Entity Framework Core

Что такое Entity Framework Core

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

Код и пояснение

Класс User

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public string? Surname { get; set; }

public string? LivePlace { get; set; }

public string? email { get; set; }

public int Age { get; set; }

}

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

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

Класс ApplicationContext

{

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

{

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

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

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

{

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

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

}

}}

Класс Program

using (ApplicationContext db = new ApplicationContext())

{

// создаем два объекта User

User artem = new User { Name = "Artem", Surname = "Stryzhakov", LivePlace = "Keila", email = "striz@gmail.com", Age = 18 };

User jaan = new User { Name = "Jaan", Surname = "Krohhin", LivePlace = "Tallinn", email = "krohha@gmail.com", Age = 20 };

// добавляем их в бд

db.Users.Add(artem);

db.Users.Add(jaan);

db.SaveChanges();

Console.WriteLine("Objektide salvestamine õnnestus");

// получаем объекты из бд и выводим на консоль

var users = db.Users.ToList();

Console.WriteLine("Objektide loend:");

foreach (User u in users)

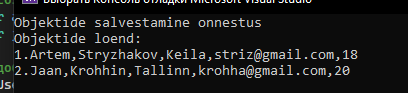
{

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

}

**}**

Результат



Подключение к существующей базе данных

Класс helloappContext

public partial class helloappContext : DbContext

{

public helloappContext()

{

}

public helloappContext(DbContextOptions<helloappContext> options)

: base(options)

{

}

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

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

if (!optionsBuilder.IsConfigured)

{

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

}

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

OnModelCreatingPartial(modelBuilder);

}

partial void OnModelCreatingPartial(ModelBuilder modelBuilder);

}

}

Переработанный класс Program

using (helloappContext db = new helloappContext())

{

// получаем объекты из бд и выводим на консоль

var users = db.Users.ToList();

Console.WriteLine("objektide loend:");

foreach (User u in users)

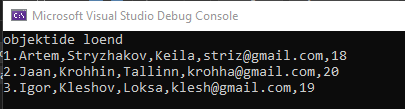
{

Console.WriteLine($"{u.Id}.{u.Name},{u.Surname},{u.LivePlace},{u.email},{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.

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("База данных доступна");

else Console.WriteLine("База данных не доступна");

}

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",Surname = "Tom",LivePlace = "Poland",email = "Tom@gmail.com", Age = 19 };

User alice = new User { Name = "Alice", Surname = "Alice", LivePlace = "Venemaa", email = "Alice@mail.ru", 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.Surname},{u.LivePlace},{u.email},{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;

user.email = "bob@gmail.com";

user.Surname = "Bobikov";

user.LivePlace = "USA";

//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.Surname},{u.LivePlace},{u.email},{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)

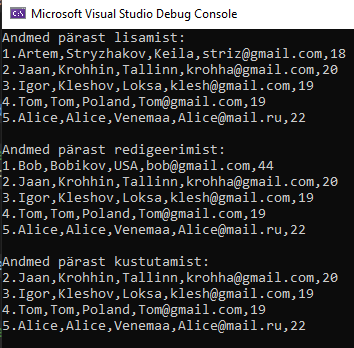
{

Console.WriteLine($"{u.Id}.{u.Name},{u.Surname},{u.LivePlace},{u.email},{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",Surname = "Tom",LivePlace = "Poland",email = "Tom@gmail.com", Age = 19 };

User alice = new User { Name = "Alice", Surname = "Alice", LivePlace = "Venemaa", email = "Alice@mail.ru", 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",Surname = "Tom",LivePlace = "Poland",email = "Tom@gmail.com", Age = 19 };

User alice = new User { Name = "Alice", Surname = "Alice", LivePlace = "Venemaa", email = "Alice@mail.ru", Age = 22 };

// Добавление

db.Users.Add(tom);

db.Users.Add(alice);

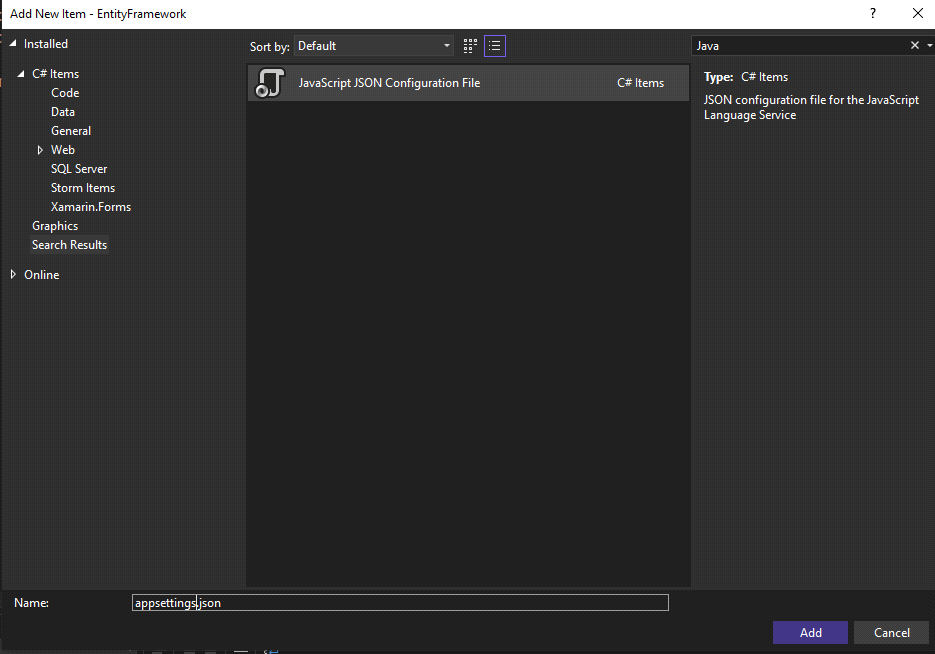
db.SaveChanges();

}

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": {

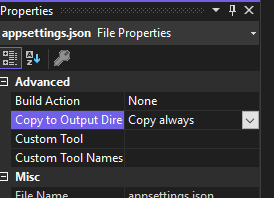
"DefaultConnection": "Data Source=helloapp.db"

}

}

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",LivePlace = "Poland",email = "Tom@gmail.com", Age = 19 };

User alice = new User { Name = "Alice", Surname = "Alice", LivePlace = "Venemaa", email = "Alice@mail.ru", Age = 22 };

// Добавление

db.Users.Add(tom);

db.Users.Add(alice);

db.SaveChanges();

}

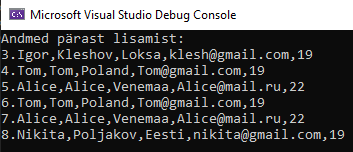
Для создания конфигурации применяется класс ConfigurationBuilder. Метод AddJsonFile() добавляет все настройки из файла конфигурации. С помощью метода Build() создается объект конфигурации, из которого мы можем получить строку подключения:

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 = "Kirill", Age = 21 };

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

db.Users.Add(user1);

db.Users.Add(user2);

db.SaveChanges();

var users = db.Users.ToList();

Console.WriteLine("kasutajate loend:");

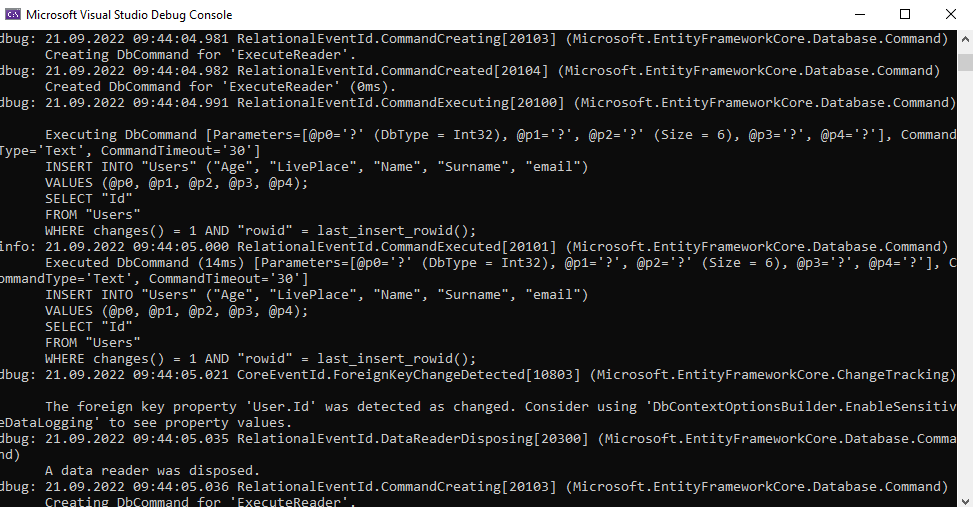
foreach (User u in users)

{

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()

{

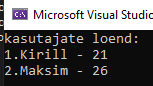
await base.DisposeAsync();

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)

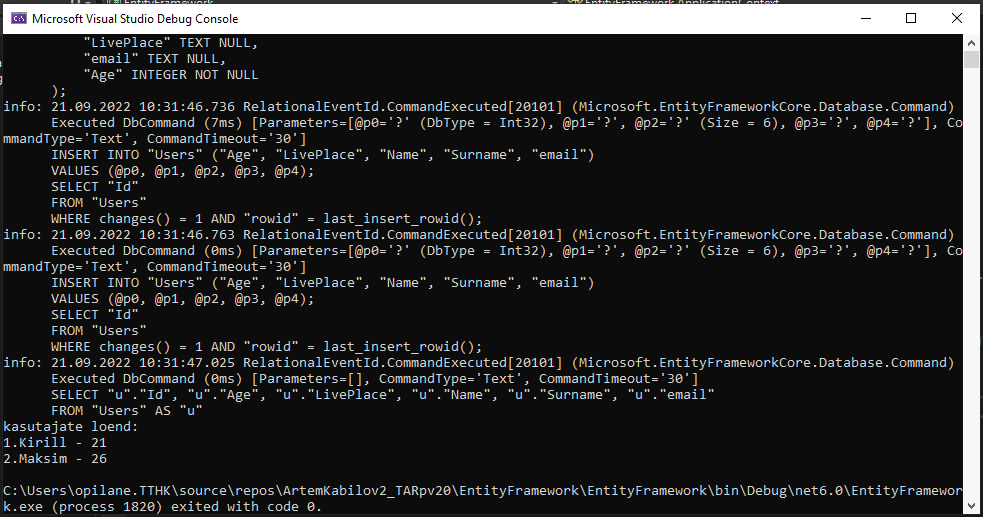
{

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();

}

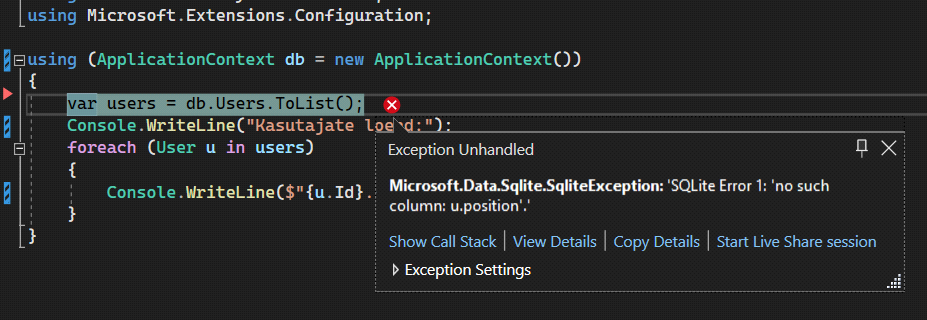
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

