(p.Name!, "%Tom%"));

foreach (User user in users)

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

}

Выражение EF.Functions.Like(p.Name!, "%Tom%")) означает, что мы ищем строки, где в свойстве Name содержиться подстрока "Tom". Поскольку Name представляет nullable-тип, то после названия свойства указывается оператор !.

На стороне БД этот запрос будет транслироваться в следующую SQL-команду:

|  |  |
| --- | --- |
| 1  2  3 | SELECT "u"."Id", "u"."Age", "u"."CompanyId", "u"."Name"        FROM "Users" AS "u"        WHERE "u"."Name" LIKE '%Tom%' |

Для определения шаблона могут применяться ряд специальных символов подстановки:

* **%**: соответствует любой подстроке, которая может иметь любое количество символов, при этом подстрока может и не содержать ни одного символа
* **\_**: соответствует любому одиночному символу
* **[ ]**: соответствует одному символу, который указан в квадратных скобках
* **[ - ]**: соответствует одному символу из определенного диапазона
* **[ ^ ]**: соответствует одному символу, который **не** указан после символа ^

Стоит отметить, что в качестве первого параметра метод принимает оцениваемое выражение в виде строки. В случае со свойством Name все просто, так как оно представляет тип string. Но если нам необходимо использовать в качестве оцеениваемого выражения другие свойства, то их следует привести к строке. Например, найдем всех пользователей у которых возраст (свойство Age) в диапазоне от 20 до 29:

Например, следующее выражение:

Стоит отметить, что в качестве первого параметра метод принимает оцениваемое выражение в виде строки. В случае со свойством Name все просто, так как оно представляет тип string. Но если нам необходимо использовать в качестве оцеениваемого выражения другие свойства, то их следует привести к строке. Например, найдем всех пользователей у которых возраст (свойство Age) в диапазоне от 20 до 29:

Например, следующее выражение:

|  |  |
| --- | --- |
| 1 | var users = db.Users.Where(u => EF.Functions.Like(u.Age.ToString(), "2%")); |

Подобным образом метод **EF.Functions.Like()** можно использовать с операторами LINQ:

|  |  |
| --- | --- |
| 1  2  3 | var users = from u in db.Users               where EF.Functions.Like(u.Age.ToString(), "2%")               select u; |

### Find/FindAsync

Для выборки одного объекта мы можем использовать метод **Find()/FindAsync()**. Данный метод не является методом Linq, он определен у класса DbSet:

|  |  |
| --- | --- |
| 1  2  3  4 | User? user = db.Users.Find(3); // выберем элемент с id=3  // асинхронная версия  // User? user = await db.Users.FindAsync(3); // выберем элемент с id=3  if(user!=null) Console.WriteLine($"{user.Name} ({user.Age})"); |

При выполнении запроса он будет трансформироваться в следующее выражение SQL:

|  |  |
| --- | --- |
| 1  2  3  4 | SELECT "u"."Id", "u"."Age", "u"."CompanyId", "u"."Name"  FROM "Users" AS "u"  WHERE "u"."Id" = @\_\_p\_0  LIMIT 1 |

### First/FirstOrDefault/FirstAsync/FirstOrDefaultAsync

Но в качестве альтернативы мы можем использовать методы Linq **First()/FirstOrDefault()** и их асинхронные версии **FirstAsync()/FirstOrDefaultAsync()**. Они получают первый элемент выборки, который соответствует определенному условию или набору условий. Использование метода FirstOrDefault() является более гибким, так как если выборка пуста, то он вернет значение null. А метод First() в той же ситуации выбросит ошибку.

С помощью данных методов можно просто получить первый объект из выборки:

|  |  |
| --- | --- |
| 1  2  3  4 | User? user = db.Users.FirstOrDefault();  // асинхронная версия  // User? user = await db.Users.FirstOrDefaultAsync();  if(user!=null) Console.WriteLine(user.Name); |

Но в качестве параметра также можно передать условие. Тогда в выборку попадают только те объекты, которые соответствуют условию:

|  |  |
| --- | --- |
| 1  2  3  4 | User? user = db.Users.FirstOrDefault(p=>p.Id==3);  // асинхронная версия  // User? user = await db.Users.FirstOrDefaultAsync(p=>p.Id==3);  if(user!=null) Console.WriteLine(user.Name); |

По тому же принципу работают пары методов **Single/SingleOrDefault** и **Last/LastOrDefault**, которые извлекают соответственно любой единственный элемент и последний элемент последовательности.

## **Cортировка и проекция из базы данных**

### Проекция

Проекция позволяет получить из набора объектов одного типа набор объектов другого типа.

Пусть у нас есть те же классы, что и в прошлой теме:

public class Company

{

public int Id { get; set; }

public string? Name { get; set; }

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

}

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public int Age { get; set; }

public int CompanyId { get; set; }

public Company? Company { get; set; }

}

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

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

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

}

}

Допустим, нам надо добавить в результат выборки название компании. Мы можем использовать метод **Include** для подсоединения к объекту связанных данных из другой таблицы: var users = db.Users.Include(p=>p.Company). Но не всегда нужны все свойства выбираемых объектов. В этом случае мы можем применить метод **Select** для проекции извлеченных данных на новый тип:

using (ApplicationContext db = new ApplicationContext())

{

var users = db.Users.Select(p => new

{

Name = p.Name,

Age = p.Age,

Company = p.Company!.Name

});

foreach (var user in users)

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

}

В итоге метод Select из полученных данных спроецирует новый тип. При выполнении этот запрос будет трансформироваться в следующее выражение SQL:

|  |  |
| --- | --- |
| 1  2  3 | SELECT "u"."Name", "u"."Age", "c"."Name" AS "Company"  FROM "Users" AS "u"  INNER JOIN "Companies" AS "c" ON "u"."CompanyId" = "c"."Id" |

В даном случае мы получим данные анонимного типа, но это также может быть определенный пользователем тип. Например:

public class UserModel

{

public string? Name { get; set; }

public string? Company { get; set; }

public int Age { get; set; }

}

И спроецируем выборку на этот тип:

var users = db.Users.Select(p => new

{

Name = p.Name,

Age = p.Age,

Company = p.Company!.Name

});

foreach (var user in users)

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

### **Сортировка**

Чтобы отсортировать полученные из бд данных по определенному критерию по возрастанию применяется метод **OrderBy**, либо оператор **orderby**. Например, отсортируем объекты по возрастанию по свойству Name:

using (ApplicationContext db = new ApplicationContext())

{

var users = db.Users.OrderBy(p => p.Name);

foreach (var user in users)

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

}

В результате Entity Framework будет генерировать следующее выражение SQL, которое будет упорядочивать данные:

|  |  |
| --- | --- |
| 1  2  3 | SELECT "u"."Id", "u"."Age", "u"."CompanyId", "u"."Name"  FROM "Users" AS "u"  ORDER BY "u"."Name" |

В качестве альтернативы методу OrderBy можно использовать оператор **orderby**:

using (ApplicationContext db = new ApplicationContext())