}

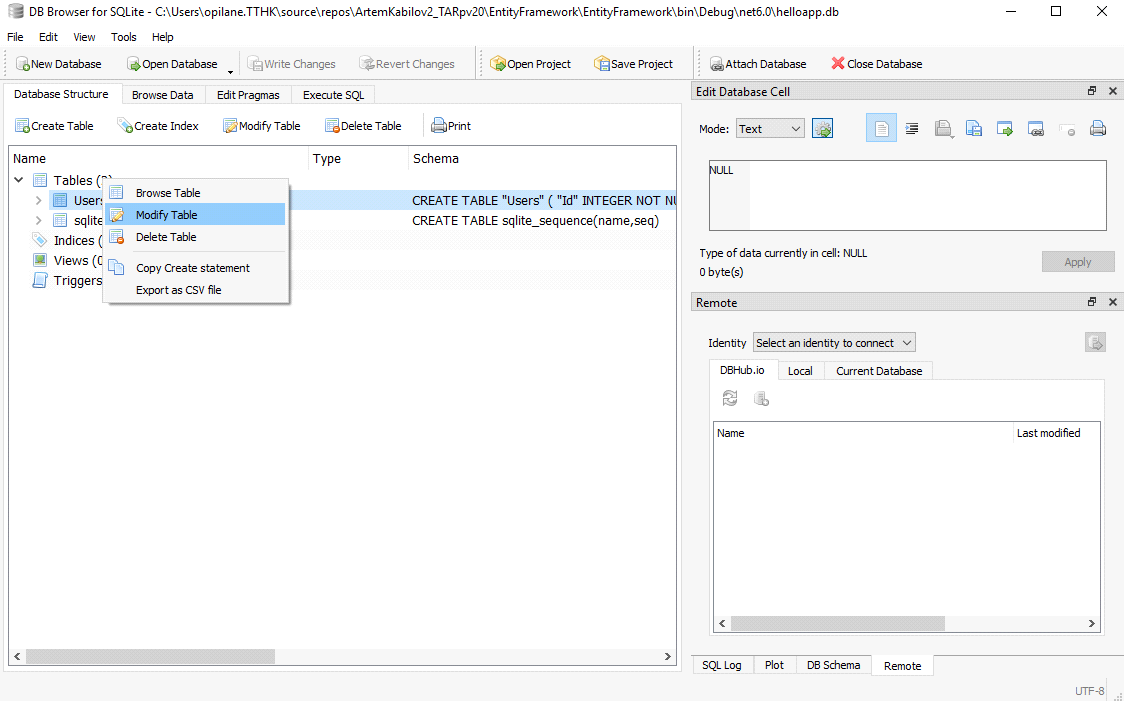
}



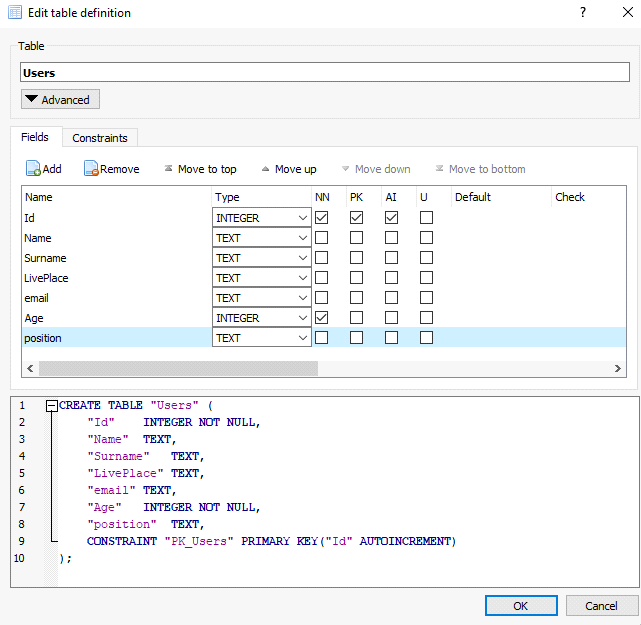
Ручное изменение базы данных

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 foHYPERLINK "https://sqlitebrowser.org/"rHYPERLINK "https://sqlitebrowser.org/" 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");

}

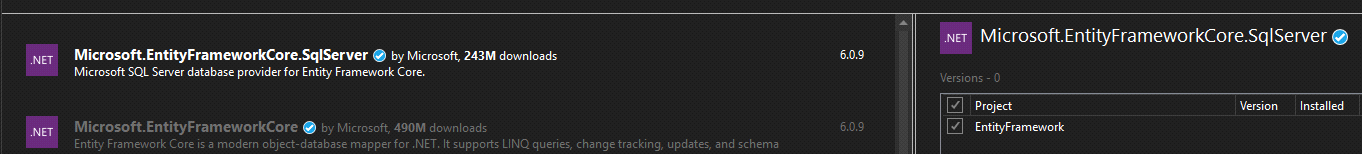
}

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.

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();

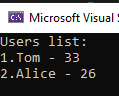
Console.WriteLine("Users list:");

foreach (User u in users)

{

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

}}



MySQL

Определим контекст данных - класс ApplicationContext:

public class ApplicationContext : DbContext

{

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

public ApplicationContext()

{

Database.EnsureCreated();

}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

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

}

}

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

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

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

using HelloApp;

using (ApplicationContext db = new ApplicationContext())

{

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

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

db.Users.AddRange(user1, user2);

db.SaveChanges();

}

// получение данных

using (ApplicationContext db = new ApplicationContext())

{

var users = db.Users.ToList();

Console.WriteLine("Список объектов:");

foreach (User u in users)

{

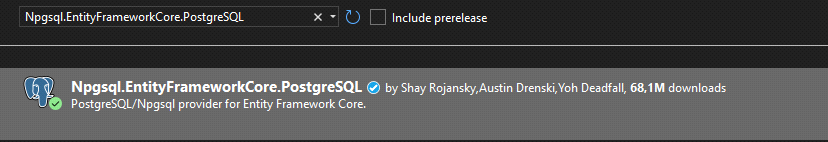
Console.WriteLine($"{u.Id}.{u.Name} - {u.Age}");

}

}

PostgreSQL

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



Определим класс контекста данных:

using HelloApp;

using Microsoft.EntityFrameworkCore;

public class ApplicationContext : DbContext

{

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

public ApplicationContext()

{

Database.EnsureCreated();

}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

}

}

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

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

// добавление данных

using HelloApp;

using (ApplicationContext db = new ApplicationContext())

{

// создаем два объекта User

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

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

// добавляем их в бд

db.Users.AddRange(user1, user2);

db.SaveChanges();

}

// получение данных

using (ApplicationContext db = new ApplicationContext())

{

// получаем объекты из бд и выводим на консоль

var users = db.Users.ToList();

Console.WriteLine("Users list:");

foreach (User u in users)

{

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

}

}



**Миграции**

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

public class ApplicationContext : DbContext

{

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

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

}

}

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

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

Add-Migration Initial

После генерации файла миграции для создания базы данных выполним команду:

Update-Database

**Подключение к существующей базе данных**

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

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

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

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

Создание модели в Entity Framework Core

Fluent API и аннотации данных

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

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

**Fluent API**

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

public class ApplicationContext : DbContext

{

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

public ApplicationContext()

{

Database.EnsureDeleted();

Database.EnsureCreated();

}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

// использование Fluent API

base.OnModelCreating(modelBuilder);

}

}

**Аннотации**

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

using System.ComponentModel.DataAnnotations.Schema;

public class User

{

[Column("user\_id")]

public int Id { get; set; }

public string? Name { get; set; }

public int Age { get; set; }

public string Gender { get; set; }

}

Определение моделей

**Включение сущностей в модель**

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

public class ApplicationContext : DbContext

{

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

public ApplicationContext()

{

Database.EnsureDeleted();

Database.EnsureCreated();

}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

}

}

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public int Age { get; set; }

public string Gender { get; set; }

}

**Включение сущностей в модель без DbSet**

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

Например, пусть у нас определены следующие сущности:

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public int Age { get; set; }

public string Gender { get; set; }

// навигационное свойство

public Company? Company { get; set; }

}

public class Company

{

public int Id { get; set; }

public string? Name { get; set; }

}

public class Country

{

public int Id { get; set; }

public string? Name { get; set; }

}

И пусть у нас будет класс контекста данных:

using Microsoft.EntityFrameworkCore;

public class ApplicationContext : DbContext

{

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

public ApplicationContext()

{

Database.EnsureDeleted();

Database.EnsureCreated();

}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

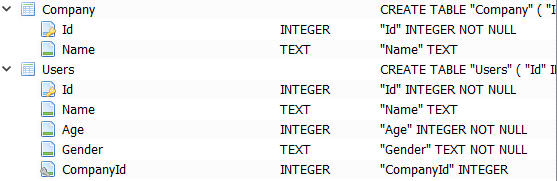
{

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

}

}

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



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

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

using Microsoft.EntityFrameworkCore;

public class ApplicationContext : DbContext

{

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

public ApplicationContext()

{

Database.EnsureDeleted();

Database.EnsureCreated();

}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

}

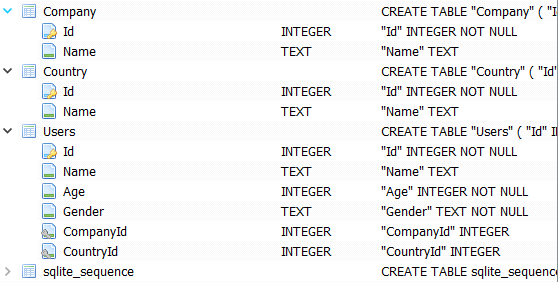
protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Country>();

}

}



Свойства сущности

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

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

using Microsoft.EntityFrameworkCore;

public class ApplicationContext : DbContext

{

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

public ApplicationContext()

{

Database.EnsureDeleted();

Database.EnsureCreated();

}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

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

}

}

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

Исключение с помощью аннотаций данных:

using System.ComponentModel.DataAnnotations.Schema;

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

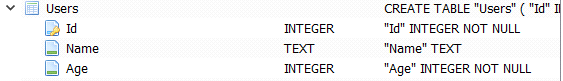
public int Age { get; set; }

[NotMapped]

public string? Address { get; set; }

}

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



**Использование полей класса**

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

public class User

{

string name;

public int Id { get; set; }

public string Name

{

get { return name; }

set { name = value; }

}

public int Age { get; set; }

public string Gender { get; set; }

}

Конструкторы сущностей

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

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

public class User

{

public int Id { get; set; }

public string Name { get; set; }

public int Age { get; set; }

public User(string name, int age)

{

Name = name;

Age = age;

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

}

}

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

Пусть у нас будет стандартный контекст данных:

using Microsoft.EntityFrameworkCore;

public class ApplicationContext : DbContext

{

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

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

}

}

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

using (ApplicationContext db = new ApplicationContext())

{

db.Database.EnsureDeleted();

db.Database.EnsureCreated();

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

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

db.Users.Add(tom);

db.Users.Add(bob);

db.SaveChanges();

}

using (ApplicationContext db = new ApplicationContext())

{

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

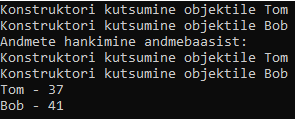
var users = db.Users.ToList();

foreach (var user in users)

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

}

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



Здесь надо учитывать несколько моментов:

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

public class User

{

public int Id { get; set; }

public string Name { get; set; }

public int Age { get; set; }

public Company? Company { get; set; }

public User(string name, int age)

{

Name = name;

Age = age;

}

}

public class Company

{

public int Id { get; set; }

public string Name { get; set; }

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

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

}

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

public class User

{

public int Id { get; set; }

public string Name { get; set; }

public int Age { get; set; }

public User(string name)

{

Name = name;

Age = 18;

}

public User(string name, int age)

{

Name = name;

Age = age;

}

}

Сопоставление таблиц и столбцов

**Сопоставление таблиц**

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

*Атрибут Table*

Атрибут Table позволяет переопределить сопоставление с таблицей по имени:

[Table("People")]

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

}

Теперь сущность User будет сопоставляться с таблицей "People".

*Метод ToTable*

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

public class ApplicationContext : DbContext

{

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

public ApplicationContext()

{

Database.EnsureDeleted();

Database.EnsureCreated();

}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

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

}

}

**Сопоставление столбцов**

По умолчанию каждое свойство сопоставляется с одноименным столбцом.

*Атрибут Column*

Атрибут **Column** переопределяет сопоставление:

public class User

{

[Column("user\_id")]

public int Id { get; set; }

public string? Name { get; set; }

public string? Gender { get; set; }

}

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

*Метод HasColumnName*

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

using Microsoft.EntityFrameworkCore;

public class ApplicationContext : DbContext

{

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

public ApplicationContext()

{

Database.EnsureDeleted();

Database.EnsureCreated();

}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

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

}

}

Обязательные и необязательные свойства

**Обязательные свойства**

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

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

Например, возьмем следующую модель:

public class User

{

public int Id { get; set; }

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

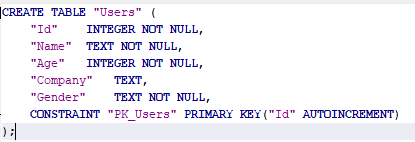
public string? Company { get; set; }

public int Age { get; set; }

public string Gender { get; set; }

}

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:

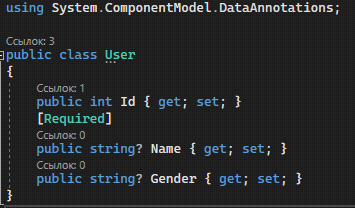


Здесь мы видим, что столбец Company допускает значение NULL, а столбец Name - не допускает благодаря установке атрибута **NOT NULL**. Хотя здесь приведен пример бд SQLite, но для других систем баз данных будет действовать аналогичная логика.

Атрибут Required

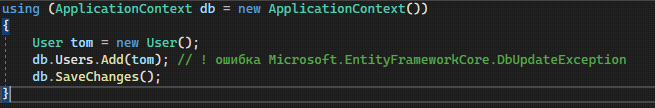
Атрибут **Required** указывает, что данное свойство обязательно для установки, то есть будет иметь определение NOT NULL в БД, даже если оно представляет nullable-тип:

using System.ComponentModel.DataAnnotations;



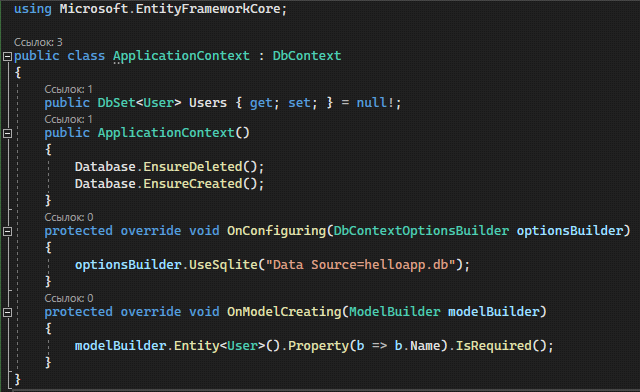
А столбец Name в базе данных будет определен как NOT NULL.

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**:



Метод IsRequired

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

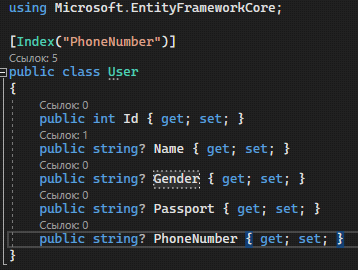


Настройка индексов

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.

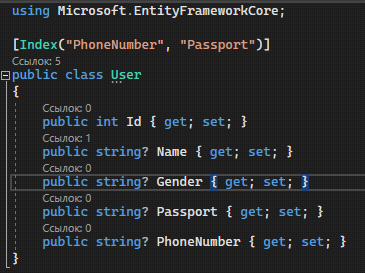
**Настройка индексов с помощью атрибутов**

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

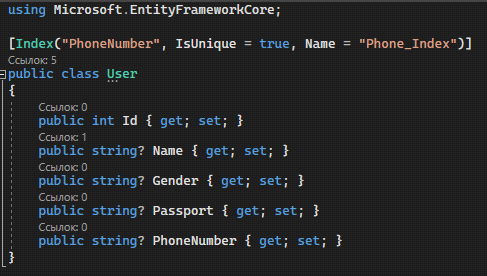


Первый и обязательный параметр атрибута указывает на свойство (или набор свойств), с которым будет ассоциирован индекс. В данном случае это свойство PhoneNumber.

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



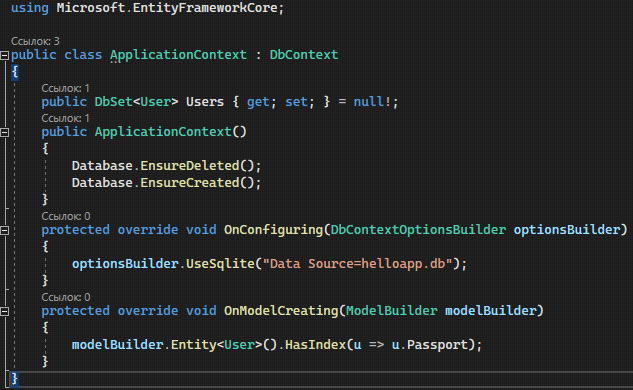
С помощью дополнительных параметров можно настроить уникальность и имя индекса:



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

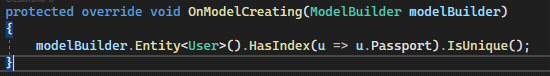
**Настройка индексов с помощью Fluent API**

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



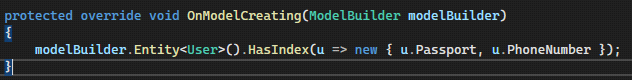
**Уникальность индексов**

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:



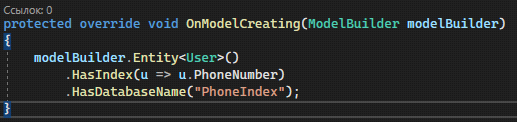
**Составные индексы**

Также можно определить индексы сразу для нескольких свойств:



**Имя индекса**

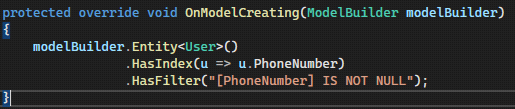
Для установки имени индекса применяется метод **HasDatabaseName()**, в который передается имя индекса:



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

**Фильтры индексов**

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



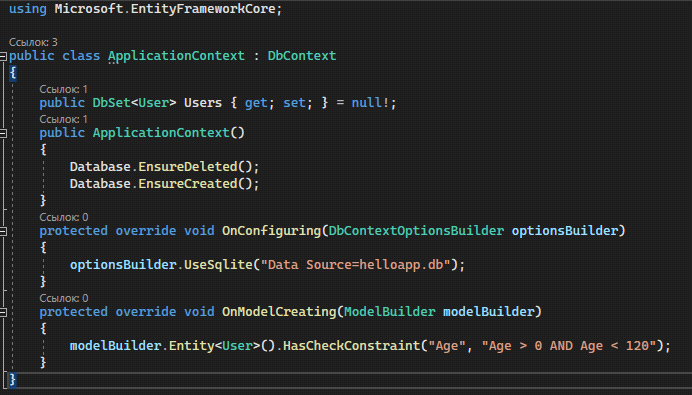
В данном случае в качестве индекса будет использоваться столбец PhoneNumber. Причем только для тех строк, у которых в столбце PhoneNumber значение не равно NULL.

Ограничения свойств

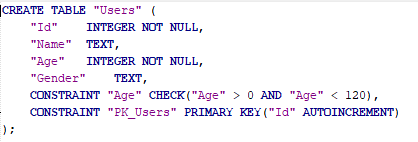
**Установка ограничений**

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

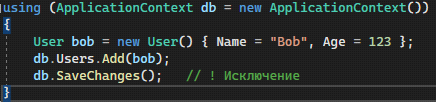
В метод **HasCheckConstraint()** передается название столбца и sql-выражение, которое выполняет проверку корректности передаваемого значения. Например, у пользователя есть возраст, который должен укладываться в некоторые допустимые рамки. К примеру, возраст не может быть меньше 0 и больше 120:



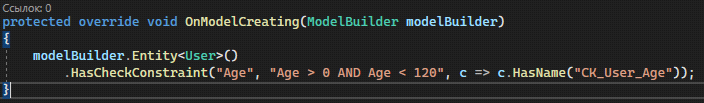
В этом случае на уровне базы данных SQLite для сущности User будет создаваться следующая таблица:



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



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



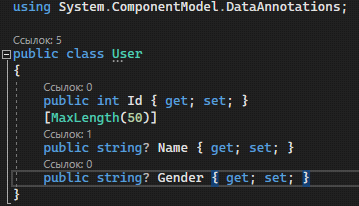
Ограничения по длине

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**:

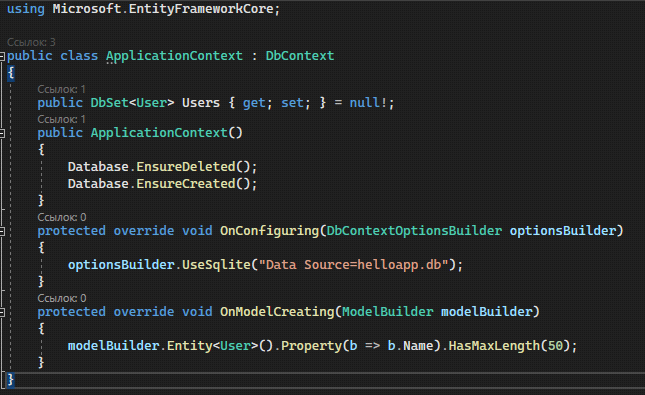


Стоит отметить, что данное ограничение будет действовать только для тех систем баз данных, которые поддерживают данную возможность. Например, для бд SQLite это не будет иметь никакого значения. А в случае с бд MS SQL Server столбец Name в базе данных будет иметь тип nvarchar(50) и тем самым иметь ограничение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

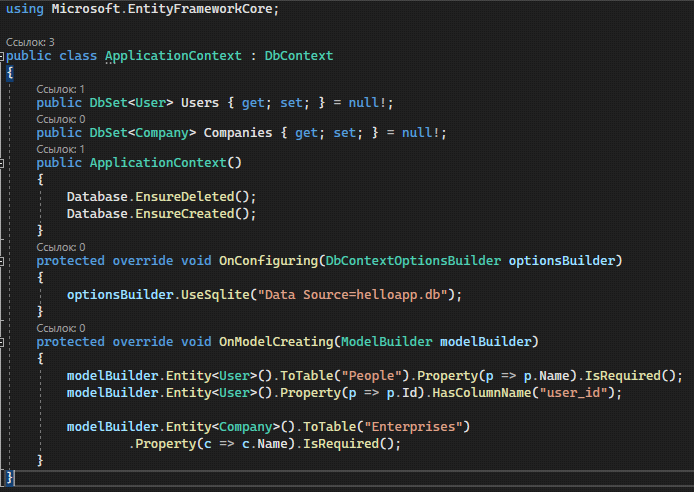


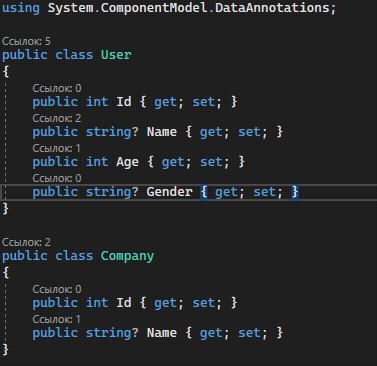
Конфигурация моделей

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.

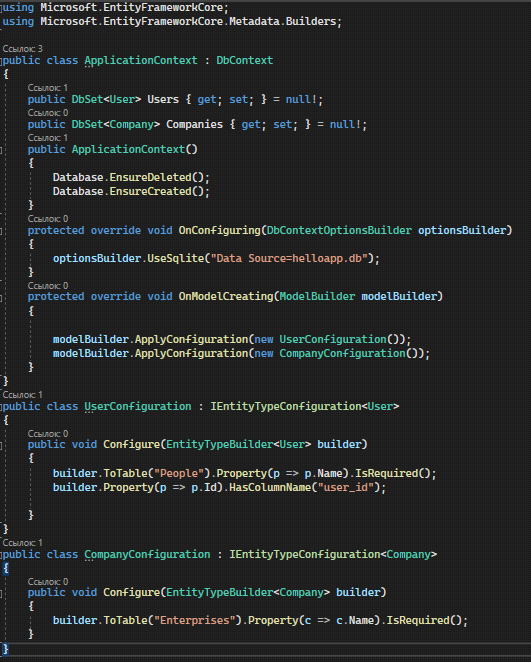
Для вынесения конфигурации во вне необходимо создать класс конфигурации, реализующий интерфейс **EntityTypeConfiguration<T>**.