{

var users = from u in db.Users

orderby u.Name

select u;

foreach (User user in users)

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

}

Для сортировки по убыванию применяется метод **OrderByDescending()**:

|  |  |
| --- | --- |
| 1 | var users = db.Users.OrderByDescending(u => u.Name); |

В этом случае к выражению SQL будет добавляться оператор DESC:

|  |  |
| --- | --- |
| 1  2  3 | SELECT "u"."Id", "u"."Age", "u"."CompanyId", "u"."Name"  FROM "Users" AS "u"  ORDER BY "u"."Name" DESC |

При необходимости упорядочить данные сразу по нескольким критериям можно использовать методы **ThenBy()**(для сортировки по возрастанию) и **ThenByDescending()**. Например, отсортируем по двум значениям:

var users = db.Users.OrderBy(u => u.Age).ThenBy(u => u.Company!.Name);

## **Соединение и группировка таблиц**

### Соединение таблиц

Для объединения таблиц по определенному критерию в Entity Framework Core используется метод **Join**. Для примера возьмем модели из прошлой темы:

public class Company

{

public int Id { get; set; }

public string? Name { get; set; }

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

}

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public int Age { get; set; }

public int CompanyId { get; set; }

public Company? Company { get; set; }

## }

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

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

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

## }

Например, в нашем случае таблица пользователей и таблица компаний имеет общий критерий - id компании, по которому можно провести объединение таблиц:

using (ApplicationContext db = new ApplicationContext())

{

var users = db.Users.Join(db.Companies, // второй набор

u => u.CompanyId, // свойство-селектор объекта из первого набора

c => c.Id, // свойство-селектор объекта из второго набора

(u, c) => new // результат

{

Name = u.Name,

Company = c.Name,

Age = u.Age

});

foreach (var u in users)

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

## }

Метод Join принимает четыре параметра:

* вторую таблицу, которая соединяется с текущей
* свойство объекта - столбец из первой таблицы, по которому идет соединение
* свойство объекта - столбец из второй таблицы, по которому идет соединение
* новый объект, который получается в результате соединения

В итоге данный запрос будет транслироваться в следующее выражение SQL:

|  |  |
| --- | --- |
| 1  2  3 | SELECT "u"."Name", "c"."Name" AS "Company", "u"."Age"  FROM "Users" AS "u"  INNER JOIN "Companies" AS "c" ON "u"."CompanyId" = "c"."Id" |

Аналогичного результата мы могли бы достигнуть, если бы использовали оператор join:

var users = from u in db.Users

join c in db.Companies on u.CompanyId equals c.Id

## select new { Name = u.Name, Company = c.Name, Age = u.Age };

### **Соединение трех таблиц**

Допустим, у нас есть три таблицы, которые связаны между собой и которые описываются следующими моделями:

public class Country

{

public int Id { get; set; }

public string? Name { get; set; }

}

public class Company

{

public int Id { get; set; }

public string? Name { get; set; }

public int CountryId { get; set; }

public Country? Country { get; set; }

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

}

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public int Age { get; set; }

public int CompanyId { get; set; }

public Company? Company { get; set; }

}

public DbSet<Country> Countries { get; set; } = null!;

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

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

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

}

}

Объединим три таблицы в один набор:

using (ApplicationContext db = new ApplicationContext())

{

// пересоздаем базу данных

db.Database.EnsureDeleted();

db.Database.EnsureCreated();

Country usa = new Country { Name = "USA" };

Company microsoft = new Company { Name = "Microsoft", Country = usa };

Company google = new Company { Name = "Google", Country = usa };

db.Companies.AddRange(microsoft, google);

User tom = new User { Name = "Tom", Age = 36, Company = microsoft };

User bob = new User { Name = "Bob", Age = 39, Company = google };

User alice = new User { Name = "Alice", Age = 28, Company = microsoft };

User kate = new User { Name = "Kate", Age = 25, Company = google };

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

db.SaveChanges();

}

using (ApplicationContext db = new ApplicationContext())

{

var users = from user in db.Users

join company in db.Companies on user.CompanyId equals company.Id

join country in db.Countries on company.CountryId equals country.Id

select new

{

Name = user.Name,

Company = company.Name,

Age = user.Age,

Country = country.Name

};

foreach (var u in users)

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

}

### Группировка

Для группировки данных по определенным критериям применяется оператор **group by**, либо метод **GroupBy()**. Например, сгруппируем пользователей по компаниям:

using (ApplicationContext db = new ApplicationContext())

{

var groups = from u in db.Users

group u by u.Company!.Name into g

select new

{

g.Key,

Count = g.Count()

};

foreach (var group in groups)

{

Console.WriteLine($"{group.Key} - {group.Count}");

}

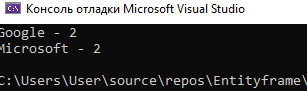
}

В данном случае критерием для объединения групп служит название компании, то есть столбец Name из связанной таблицы Companies. Критерий, по которому проводится группировка, является ключом. Ключ мы можем получить через свойство **Key**, которое есть у группы. Кроме того, у группы есть метод Count(), с помощью которого можно получить количество объектов в группе.

При выполнении запроса будет создаваться следующее SQL-выражение:

|  |  |
| --- | --- |
| 1  2  3  4 | SELECT "c"."Name" AS "Key", COUNT(\*) AS "Count"  FROM "Users" AS "u"  INNER JOIN "Companies" AS "c" ON "u"."CompanyId" = "c"."Id"  GROUP BY "c"."Name" |

В итоге мы получим несколько групп, каждая из которых будет содержать несколько элементов. Например, в моем случае вывод будет следующим:



Аналогично работает метод **GroupBy()**:

var groups = db.Users.GroupBy(u => u.Company!.Name).Select(g => new

{

g.Key,

Count = g.Count()

});

## **Операции с множествами: объединение, пересечение, разность**

Ряд методов Linq позволяют работать с результатами выборки как со множествами, производя операции на объединение, пересечение, разность двух выборок.

Но перед использованием данных методов надо учитывать, что они проводятся над однородными выборками с одинаковым определением строк, то есть которые совпадают по составу столбцов.