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



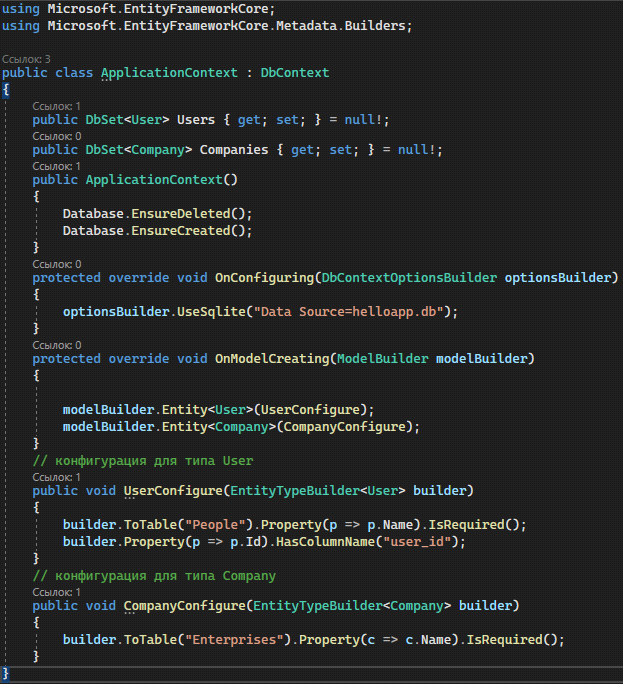


Вся конфигурация здесь определена в методе **OnModelCreating()**. В принципе он не содержит много кода, однако при наличии гораздо большего количества сущностей и более изощренной их конфигурации с помощью Fluent API данный метод мог бы сильно раздуться в размерах. И теперь изменим определение контекста,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



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.

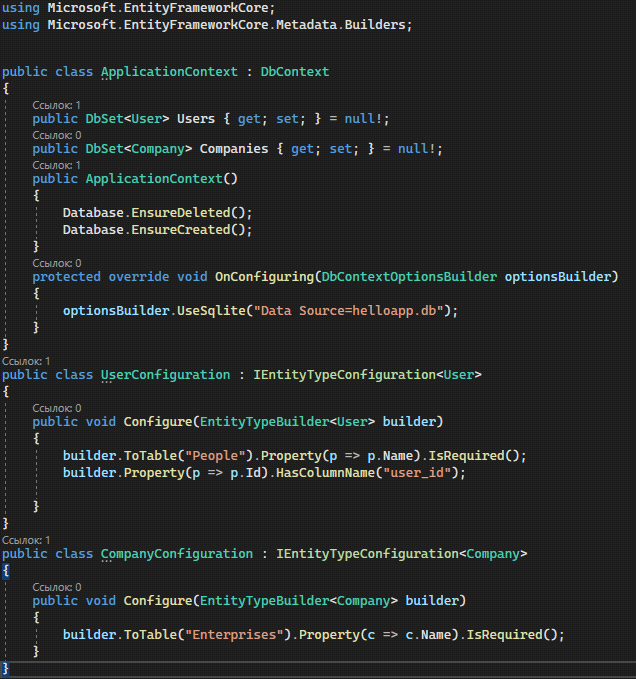
В качестве альтернативы мы могли бы использовать еще один вариант. Вместо выделения отдельных классов конфигураций определить конфигурацию в виде отдельных методов в том же классе контекста данных:

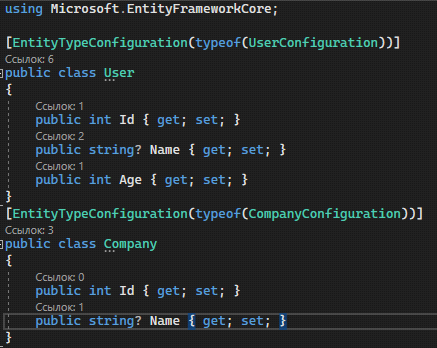


Здесь конфигурация определяется для каждого типа в отдельном методе, который в качестве параметра принимает объект EntityTypeBuilder<T>. Затем метод передается в вызов modelBuilder.Entity<T>() для соответствующей модели.

Атрибут EntityTypeConfiguration

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

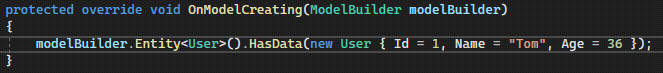




Инициализация БД начальными данными

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:



Например, инициализируем БД набором данных:



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

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

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

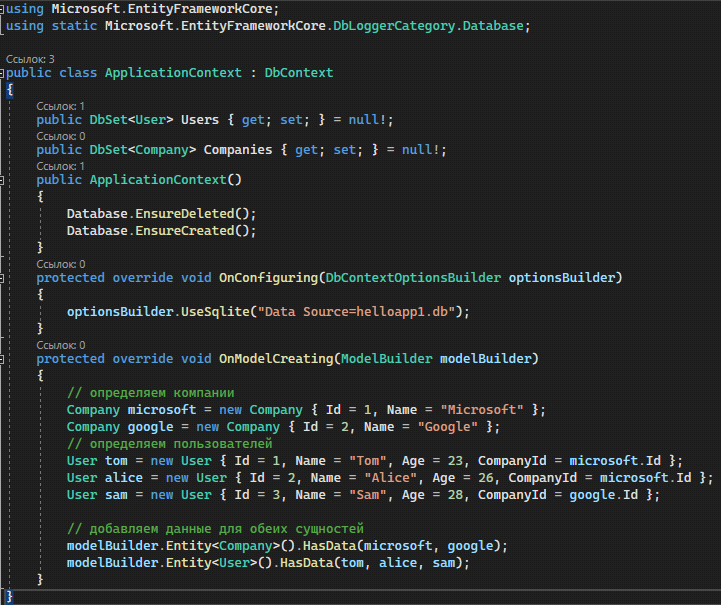
* При выполнении миграции. (При создании миграции добавляемые данные автоматически включаются в скрипт миграции)
* При вызове метода **Database.EnsureCreated()**, который создает БД при ее отсутствии

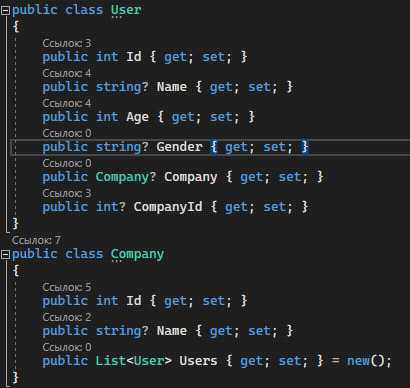
В случае выше в конструкторе применяется метод **Database.EnsureCreated()**, поэтому при создании контекста данных



будет автоматически производиться инициализация бд начальными данными.

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

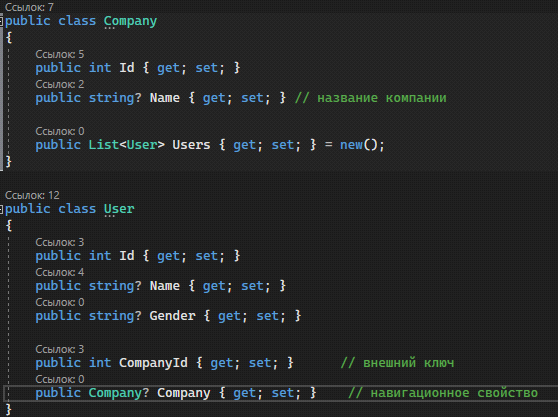




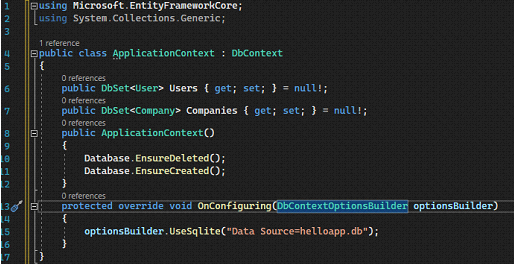
**Отношения между сущностями**

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:



И пусть у нас будет следующий контекст данных:



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:

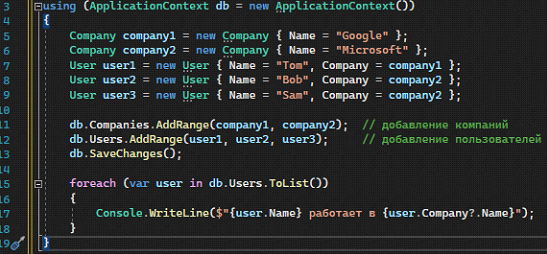
⦁ Имя\_навигационного\_свойства+Имя ключа из связанной сущности - в нашем случае имя авигационного свойства Company, а ключа из модели Company - Id, поэтому в нашем случае нам надо обозвать свойство CompanyId, что собственно и было сделано в вышеприведенном коде.

⦁ 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.

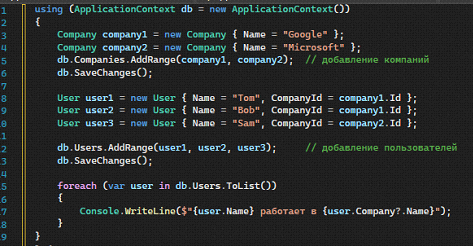
Установка главной сущности по навигационному свойству зависимой сущности

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



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

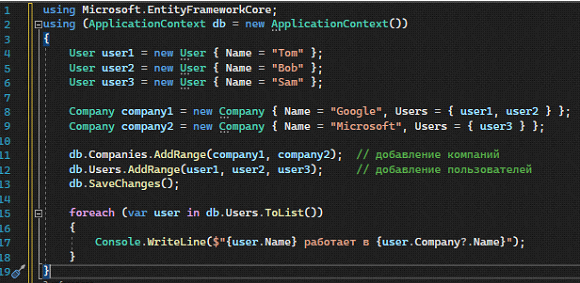
Также можно использовать свойство-внешний ключ для установки связи:



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.

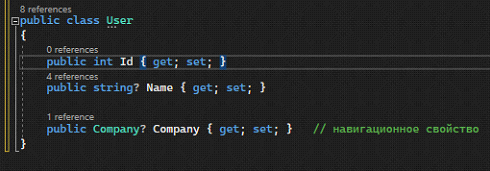
*Установка зависимой сущности через навигационное свойство главной сущности*

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:



*Отсутствие свойства внешнего ключа и навигационного свойства*

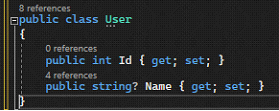
Нам необязательно определять внешний ключ в зависимой сущности. Его можно опустить:



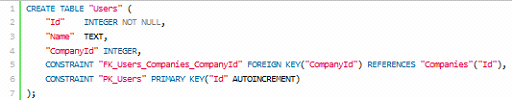
В этом случае Entity Framework сам автоматически сгенерирует столбец для внешнего ключа в таблице Users.

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:



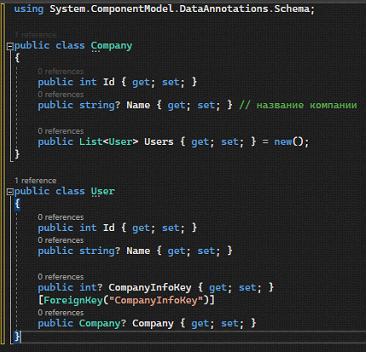
Но за счет того, что в классе Company также определено навигационное свойство Users все равно будет создаваться внешний ключ и связь таблицы Users и таблицы Companies. В частности, тогда в случае БД SQLite определение таблицы Users будет выглядеть следующим образом:



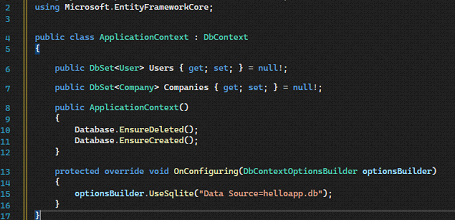
Настройка внешнего ключа через аннотации данных и Fluent API

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]:

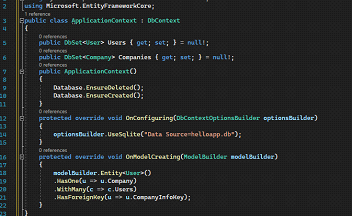


И пусть будет следующий контекст данных:



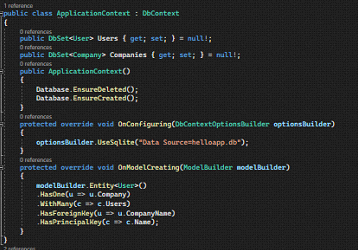
Настройка ключа с помощью Fluent API

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



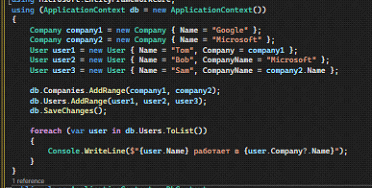
Установка в качестве внешнего ключа произвольного свойства

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



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.