Для примеров возьмем модели из прошлых тем:

public class Company

{

public int Id { get; set; }

public string? Name { get; set; }

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

}

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public int Age { get; set; }

public int CompanyId { get; set; }

public Company? Company { get; set; }

## }

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

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

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

## }

## На примере следующих данных:

using(ApplicationContext db = new ApplicationContext())

{

// пересоздаем 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", Age = 36, Company = microsoft };

User bob = new User { Name = "Bob", Age = 39, Company = google };

User alice = new User { Name = "Alice", Age = 28, Company = microsoft };

User kate = new User { Name = "Kate", Age = 25, Company = google };

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

db.SaveChanges();

## }

### Объединение

Для объединения двух выборок используется метод **Union()**:

using (ApplicationContext db = new ApplicationContext())

{

var users = db.Users.Where(u => u.Age < 30)

.Union(db.Users.Where(u => u.Name!.Contains("Tom")));

foreach (var user in users)

Console.WriteLine(user.Name);

## }

## Консольный вывод:

## 

Метод Union в качестве параметра принимает результаты второй выборки и объединяет ее с исходной. В результате мы получим два подзапроса SQL, результаты которых объединяются в общий набор:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | SELECT "u"."Id", "u"."Age", "u"."CompanyId", "u"."Name"  FROM "Users" AS "u"  WHERE "u"."Age" < 30  UNION  SELECT "u0"."Id", "u0"."Age", "u0"."CompanyId", "u0"."Name"  FROM "Users" AS "u0"  WHERE ('Tom' = '') OR (instr("u0"."Name", 'Tom') > 0) |

При этом мы не можем объединить две разнородные выборки, например, таблицу, пользователей и таблицу компаний. Однако уместна следующая запись:

var result = db.Users.Select(p => new { Name = p.Name })

## .Union(db.Companies.Select(c => new { Name = c.Name }));

Первая выборка после метода Select будет формировать набор элементов с одним столбцом Name. Вторая выборка из таблицы компаний после метода Select также будет формировать набор элементов с одним столбцом Name. Поэтому строки в обоих выборках будут однородны, и мы их сможем объединять.

### **Пересечение**

Чтобы найти пересечение выборок, то есть те элементы, которые присутствуют сразу в двух выборках, используется метод **Intersect()**. Например, выберем все пользователей, у которых возраст больше 30 и в имени содержится подстрока "Tom":

using (ApplicationContext db = new ApplicationContext())

{

var users = db.Users.Where(u => u.Age > 30)

.Intersect(db.Users.Where(u => u.Name!.Contains("Tom")));

foreach (var user in users)

Console.WriteLine(user.Name);

## }

## 

Несмотря на то, что операция пересечения не равна операции объединения, рассмотренной выше, но при выполнении пересечение будет выполнять те же SQL-выражения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | SELECT "u"."Id", "u"."Age", "u"."CompanyId", "u"."Name"  FROM "Users" AS "u"  WHERE "u"."Age" > 30  INTERSECT  SELECT "u0"."Id", "u0"."Age", "u0"."CompanyId", "u0"."Name"  FROM "Users" AS "u0"  WHERE ('Tom' = '') OR (instr("u0"."Name", 'Tom') > 0) |

### Разность

Если нам надо найти элементы первой выборки, которые отсутствуют во второй выборке, то мы можем использовать метод **Except**:

using (ApplicationContext db = new ApplicationContext())

{

var selector1 = db.Users.Where(u => u.Age > 30); //

var selector2 = db.Users.Where(u => u.Name!.Contains("Tom"));

var users = selector1.Except(selector2);

foreach (var user in users)

Console.WriteLine(user.Name);

## }

## 

В результате будет формироваться следующий SQL-запрос:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | SELECT "u"."Id", "u"."Age", "u"."CompanyId", "u"."Name"  FROM "Users" AS "u"  WHERE "u"."Age" > 30  EXCEPT  SELECT "u0"."Id", "u0"."Age", "u0"."CompanyId", "u0"."Name"  FROM "Users" AS "u0"  WHERE ('Tom' = '') OR (instr("u0"."Name", 'Tom') > 0) |

## **Агрегатные операции**

## Entity Framework Core поддерживает ряд агрегатных операций, например, нахождение суммы по столбцу, количества элементов в выборке и т.д. Эти операции представлены такими методами как **All, Any, Count, Sum, Min, Max, Average**. Для рассмотрения агрегатных операций возьмем модели из прошлой темы:

public class Company

{

public int Id { get; set; }

public string? Name { get; set; }

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

}

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public int Age { get; set; }

public int CompanyId { get; set; }

public Company? Company { get; set; }

}

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

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

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

}

### Наличие элементов

Метод **Any()** позволяет проверить, есть ли в базе данных элемент с определенными признаками, и если есть, то метод возвратит значение true. Например, проверим, есть ли в базе данных пользователи, которые работают в компании Google:

bool result = db.Users.Any(u => u.Company!.Name == "Google");

|  |
| --- |
| SELECT EXISTS (      SELECT 1      FROM "Users" AS "u"      INNER JOIN "Companies" AS "c" ON "u"."CompanyId" = "c"."Id"      WHERE "c"."Name" = 'Google') |

Метод **All()** позволяет проверит, удовлетворяют ли все элементы в базе данных определенному критерию. Например, проверим, все ли пользователи работают в компании Microsoft:

bool result = db.Users.All(u => u.Company!.Name == "Microsoft");

При вызове будет выполняться следующий SQL-запрос:

|  |  |
| --- | --- |
| 1  2  3  4  5 | SELECT NOT EXISTS (      SELECT 1      FROM "Users" AS "u"      INNER JOIN "Companies" AS "c" ON "u"."CompanyId" = "c"."Id"      WHERE ("c"."Name" <> 'Microsoft') OR "c"."Name" IS NULL) |

### Количество элементов в выборке

Метод Count() позволяет найти количество элементов в выборке:

using (ApplicationContext db = new ApplicationContext())

{

int number1 = db.Users.Count();

// найдем кол-во пользователей, которые в имени содержат подстроку Tom

int number2 = db.Users.Count(u => u.Name!.Contains("Tom"));

Console.WriteLine(number1);

Console.WriteLine(number2);

}

В результате будут выполняться варажения SQL наподобие:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | SELECT COUNT(\*)  FROM "Users" AS "u"    SELECT COUNT(\*)  FROM Users AS u  WHERE ('Tom' == '') || (instr(u.Name, 'Tom') > 0)) |

### Минимальное, максимальное и среднее значения

Для нахождения минимального, максимального и среднего значений по выборке применяются функции **Min()**, **Max()** и **Average()** соответственно. Найдем минимальный, максимальный и средний возраст пользователей:

using (ApplicationContext db = new ApplicationContext())

{

// минимальный возраст

int minAge = db.Users.Min(u => u.Age);

// максимальный возраст

int maxAge = db.Users.Max(u => u.Age);

// средний возраст пользователей, которые работают в Microsoft

double avgAge = db.Users.Where(u => u.Company!.Name == "Microsoft")

.Average(p => p.Age);

Console.WriteLine(minAge);

Console.WriteLine(maxAge);

Эти запросы трансформируются в SQLite в следующие SQL-выражения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | SELECT MIN("u"."Age")  FROM "Users" AS "u"    SELECT MAX("u"."Age")  FROM "Users" AS "u"    SELECT AVG(CAST("u"."Age" AS REAL))  FROM "Users" AS "u"  INNER JOIN "Companies" AS "c" ON "u"."CompanyId" = "c"."Id"  WHERE "c"."Name" = 'Microsoft' |

Для получения суммы значений используется метод **Sum()**:

using (ApplicationContext db = new ApplicationContext())

{

// суммарный возраст всех пользователей

int sum1 = db.Users.Sum(u => u.Age);

// суммарный возраст тех, кто работает в Microsoft

int sum2 = db.Users.Where(u => u.Company!.Name == "Microsoft")

.Sum(u => u.Age);

Console.WriteLine(sum1);

Console.WriteLine(sum2);

}

Во втором случае будет выполняться следующее SQL-выражение:

|  |  |
| --- | --- |
| 1  2  3  4 | SELECT COALESCE(SUM("u"."Age"), 0)  FROM "Users" AS "u"  INNER JOIN "Companies" AS "c" ON "u"."CompanyId" = "c"."Id"  WHERE "c"."Name" = 'Microsoft' |

## **Отслеживание объектов и AsNoTracking**

Запросы могут быть отслеживаемыми и неотслеживаемыми. По умолчанию все запросы, которые возвращают объекты классов моделей являются отслеживаемыми. Когда контекст данных извлекает данные из базы данных, Entity Framework Core помещает извлеченные объекты в кэш и отслеживает изменения, которые происходят с этими объектами вплоть до использования метода **SaveChanges()/SaveChangesAsync()**, который фиксирует все изменения в базе данных. Но нам не всегда необходимо отслеживать изменения. Например, нам надо просто вывести данные для просмотра.

Чтобы данные не помещались в кэш, применяется метод **AsNoTracking()**. Этот метод применяется к объекту **IQueryable**. При его применении возвращаемые из запроса данные не кэшируются. То есть запрос является неотслеживаемым. А это означает, что Entity Framework Core не производит какую-то дополнительную обработку и не выделяет дополнительное место для хранения извлеченных из БД объектов. И поэтому такие запросы работают быстрее.

Небольшой пример. Допустим, у нас есть следующие модели и контекст данных:

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!;

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

}

}

Пусть в базе данных есть есть несколько объектов User:

using (ApplicationContext db = new ApplicationContext())

{

// пересоздаем базу данных

db.Database.EnsureDeleted();

db.Database.EnsureCreated();

User tom = new User { Name = "Tom", Age = 36 };

User bob = new User { Name = "Bob", Age = 39 };

User alice = new User { Name = "Alice", Age = 28 };

User kate = new User { Name = "Kate", Age = 25 };

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

db.SaveChanges();

}

При обычном выполнении:

using (ApplicationContext db = new ApplicationContext())

{

var user = db.Users.FirstOrDefault();

if (user != null)

{

user.Age = 18;

db.SaveChanges();

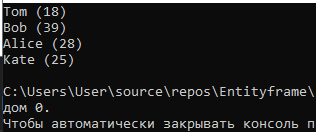
}

var users = db.Users.ToList();

foreach (var u in users)

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

}



Мы видим, что в наборе users первый элемент имеет у свойства Age значение 18.

Причем в данном случае мы можем и не вызывать метод **SaveChanges**, элемент уже и так будет закэширован. Метод **SaveChanges** необходим, чтобы применить все изменения с объектами в базе данных.

Но если бы мы использовали метод **AsNoTracking()**, то результат был бы другой:

using (ApplicationContext db = new ApplicationContext())

{

var user = db.Users.AsNoTracking().FirstOrDefault();

if (user != null)

{

user.Age = 22;

db.SaveChanges();

}

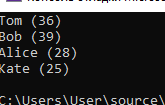
var users = db.Users.AsNoTracking().ToList();

foreach (var u in users)

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

}

Консольный вывод будет тот же, что и в предыдущем случае:



Так как при получении первого элемента используется AsNoTracking, он не будет отслеживаться, и поэтому вызов db.SaveChanges() никак не повлияет на базу данных, а первый элемент сохранит свое первоначальное значение у свойства Age.

### Свойство ChangeTracker

Кроме использования метода AsNoTracking, можно отключить отслеживание в целом для объекта контекста. Для этого надо установить значение **QueryTrackingBehavior.NoTracking** для свойства **db.ChangeTracker.QueryTrackingBehavior**:

using (ApplicationContext db = new ApplicationContext())

{

db.ChangeTracker.QueryTrackingBehavior = QueryTrackingBehavior.NoTracking;

var user = db.Users.FirstOrDefault();

if (user != null)

{

user.Age = 8;

db.SaveChanges();

}

var users = db.Users.ToList();

foreach (var u in users)

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

}

Также можно отключить отслеживание на уровне всего контекста данных, например, в его конструкторе

public ApplicationContext()

{

ChangeTracker.QueryTrackingBehavior = QueryTrackingBehavior.NoTracking;

}

Вобще через свойство **ChangeTracker** мы можем управлять отслеживанием объектом и получать разнообразную информацию. Например, мы можем узнать, сколько объектов отслеживается в текущий момент:

using (ApplicationContext db = new ApplicationContext())

{

var users = db.Users.ToList();

foreach (var u in users)

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

int count = db.ChangeTracker.Entries().Count();

Console.WriteLine($"{count}");

}

## **Выполнение запросов**

Если запрос является не отслеживаемым, то есть отслеживание отключено с помощью метода **AsNoTracking()** или установки свойства ChangeTracker.QueryTrackingBehavior = QueryTrackingBehavior.NoTracking, то EF создает по этим данным новые объекты и возвращает их пользователю. В отличие от отслеживаемых запросов созданные объекты не добавляются в контекст и не отслеживаются.

Рассмотрим несколько примеров. Для этого возьмем следующую модель User и контекст данных:

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

## }

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)

{

// добавляем один объект

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

}

## }

## Отслеживаемые запросы:

using (ApplicationContext db = new ApplicationContext())

{

var user1 = db.Users.FirstOrDefault();

var user2 = db.Users.FirstOrDefault();

if (user1 != null && user2 != null)

{

Console.WriteLine($"Before User1: {user1.Name} User2: {user2.Name}");

user1.Name = "Bob";

Console.WriteLine($"After User1: {user1.Name} User2: {user2.Name}");

}

## }

## 

Так как запрос db.Users.FirstOrDefault() является отслеживаемым, то при получении данных, по ним будет создаваться объект user1, который добавляется в контекст и начинает отслеживаться.

Далее повторно вызывается данный запрос для получения объекта user2. Этот запрос то же является отслеживаемым, поэтому EF увидит, что такой объект уже есть в контексте, он уже отслеживается, и возвратит ссылку на тот же объект. Поэтому все изменения с переменной user1 затронут и переменную user2.

Рассмотрим другой пример:

using (ApplicationContext db = new ApplicationContext())

{

var user1 = db.Users.FirstOrDefault();

var user2 = db.Users.AsNoTracking().FirstOrDefault();

if (user1 != null && user2 != null)

{

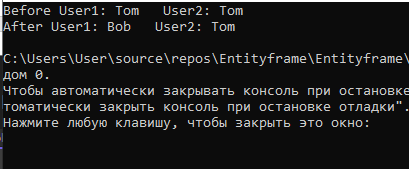
Console.WriteLine($"Before User1: {user1.Name} User2: {user2.Name}");

user1.Name = "Bob";

Console.WriteLine($"After User1: {user1.Name} User2: {user2.Name}");

}

}



С первым объектом user1 все по прежнему: он также попадает в контекст и отслеживается. А вот второй запрос теперь является неотслеживаемым, так как использует метод AsNoTracking.