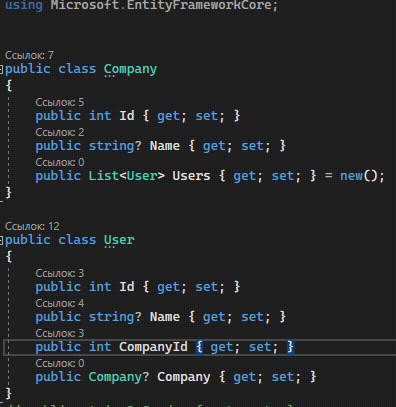
В программе при добавлении объектов в БД в этом случае можно установить как навигационное свойство, так и свойство внешнего ключа:

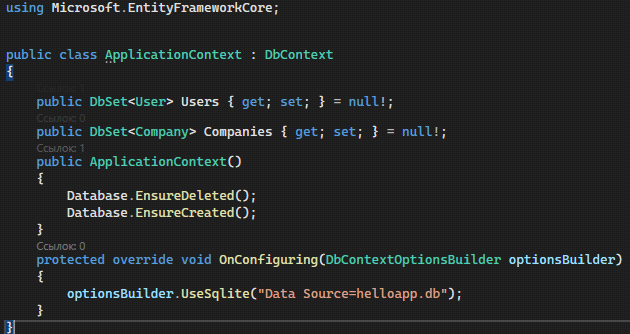


Каскадное удаление

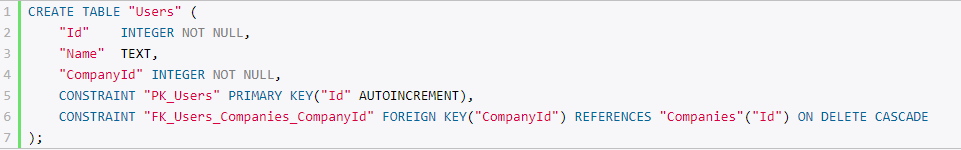
Каскадное удаление представляет автоматическое удаление зависимой сущности после удаления главной.

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



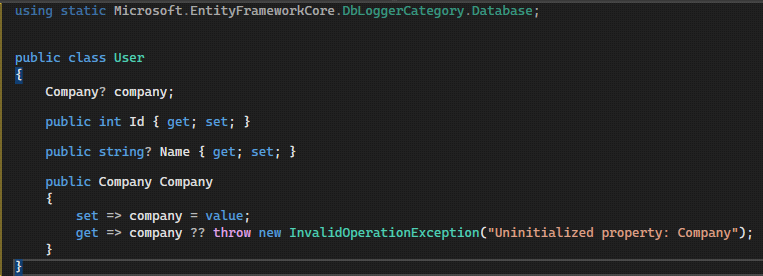


Здесь свойство внешнего ключа имеет тип int, оно не допускает значения null и требует наличия конкретного значения - id связанного объекта Company (При этом то, что навигационное свойство Company допускает null, не имеет значения). То есть для объекта User обязательно необходимо наличия связанного объекта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

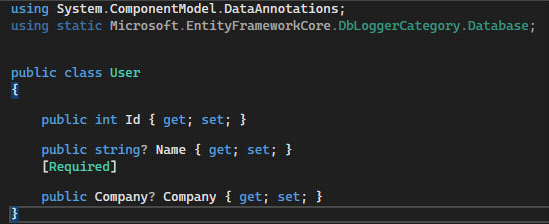


В определении внешнего ключа устанавливается каскадное удаление: ON DELETE CASCADE

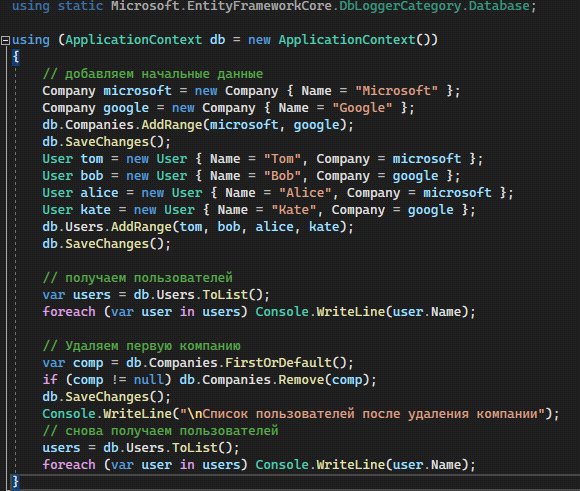
Аналогичная связь будет устанавливаться, если свойство-внешний ключа отсутствует, а навигационное свойство НЕ представляет nullable-тип:



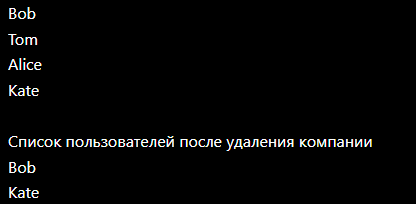
Такая же таблица создается, если навигационное свойство представляет nullable-тип, но оно определено как обязательное, например, с помощью атрибута **Required**:



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

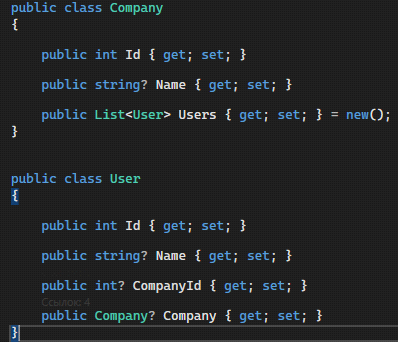


Консольный вывод программы:

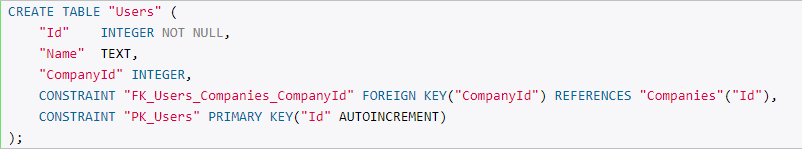


Удаление главной сущности - компании привело к удалению двух зависимых сущностей - пользователей.

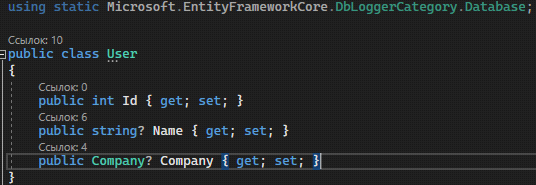
Теперь изменим модели, указав необязательность наличия объекта Company:



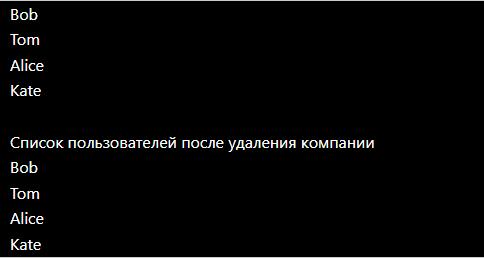
Теперь внешний ключ имеет тип Nullable<int>, то есть он допускает значение null. Когда пользователь не будет принадлежать ни одной компании, это свойство будет иметь значение null. И в этом случае скрипт таблицы Users будет выглядеть следующим образом:



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

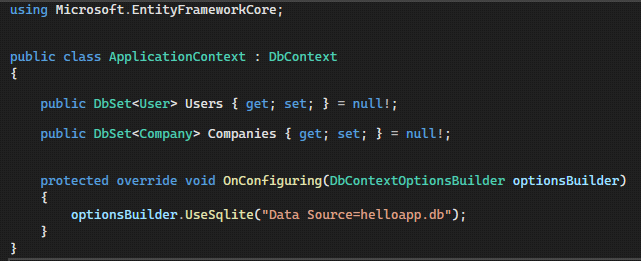


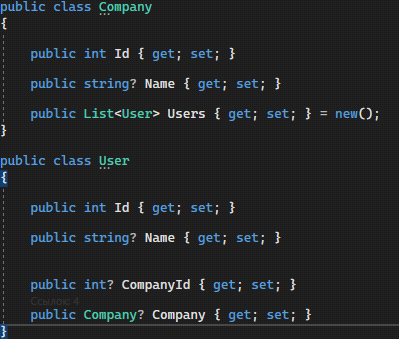
Если мы запустим ту же самую программу, то получим уже другой консольный вывод:



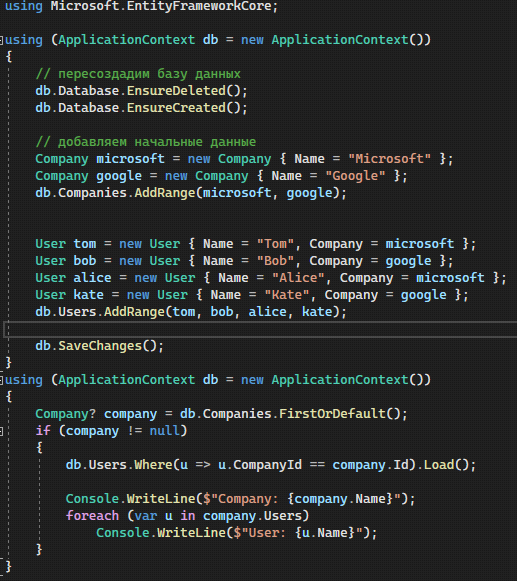
Explicit loading

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



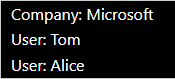


Загрузим данные по первой компании:



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. После этого нам не надо подгружать связанные данные, так как они уже есть в контексте.

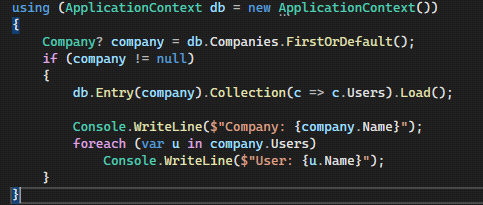
Консольный вывод программы:



Важно, что здесь подгружаются только те пользователи, которые непосредственно связаны с компанией. Если нам надо загрузить в контекст вообще все объекты из таблицы Users, то можно было бы использовать следующее выражение db.Users.Load()

Для загрузки связанных данных мы также можем использовать

методы **Collection()** и **Reference**. Метод **Collection** применяется, если навигационное свойство представляет коллекцию:

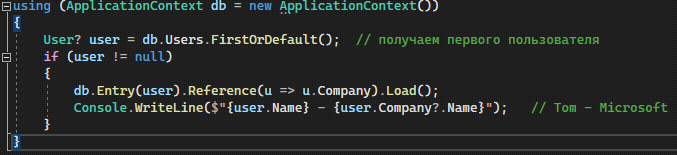


На уровне базы данных вызов db.Entry(company).Collection(t=>t.Users).Load() будет трансформироваться в следующую команду SQL:



@\_\_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**:



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



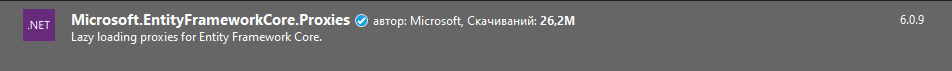
где @\_\_p\_0 в данном случае это автоматически передаваемый параметр, который представляет свойство CompanyId выбранного пользователя.

Lazy loading

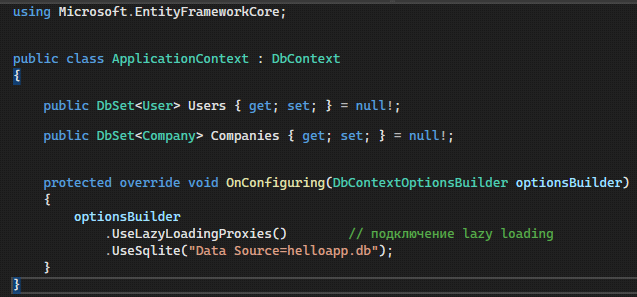
**Lazy loading** или ленивая загрузка предполагает неявную автоматическую загрузку связанных данных при обращении к навигационному свойству. Однако здесь есть ряд условий:

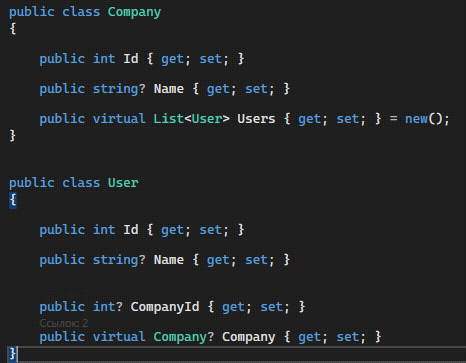
* При конфигурации контекста данных вызвать метод **UseLazyLoadingProxies()**
* Все навигационные свойства должны быть определены как виртуальные (то есть с модификатором **virtual**), при этом сами классы моделей должны быть открыты для наследования

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



Далее определим следующие модели и контекст данных:

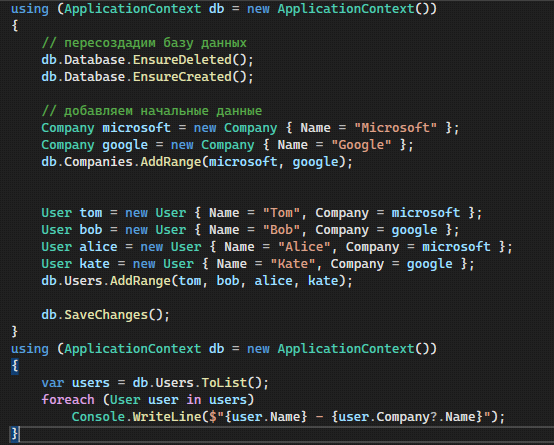




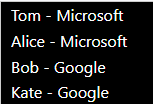
Прежде всего в методе OnConfiguring у объекта DbContextOptionsBuilder вызывается метод **UseLazyLoadingProxies()**, который делает доступной ленивую загрузку.

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:



Консольный вывод:



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:



@\_\_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:

// 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"

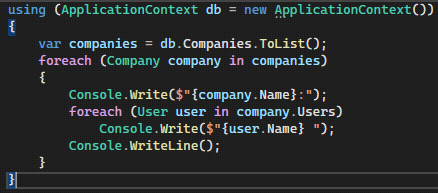
WHERE "c"."Id" = @\_\_p\_0

Bob - Google

// компания этого пользователя уже в контексте, не надо выполнять новый запрос

Kate - Google

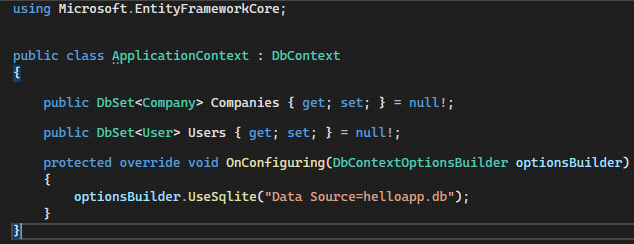
Таким же образом можно загрузить компании и связанных с ними пользователей:

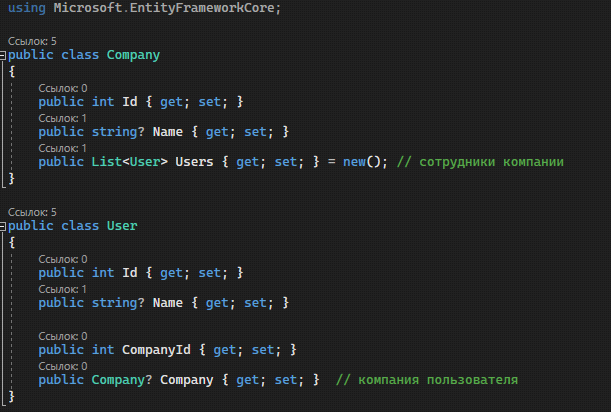


Однако при использовании lazy loading следует учитывать что если в базе данных произошли какие-нибудь изменения, например, другой пользователь изменил данные, то данные не перезагужаются, контекст продолжает использовать те данные, которые были ранее загружены, как собственно было показано выше.

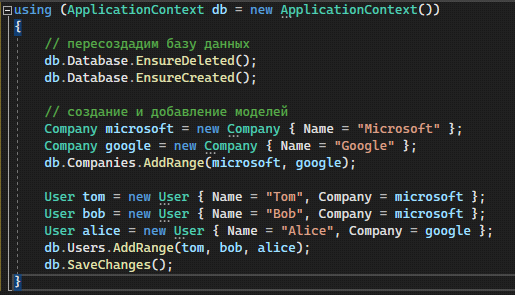
Отношение один ко многим

Отношение один-ко-многим (**one-to-many**) представляет ситуацию, когда одна сущность хранит ссылку на один объект другой сущности, а вторая сущность может ссылаться на коллекцию объектов первой сущности. Например, в одной компании может работать несколько сотрудников, а каждый сотрудник в свою очередь может официально работать только в одной компании:

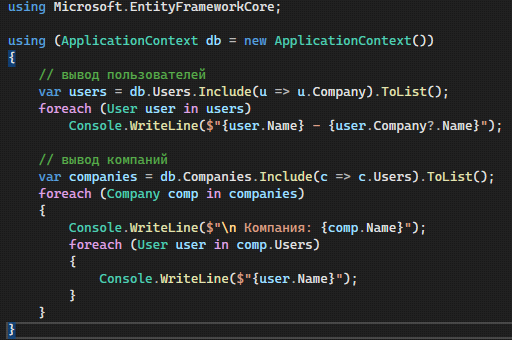




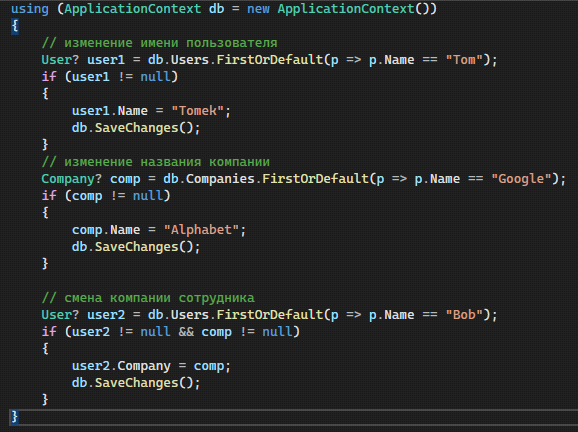
Добавление данных:



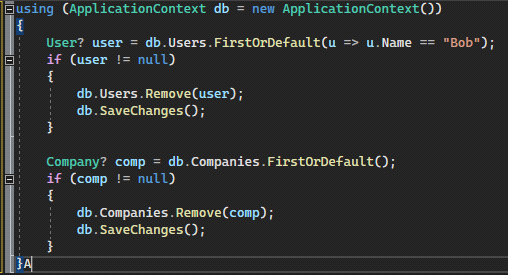
Получение данных:



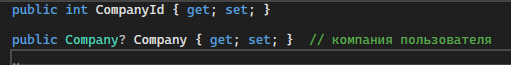
Редактирование:



Удаление:



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:



Соответственно при удалении компании будут удалены и все связанные с ней пользователи.

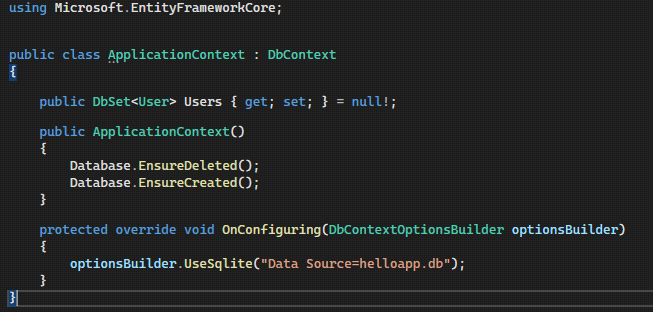
Комплексные типы

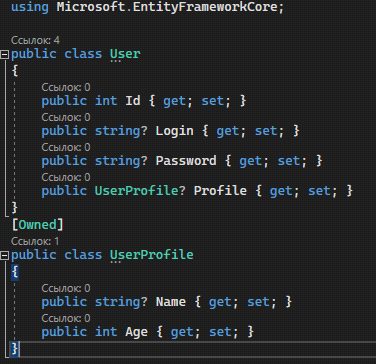
Обычно Entity Framework Core сопоставляет каждый класс с отдельной таблицей. Однако бывают случаи, когда определенный класс существует не сам по себе, а несет некоторую дополнительную информацию по отношению к другой главной модели. Данная функциональность предполагает, что у нас есть несколько типов, 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

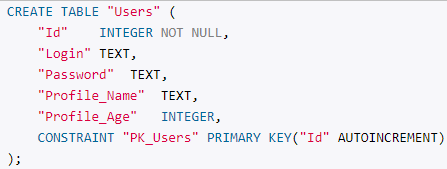
Атрибут **OwnedAttribute** позволяет установить зависимый тип. Например:



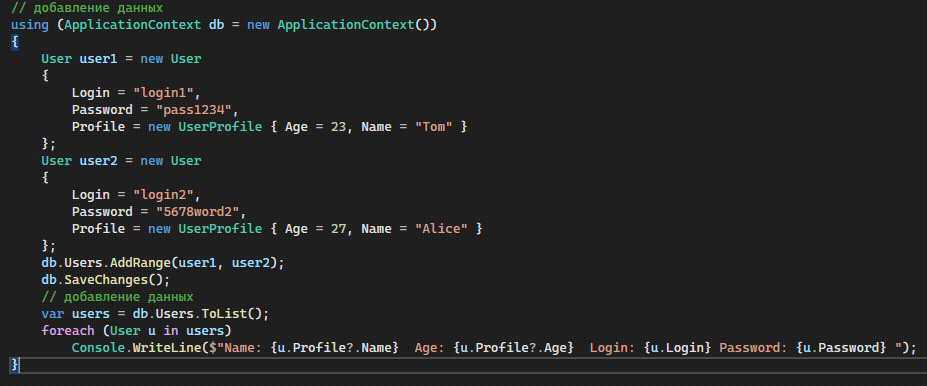


В данном случае есть тип-владелец User и есть владеемый им тип UserProfile. Класс User определяет основные настройки учетной записи - логин и пароль, а UserProfile - всторостепенные данные - имя, возраст и т.д. Чтобы указать, что User владеет типом UserProfile, над UserProfile указывается атрибут OwnedAttribute.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

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



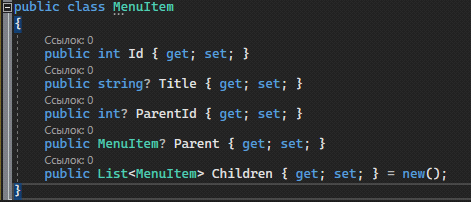
Использование типов:



Иерархические данные

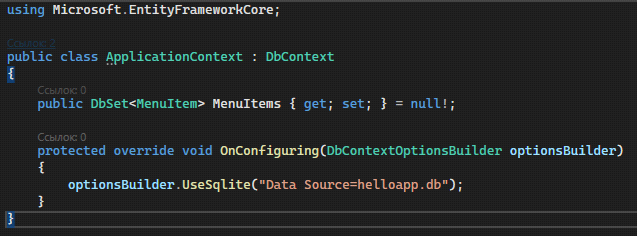
Некоторые данные представляют иерархическую структуру по типу деревьев, например, меню, где один пункт меню может иметь подменю. Или файловая система, где в каталоге могут быть другие каталоги. Рассмотрим, как в Entity Framework мы можем создать и использовать подобную иерархическую структуру.