**IEnumerable и IQueryable**

При вызове методов LINQ мы только создаем запрос. Его непосредственное выполнение происходит, когда мы начинаем потреблять результаты этого запроса. Нередко это происходит при переборе результата запроса в цикле **for** или при применении к нему ряда методов - ToList или ToArray, а также если запрос представляет скалярное значение, например, метод Count.

При вызове методов LINQ мы только создаем запрос. Его непосредственное выполнение происходит, когда мы начинаем потреблять результаты этого запроса. Нередко это происходит при переборе результата запроса в цикле **for** или при применении к нему ряда методов - ToList или ToArray, а также если запрос представляет скалярное значение, например, метод Count.

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

}

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)

{

// добавляем один объект

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

}

}

Возьмем два вроде бы идентичных выражения. Объект IEnumerable:

using (ApplicationContext db = new ApplicationContext())

{

IEnumerable<User> userIEnum = db.Users;

var users = userIEnum.Where(p => p.Id < 10).ToList();

Для примера здесь получаем всех пользователей, у которых id меньше 10.

Здесь запрос будет иметь следующий вид:

|  |  |
| --- | --- |
| 1  2 | SELECT "u"."Id", "u"."Name"  FROM "Users" AS "u" |

Фильтрация результата, обозначенная с помощью метода Where(p => p.Id > id) будет идти уже после выборки из бд в самом приложении.

Чтобы совместить фильтры, нам надо было сразу применить метод Where: db.Users.Where(p => p.Id < 10);

Объект IQueryable:

using (ApplicationContext db = new ApplicationContext())

{

IQueryable<User> userIQuer = db.Users;

var users = userIQuer.Where(p => p.Id < 10).ToList();

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

}

Здесь запрос будет иметь следующий вид:

|  |  |
| --- | --- |
| 1  2  3 | SELECT "u"."Id", "u"."Name"  FROM "Users" AS "u"  WHERE "u"."Id" < 10 |

Таким образом, все методы суммируются, запрос оптимизируется, и только потом, когда идет обращеие к методу **ToList()**, происходит выборка из базы данных.

Это позволяет динамически создавать сложные запросы. Например, мы можем последовательно наслаивать в зависимости от условий выражения для фильтрации:

IQueryable<User> userIQuer = db.Users;

userIQuer = userIQuer.Where(p => p.Id < 10);

userIQuer = userIQuer.Where(p => p.Name == "Tom");

var users = userIQuer.ToList();

В данном случае будет создаваться следующий SQL-запрос:

|  |  |
| --- | --- |
| 1  2  3 | SELECT "u"."Id", "u"."Name"  FROM "Users" AS "u"  WHERE ("u"."Id" < 10) AND ("u"."Name" = 'Tom') |

Что же лучше использовать? Все зависит от конкретной ситуации. Если разработчику нужен весь набор возвращаемых данных, то лучше использовать IEnumerable, предоставляющий максимальную скорость. Если же нам не нужен весь набор, а то только некоторые отфильтрованные данные, то лучше применять IQueryable.

## Фильтры запросов уровня модели

Фильтры запросов уровня модели (Model-level query filters) позволяют определить предикат запроса LINQ непосредственно в метаданных модели (обычно в методе OnModelCreating контекста данных). Такие фильтры автоматически применяются к любым запросам LINQ, в которых используются классы, для которых определен фильтр.

Пусть у нас определены следующие классы:

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public int Age { get; set; }

public int RoleId { get; set; }

public Role? Role { get; set; }

}

public class Role

{

public int Id { get; set; }

public string? Name { get; set; }

}

Здесь класс пользователя имеет ссылку на класс роли, к которой он принадлежит. И также определим контекст данных:

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

public DbSet<Role> Roles { get; set; } = null!;

public int RoleId { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<User>().HasQueryFilter(u => u.Age > 17 && u.RoleId == RoleId);

}

В метод **HasQueryFilter()** передается предикат, которому должен удовлетворять объект User, чтобы быть извлеченным из базы данных. То есть в результате запросов будут извлекаться только те объекты User, у которых значение свойства Age больше 17, а свойство RoleId равно значению свойства RoleId их контекста данных.

Используем данные классы:

using (ApplicationContext db = new ApplicationContext())

{

// пересоздадим базу данных

db.Database.EnsureDeleted();

db.Database.EnsureCreated();

Role adminRole = new Role { Name = "admin" };

Role userRole = new Role { Name = "user" };

User user1 = new User { Name = "Tom", Age = 17, Role = userRole };

User user2 = new User { Name = "Bob", Age = 18, Role = userRole };

User user3 = new User { Name = "Alice", Age = 19, Role = adminRole };

User user4 = new User { Name = "Sam", Age = 20, Role = adminRole };

db.Roles.AddRange(userRole, adminRole);

db.Users.AddRange(user1, user2, user3, user4);

db.SaveChanges();

}

using (ApplicationContext db = new ApplicationContext() { RoleId = 2 })

{

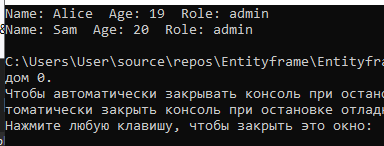
var users = db.Users.Include(u => u.Role).ToList();

foreach (User user in users)

Console.WriteLine($"Name: {user.Name} Age: {user.Age} Role: {user.Role?.Name}");

}

Результатом будет следующий консольный вывод:



То есть только два объекта из добавленных четырех соответствуют тому предикату, который был передан в HasQueryFilter. И данный фильтр будет действовать для всех запросов к базе данных, которые извлекают данные из таблицы Users. Например, нахождение минимального возраста:

using(ApplicationContext db = new ApplicationContext() { RoleId = 2 })

{

int minAge = db.Users.Min(x => x.Age);

Console.WriteLine(minAge); // 19

}

Несмотря на то, что минимальный возраст по всем 4 объектам составляет 17, но так как действует фильтр, при запросе будут учитываться только те объекты в бд, которые соответствуют фильтру.

Если необходимо во время запроса отключить фильтр, то применяется метод **IgnoreQueryFilters()**:

using(ApplicationContext db = new ApplicationContext() { RoleId = 2 })

{

int minAge = db.Users.IgnoreQueryFilters().Min(x => x.Age);

Console.WriteLine(minAge); // 17

}

# 7 ГЛАВА

# SQL в Entity Framework Core

## **Выполнение SQL-запросов**

Кроме использования инфраструктуры LINQ to Entities для создания запросов Entity Framework Core также позволяет писать запросы к базе данных сразу на языке SQL. Это может быть удобно, если запрос очень сложный по своей структуре или если Entity Framework Core на основе Linq to Entities создает не очень оптимальный sql-запрос.