Определим следующую сущность MenuItem:

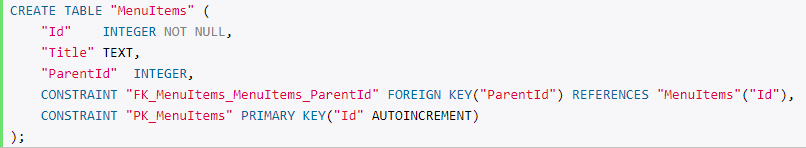


MenuItem представляет пункт меню, который может иметь подменю. Свойство Title представляет некоторый текст пункта меню. Свойство Parent хранит ссылку на родительский пункт меню при его наличии, а ParentId - id родительского пункта меню. Причем поскольку у пунктов меню верхнего уровня не может быть родительских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

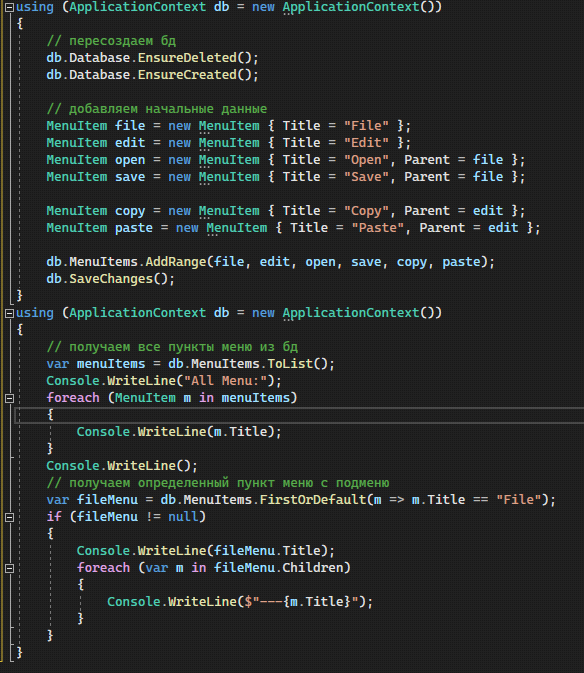
И определим следующий контекст данных:



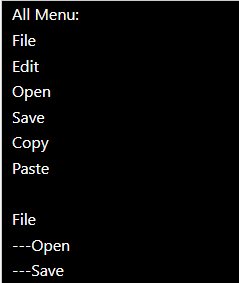
В итоге при создании базы данных будет генерироваться следующая таблица:



Использование MenuItem:



Консольный вывод программы:

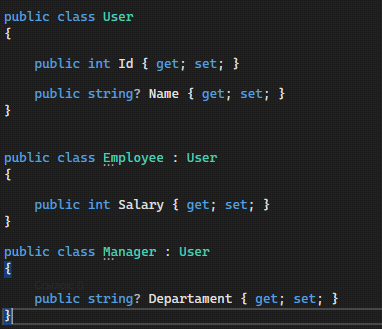


Наследование

Подход TPH - Table Per Hierarchy

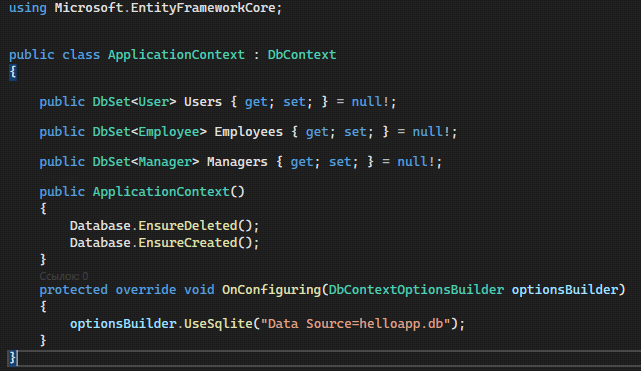
По умолчанию при работе с цепочками наследования классов Entity Framework Core использует подход TPH (Table Per Hierarchy / Таблица на одну иерархию классов). При использовании данного подхода TPH для всех классов из одной иерархии в базе данных создается одна таблица. А чтобы определить, к какому именно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

Например, у нас есть следующая иерархия классов:



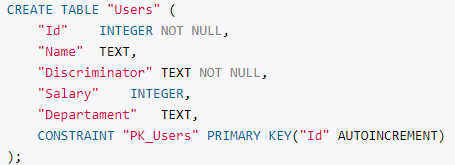
Есть базовый класс User, который представляет пользователя и от которого наследуются класс Employee - класс работника и Manager - класс управляющего.

Определим контекст данных:

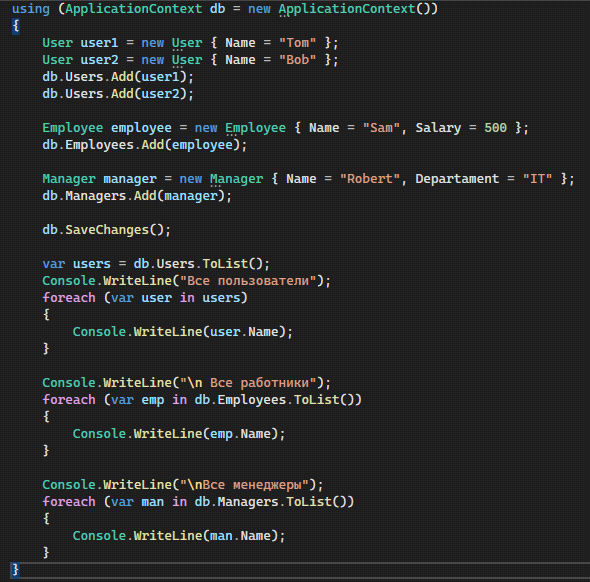


Чтобы включить все классы из иерархии наследования в базу данных, в контексте данных для каждого типа должен быть определен набор DbSet.

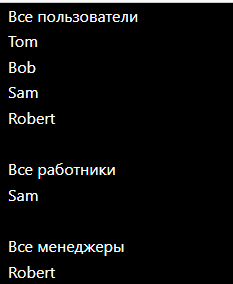
Сгенерированная база данных будет содержать для всех типов одну таблицу Users. Кроме всех свойств классов User, Employee и Manager здесь также появляется еще один столбец - Discriminator. Он имеет тип nvarchar (то есть строка), а в качестве значения он принимает название класса, к которому относится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



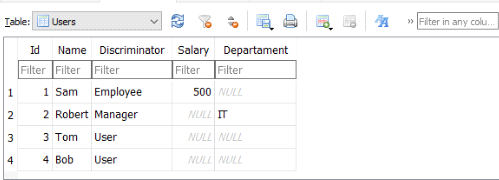
Пример использования:



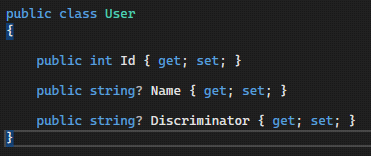
Консольный вывод:



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



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



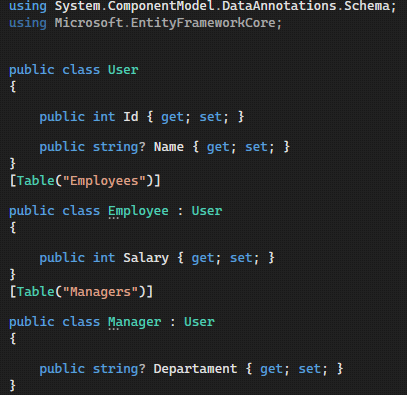
Однако надо учитывать, что на уровне базы данных соответствующий столбец установлен как readonly, то есть только для чтения. Поэтому мы сможем только получать его значения, но не изменять.

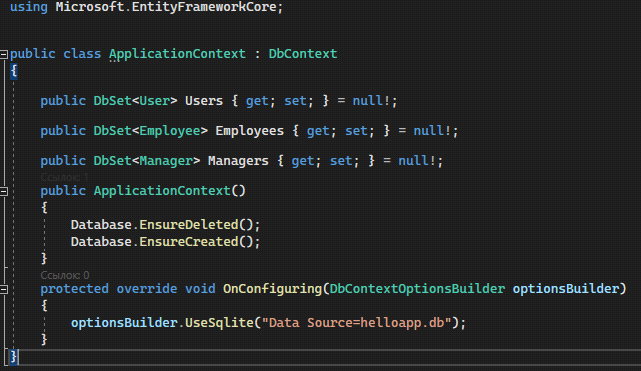
Подход TPT - Table Per Type

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.

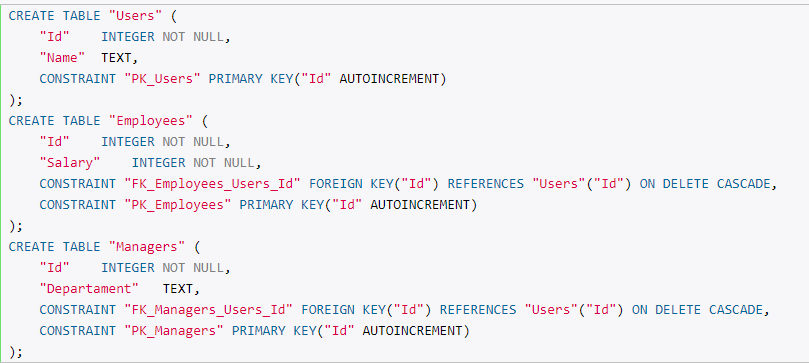
Применение TPT на основе атрибутов

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



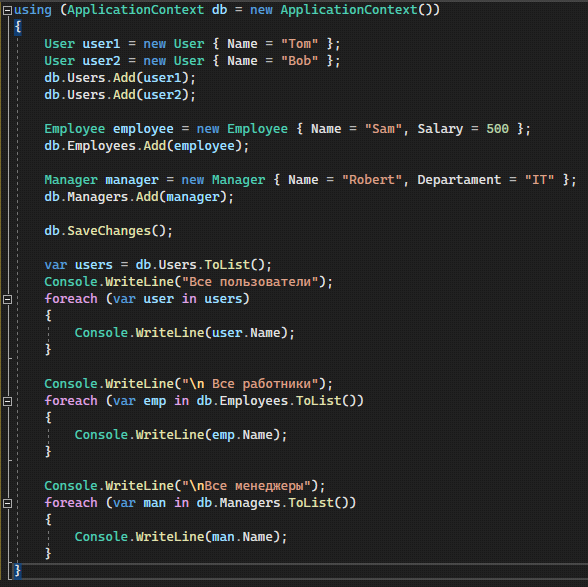


В этом случае в бд будут создаваться следующие три таблицы:

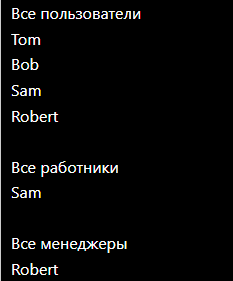


Здесь мы видим, что все свойства базового класса User будут храниться в одной таблице, а те данные, которые относятся только к производным классам, хранятся в отдельных таблицах.

Пример использования:



Консольный вывод:



В остальном применение классов будет аналогично примеру с атрибутом [Table].

Запросы LINQ to Entities

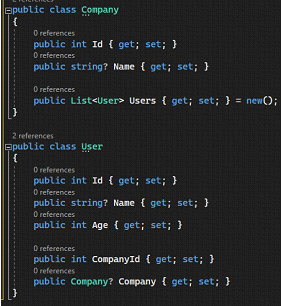
Введение в LINQ to Entities

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.