### FromSqlRaw

Для получения данных из БД у объектов DbSet определен метод **FromSqlRaw()**, который принимает в качестве параметра sql-выражение и набор параметров. В качестве результата FromSqlRaw возвращает набор полученных из бд объектов.

При этом надо учитывать, что передаваемое в метод FromSqlRaw SQL-выражение не должно извлекать связанные данные, а полученные значения должны соответствовать определению какого-либо класса.

Для примера возьмем следующие модели и контекст данных:

public class Company

{

public int Id { get; set; }

public string? Name { get; set; }

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

}

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public int Age { get; set; }

public int CompanyId { get; set; }

public Company? Company { get; set; }

}

public class ApplicationContext : DbContext

{

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

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

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

}

}

Пусть ранее у нас были добавлены следующие данные:

using (ApplicationContext db = new ApplicationContext())

{

// пересоздаем 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", Age = 36, Company = microsoft };

User bob = new User { Name = "Bob", Age = 39, Company = google };

User alice = new User { Name = "Alice", Age = 28, Company = microsoft };

User kate = new User { Name = "Kate", Age = 25, Company = google };

User tomas = new User { Name = "Tomas", Age = 22, Company = microsoft };

User tomek = new User { Name = "Tomek", Age = 42, Company = google };

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

db.SaveChanges();

}

Например, получим все объекты из таблицы Companies:

using (ApplicationContext db = new ApplicationContext())

{

var comps = db.Companies.FromSqlRaw("SELECT \* FROM Companies").ToList();

foreach (var company in comps)

Console.WriteLine(company.Name);

}

Выражение SELECT извлекает данные из таблицы. Так как эта таблица сопоставляется с моделью Company и хранит объекты этой модели, то в результате мы получим набор объектов класса Company.

При этом мы можем добавлять к методу другие методы LINQ, которые все вместе будут конкатенироваться в одно общее SQL-выражение:

var comps = db.Companies.FromSqlRaw("SELECT \* FROM Companies").ToList();

foreach (var company in comps)

Console.WriteLine(company.Name);

|  |  |
| --- | --- |
| 1  2  3  4  5 | SELECT "c"."Id", "c"."Name"  FROM (      SELECT \* FROM Companies  ) AS "c"  ORDER BY "c"."Name" |

Также можно использовать метод Include для подгрузки связанных данных:

var users = db.Users.FromSqlRaw("SELECT \* FROM Users")

.Include(c=>c.Company).ToList();

foreach (var user in users)

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

### Параметры

Другая версия метода FromSqlRaw() позволяет использовать параметры. Например, выберем из бд все модели, в названии которых есть подстрока "Tom":

using (ApplicationContext db = new ApplicationContext())

{

SqliteParameter param = new SqliteParameter("@name", "%Tom%");

var users = db.Users.FromSqlRaw("SELECT \* FROM Users WHERE Name LIKE @name", param).ToList();

foreach (var user in users)

Console.WriteLine(user.Name);

}

Поскольку выше приведенный контекст данных подключается к бд SQLite, то для представления параметров применяется тип **SqliteParameter** из пространства имен **Microsoft.Data.Sqlite**. Конструктор SqliteParameter имеет ряд версий. В данном случае в конструктор передается два параметра. Первый параметр - название параметра, через который на него можно ссылаться в строке запроса - в данном случае "@name". Второй параметр - значение параметра.

Если бы подключение шло к бд MS SQL Server, то для представления параметров применялся бы класс **SqlParameter** из пространства имен **Microsoft.Data.SqlClient**:

using (ApplicationContext db = new ApplicationContext())

{

SqlParameter param = new SqlParameter("@name", "%Tom%");

var users = db.Users.FromSqlRaw("SELECT \* FROM Users WHERE Name LIKE @name", param).ToList();

foreach (var user in users)

Console.WriteLine(user.Name);

}

Также мы можем определять параметры как простые переменные:

using (ApplicationContext db = new ApplicationContext())

{

var name = "%Tom%";

var users = db.Users.FromSqlRaw("SELECT \* FROM Users WHERE Name LIKE {0}", name).ToList();

foreach (var user in users)

Console.WriteLine(user.Name);

var age = 30;

users = db.Users.FromSqlRaw("SELECT \* FROM Users WHERE Age > {0}", age).ToList();

foreach (var user in users)

Console.WriteLine(user.Name);

}

### ExecuteSqlRaw

Метод FromSqlRaw() осуществляет выборку из БД, но кроме выборки нам, возможно, придется удалять, обновлять уже существующие или вставлять новые записи. Для этой цели применяется метод **ExecuteSqlRaw()** и его асинхронная версия - **ExecuteSqlRawAsync()**, которые возвращают количество затронутых командой строк. Для его вызова у контекста данных используется свойство Database:

// вставка

string newComp = "Apple";

int numberOfRowInserted = db.Database.ExecuteSqlRaw("INSERT INTO Companies (Name) VALUES ({0})", newComp);

// асинхронная версия

// int numberOfRowInserted2 = await db.Database.ExecuteSqlRawAsync("INSERT INTO Companies (Name) VALUES ({0})", newComp);

// обновление

string appleInc = "Apple Inc.";

string apple = "Apple";

int numberOfRowUpdated = db.Database.ExecuteSqlRaw("UPDATE Companies SET Name={0} WHERE Name={1}", appleInc, apple);

// удаление

int numberOfRowDeleted = db.Database.ExecuteSqlRaw("DELETE FROM Companies WHERE Name={0}", appleInc);

### Интерполяция строк

Для методов FromSqlRaw и ExecuteSqlRaw в EF Core определены их двойники - методы **FromSqlInterpolated()** и **ExecuteSqlInterpolated()** (асинхронная версия - **ExecuteSqlInterpolatedAsync()**), которые позволяют использовать интерполяцию строк для передачи параметров. Пример с **FromSqlInterpolated**:

using (ApplicationContext db = new ApplicationContext())

{

var name = "%Tom%";

var age = 30;

var users = db.Users.FromSqlInterpolated($"SELECT \* FROM Users WHERE Name LIKE {name} AND Age > {age}").ToList();

foreach (var user in users)

Console.WriteLine(user.Name);

}

В этом случае на стороне сервера также будут создаваться параметры для передачи значений name и age наподобие:

@p0 = '%Tom%'

@p1 = 30

SELECT\* FROM Users WHERE Name LIKE @p0 AND Age > @p1

Использование **ExecuteSqlInterpolated()/ExecuteSqlInterpolatedAsync()**:

using (ApplicationContext db = new ApplicationContext())

{

string jetbrains = "JetBrains";

db.Database.ExecuteSqlInterpolated($"INSERT INTO Companies (Name) VALUES ({jetbrains})");

// асинхронная версия

// await db.Database.ExecuteSqlInterpolatedAsync($"INSERT INTO Companies (Name) VALUES ({jetbrains})");

foreach (var comp in db.Companies.ToList())

Console.WriteLine(comp.Name);

}

## **Хранимые функции**

Вместе с обычными sql-запросами Entity Framework Core также позволяет выполнять хранимые функции, которые созданы в базе данных. Рассмотрим вызов хранимой функции в приложении на C# на примере БД MS SQL Server.

Пусть наша база данных описывается следующим контекстом данных и сущностями:

public class Company

{

public int Id { get; set; }

public string? Name { get; set; }

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

}

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public int Age { get; set; }

public int CompanyId { get; set; }

public Company? Company { get; set; }

}

public class ApplicationContext : DbContext

{

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

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

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

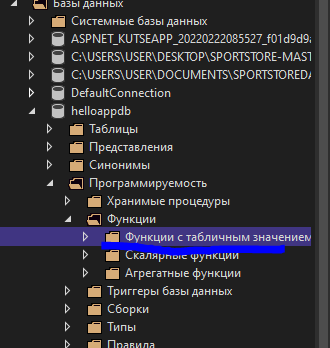
optionsBuilder.UseSqlServer(@"Server=(localdb)\mssqllocaldb;Database=helloappdb;Trusted\_Connection=True;");

}

}

Вначале создадим саму хранимую функцию, к которой мы будем обращаться. Для этого в Visual Studio в окне **SQL Server Object Explorer** откроем узел базы данных. В узле базы данных откроем подузел **Programmability -> Functions -> Table-valued Functions**:

Вначале создадим саму хранимую функцию, к которой мы будем обращаться. Для этого в Visual Studio в окне **SQL Server Object Explorer** откроем узел базы данных. В узле базы данных откроем подузел **Programmability -> Functions -> Table-valued Functions**:



Далее нажмем на этот узел правой кнопкой мыши и выберем **Add New -> Table-valued Function**:

После этого Visual Studio генерирует и открывает файл скрипта со следующим содержимым:

CREATE FUNCTION [dbo].[Function]

(

@param1 int,

@param2 char(5)

)

RETURNS @returntable TABLE

(

c1 int,

c2 char(5)

)

AS

BEGIN

INSERT @returntable

SELECT @param1, @param2

RETURN

END

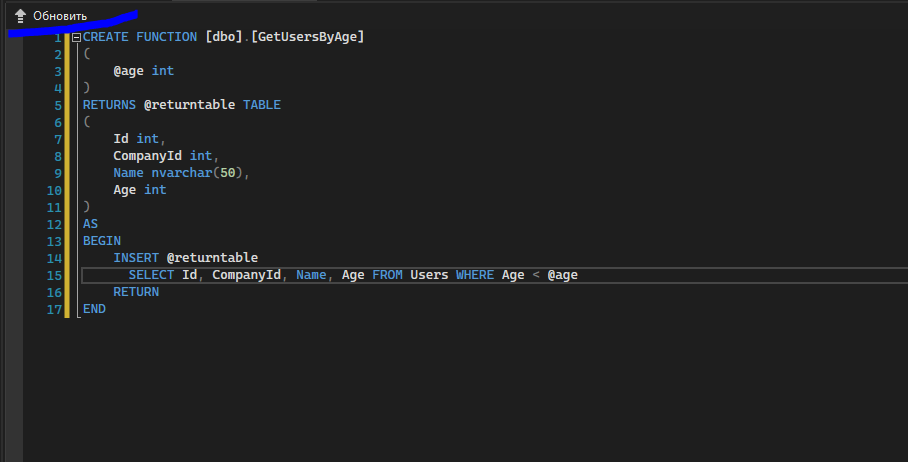
Изменим скрипт следующим образом:

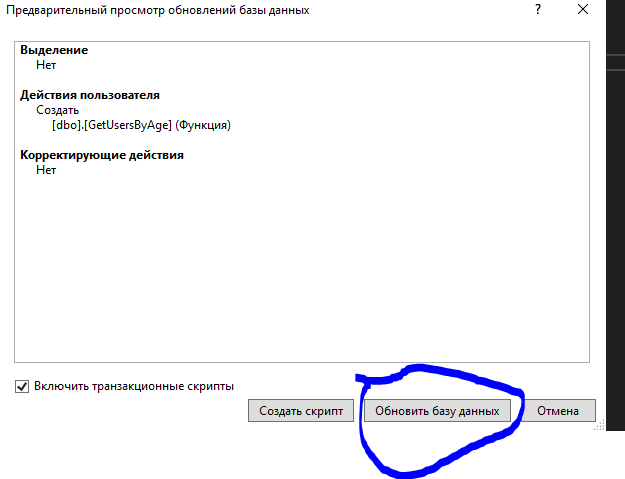
Выражение CREATE FUNCTION [dbo].[GetUsersByAge] создае новую функцию GetUsersByAge. После ее названия определяется список параметров. Пусть наша функция принимает только один параметр @age, который имеет тип int, то есть целочисленное значение.

Затем идет определение возвращаемого объекта-таблицы в выражении RETURNS @returntable TABLE(...). В скобках идет перечисление столбцов возвращаемой таблицы. В данном случае они совпадают с определением таблицы Users. То есть таблица будет содержать объекты класса User.

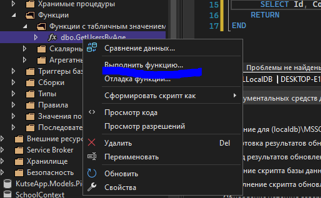
Между выражениями BEGIN и END идет собственно выполнение функции. В данном случае с помощью оператора WHERE функция будет находить все строки, у которых столбец Age содержит меньшее значение, чем в параметре @age.

Теперь добавим функцию в базу данных. Для этого нажмем на кнопку Update:

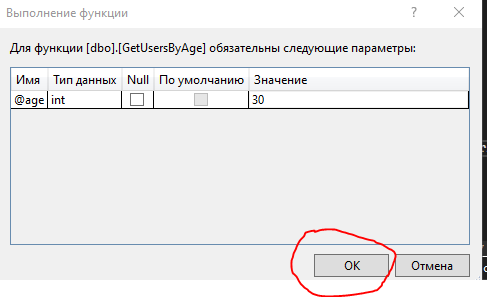
 Затем в появившемся окне нажмем на кнопку **Update Database**

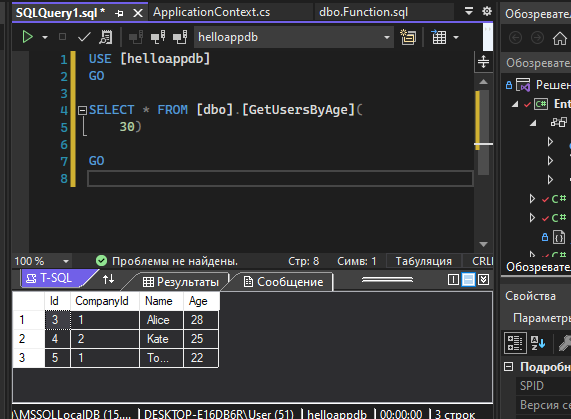


После этого в базу данных будет добавлена функция, и ее можно будет использовать. Но перед обращением к ней из кода c# мы ее протестируем, чтобы убедиться, что она работает как надо. Для этого нажмем на функцию правой кнопкой мыши и в появившемся меню выберем пункт **Execute Function**:



После этого откроется окно для установки входных параметров функции. Введем в поле Value какое-нибудь число, которое будет передаваться в функцию в качестве параметра:





Как видно, я получил ожидаемые результаты, значит, функция работает правильно. Теперь обратимся к функции в коде на C#. Для этого мы можем воспользоватся одним из двух подходов.

### Обращение к функции в запросе SQL

Первый подход предполагает обращение к хранимой функции в запросе SQL, который отправляется из кода C#:

using (ApplicationContext db = new ApplicationContext())

{

SqlParameter param = new SqlParameter("@age", 30);

var users = db.Users.FromSqlRaw("SELECT \* FROM GetUsersByAge (@age)", param).ToList();

foreach (var u in users)

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

}

В данном случае в запросе вместо таблицы указываем имя вызов функции с переданными ей параметрами: GetUsersByAge (@age)

В итоге результат данного запроса будет таким же, что и при выполнении скрипта выше.

### Проецирование хранимой функции на метод класса

Второй подход предполагает определение в классе контекста метода, который проецируется на хранимую функцию и через который можно вызывать данную функцию.

Например, выше была определена хранимая табличная функция GetUsersByAge, которая в качестве параметра принимает некоторое число - возраст и возвращает набор пользователей (по сути набор объектов User). Создадим для этой функции метод. Для этого изменим класс контекста следующим образом:

public class ApplicationContext : DbContext

{

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

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

public IQueryable<User> GetUsersByAge(int age) => FromExpression(() => GetUsersByAge(age));

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.HasDbFunction(() => GetUsersByAge(default));

}

}

Здесь добавлен метод GetUsersByAge(), который соответствует хранимой функции в БД. Он принимает параметр типа int и возвращает объект IQueryable<User>. Этот метод с помощью встроенного в классе DbContext метода FromExpression вызывает GetUsersByAge(age).

public IQueryable<User> GetUsersByAge(int age) => FromExpression(() => GetUsersByAge(age));

Далее в переопрепределенном методе OnModelCreating() класса контекста нам надо зарегистрировать метод GetUsersByAge с помощью вызова **HasDbFunction()**:

modelBuilder.HasDbFunction(() => GetUsersByAge(default));

Далее мы можем обратиться к хранимой функции GetUsersByAge в коде следующим образом:

using (ApplicationContext db = new ApplicationContext())

{

var users = db.GetUsersByAge(30); // обращение к хранимой функции

foreach (var u in users)

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

}

## **Хранимые процедуры**

Рассмотрим, как вызывать хранимые процедуры из кода на C# через Entity Framework Core на примере БД MS SQL Server.

Пусть наша база данных, как и в прошлой теме, описывается следующим контекстом данных и сущностями:

public class Company

{

public int Id { get; set; }

public string? Name { get; set; }

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

}

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public int Age { get; set; }

public int CompanyId { get; set; }

public Company? Company { get; set; }

## }

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

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

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

## }

Вначале создадим саму хранимую процедуру, к которой мы будем обращаться. Для этого в Visual Studio в окне **SQL Server Object Explorer** откроем узел базы данных. В узле базы данных откроем подузел Programmability -> Stored Procedures:

## 

Далее нажмем на этот узел правой кнопкой мыши и выберем **Add New Stored Procedure**:

После этого Visual Studio генерирует и октрывает файл скрипта хранимой процедуры со следующим содержимым:

CREATE PROCEDURE [dbo].[Procedure]

@param1 int = 0,

@param2 int

AS

SELECT @param1, @param2

## RETURN 0

|  |
| --- |
|  |

Изменим скрипт следующим образом:

CREATE PROCEDURE [dbo].[GetUsersByCompany]

@name nvarchar(50)

AS

SELECT \* FROM Users

WHERE CompanyId=(SELECT Id FROM Companies WHERE Name=@name)

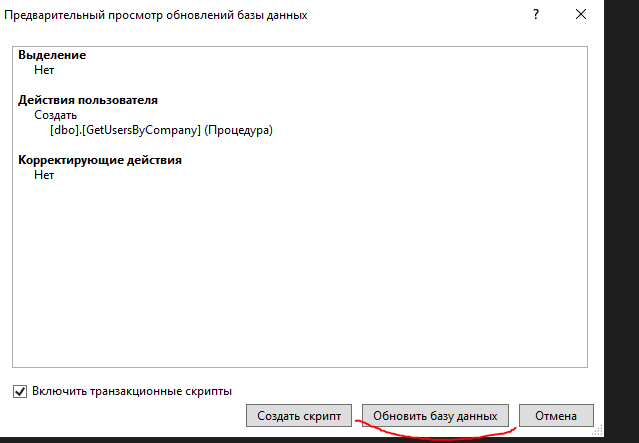
## GO

Данная процедура ищет все строки, где значение столбца название компании равно строке, переданной через параметр @name.

Далее нажмем на кнопку Update для добавления хранимой процедуры:

## 

И затем в появившемся окошке нажимаем кнопку Update Database:



После этого в узле **Stored Procedures** появится новая хранимая процедура, и мы сможем ее использовать.

Теперь обратимся к процедуре в коде C#:

using (ApplicationContext db = new ApplicationContext())

{

SqlParameter param = new("@name", "Microsoft");

var users = db.Users.FromSqlRaw("GetUsersByCompany @name", param).ToList();

foreach (var p in users)

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

}

Параметр в методе FromSqlRaw принимает название процедуры, после которого идет перечисление параметров: GetUsersByCompany @name

### Выходные параметры процедуры

В некоторых случаях процедура может возвращать не набор данных из таблиц, а какие-то отдельные данные. Как правило, для этого используются выходные параметры. Например, нам надо получить имя пользователя с максимальным возрастом. Для этого определим следующую хранимую процедуру:

CREATE PROCEDURE [dbo].[GetUserWithMaxAge]

@name varchar(50) OUTPUT

AS

SELECT @name = [Name] FROM Users WHERE Age = (SELECT MAX(Age) FROM Users)

RETURN 0

Параметр name здесь определен как выходной с ключевым словом **OUTPUT**. Через него будет передаваться имя пользователя.

Обратимся к этой процедуре в коде C#:

using (ApplicationContext db = new ApplicationContext())

{

SqlParameter param = new()

{

ParameterName = "@userName",

SqlDbType = System.Data.SqlDbType.VarChar,

Direction = System.Data.ParameterDirection.Output,

Size = 50

};

db.Database.ExecuteSqlRaw("GetUserWithMaxAge @userName OUT", param);

Console.WriteLine(param.Value);

}

Здесь параметр userName определен как выходной. Так как нам в данном случае не надо возвращать набор данных, который соответствует одной из моделей, то для выполнения запроса используется метод db.Database.ExecuteSqlRaw(). И после его выполнения через свойство **param.Value** мы сможем получить значение, переданное через параметр.