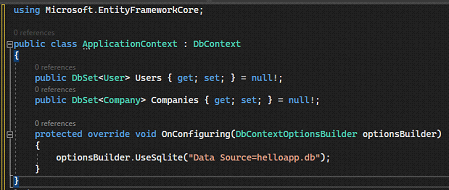
Хотя при работе с базой данных мы оперируем запросами LINQ, но, реляционные базы данных как MS SQL Server или SQLite, понимают только запросы на языке SQL. Поэтому Entity Framework Core, используя выражения LINQ to Entities, транслирует их в определенные запросы, понятные для используемого источника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

Построение запросов

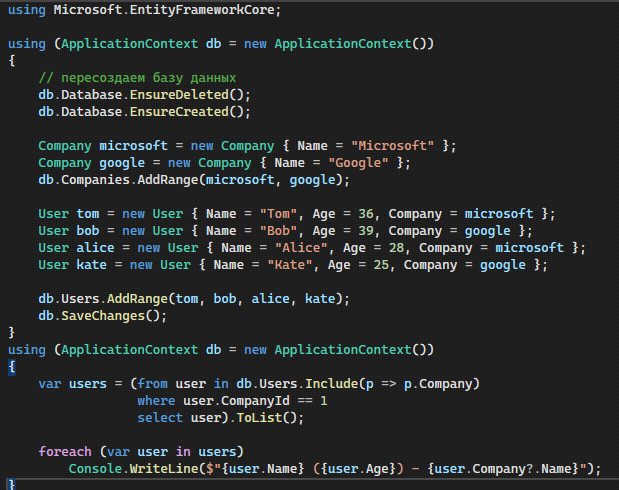
Создавать запросы мы можем двумя способами: через операторы LINQ и через методы расширения. Пусть для рассмотрения основных запросов в LINQ to Entities у нас будут следующие модели со связью один-ко-многим:



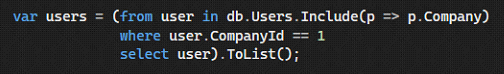
И пусть будет следующий контекст данных:



Вначале используем некоторые операторы LINQ:

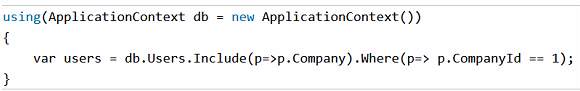


Вначале здесь происходит добавление данных, если они отсутствуют. Далее примяются Linq-операторы:



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.

Теперь для получения данных используем методы расширения LINQ:

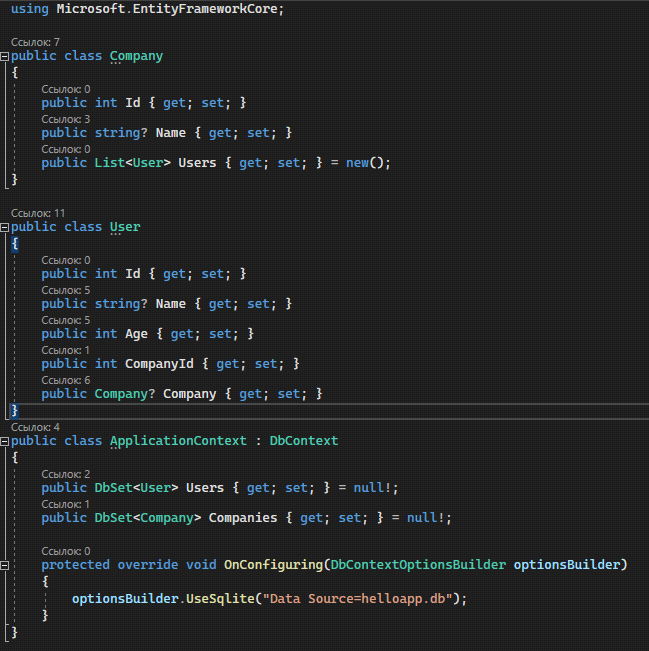


Основные методы, которые мы можем использовать для создания запросов в Entity Framework Core:

* **All / AllAsync**: возвращает true, если все элементы набора удовлятворяют определенному условию, иначе возвращает false
* **Any / AnyAsync**: возвращает true, если хотя бы один элемент набора определенному условию
* **Average / AverageAsync**: подсчитывает cреднее значение числовых значений в наборе
* **Contains / ContainsAsync**: определяет, содержит ли набор определенный элемент
* **Count / CountAsync**: подсчитывает количество элементов в наборе
* **First / FirstAsync**: выбирает первый элемент коллекции
* **FirstOrDefault / FirstOrDefaultAsync**: выбирает первый элемент коллекции или возвращает значение по умолчанию
* **Single / SingleAsync**: выбирает единственный элемент коллекции, если коллекция содердит больше или меньше одного элемента, то генерируется исключение
* **SingleOrDefault / SingleOrDefaultAsync**: выбирает первый элемент коллекции или возвращает значение по умолчанию
* **Select**: определяет проекцию выбранных значений
* **Where**: определяет фильтр выборки
* **OrderBy**: упорядочивает элементы по возрастанию
* **OrderByDescending**: упорядочивает элементы по убыванию
* **ThenBy**: задает дополнительные критерии для упорядочивания элементов возрастанию
* **ThenByDescending**: задает дополнительные критерии для упорядочивания элементов по убыванию
* **Join**: соединяет два набора по определенному признаку
* **GroupBy**: группирует элементы по ключу
* **Except**: возвращает разность двух наборов, то есть те элементы, которые содератся только в одном наборе
* **Union**: объединяет два однородных набора
* **Intersect**: возвращает пересечение двух наборов, то есть те элементы, которые встречаются в обоих наборах элементов
* **Sum / SumAsync**: подсчитывает сумму числовых значений в коллекции
* **Min / MinAsync**: находит минимальное значение
* **Max / MaxAsync**: находит максимальное значение
* **Take**: выбирает определенное количество элементов с начала последовательности
* **TakeLast**: выбирает определенное количество элементов с конца последовательности
* **Skip**: пропускает определенное количество элементов с начала последовательности
* **SkipLast**: пропускает определенное количество элементов с конца последовательности
* **TakeWhile**: возвращает цепочку элементов последовательности, до тех пор, пока условие истинно
* **SkipWhile**: пропускает элементы в последовательности, пока они удовлетворяют заданному условию, и затем возвращает оставшиеся элементы
* **ToList / ToListAsync**: получения списка объектов

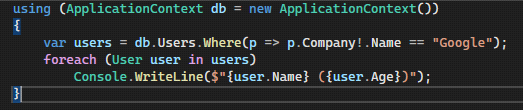
Выборка и фильтрация

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

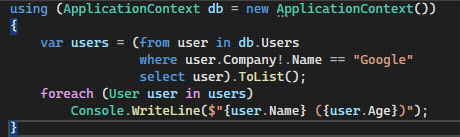


Where

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

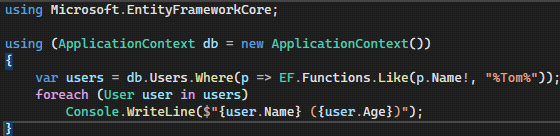


Аналогичный запос с помощью операторов LINQ:



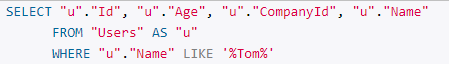
EF.Functions.Like

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"):



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:



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:



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



Cортировка и проекция из базы данных

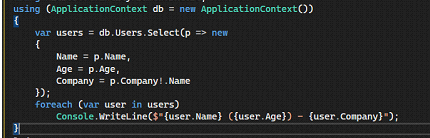
Проекция

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

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



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:



В итоге метод Select из полученных данных спроецирует новый тип.

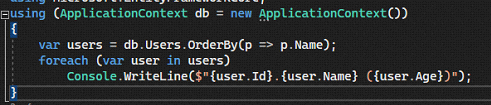
В даном случае мы получим данные анонимного типа, но это также может быть определенный пользователем тип. Например:



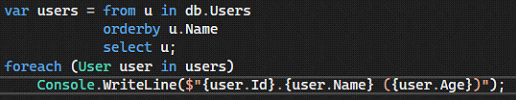
И спроецируем выборку на этот тип:

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:



В качестве альтернативы методу OrderBy можно использовать оператор orderby:



Для сортировки по убыванию применяется метод OrderByDescending():



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



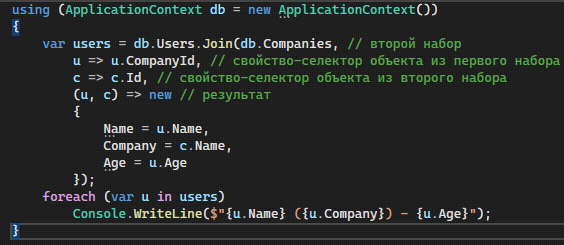
Соединение и группировка таблиц

Соединение таблиц

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



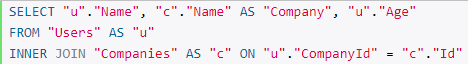
Например, в нашем случае таблица пользователей и таблица компаний имеет общий критерий - id компании, по которому можно провести объединение таблиц:



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:

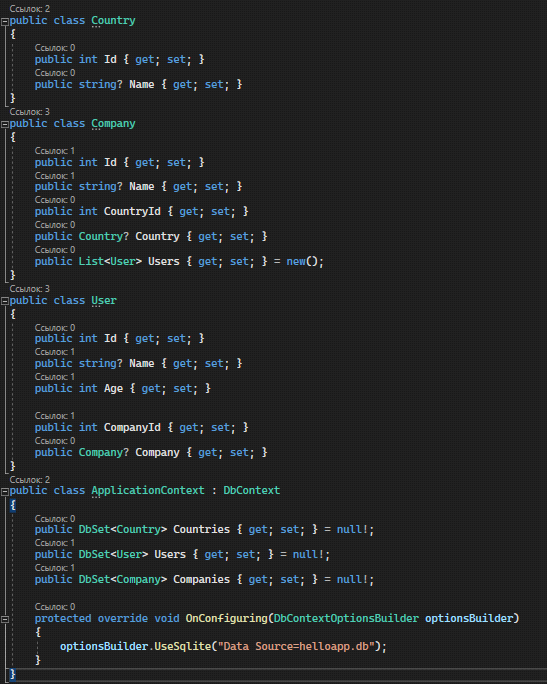


Аналогичного результата мы могли бы достигнуть, если бы использовали оператор join:

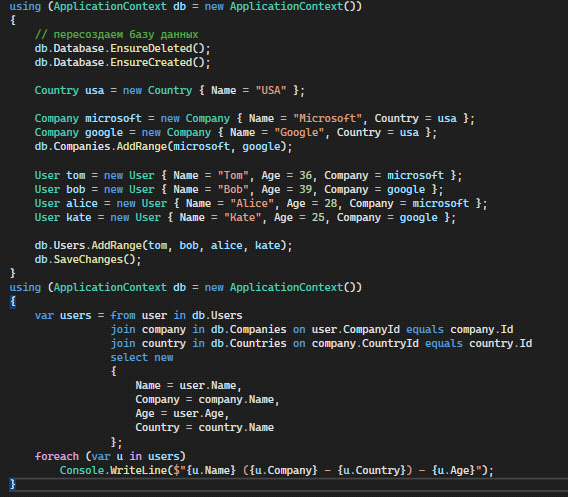


Соединение трех таблиц

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

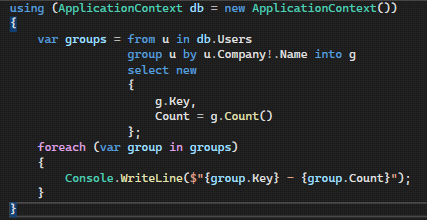


Объединим три таблицы в один набор:



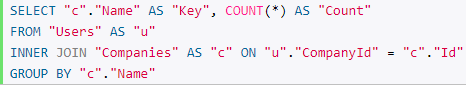
Группировка

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



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.

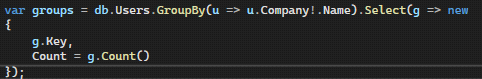
При выполнении запроса будет создаваться следующее SQL-выражение:



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



Аналогично работает метод **GroupBy()**:



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:



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:



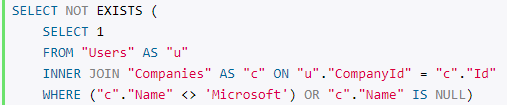
При выполнении это выражение преобразуется в следующий SQL-запрос:



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

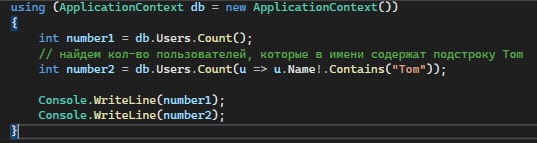


При вызове будет выполняться следующий SQL-запрос:



Количество элементов в выборке

Метод Count() позволяет найти количество элементов в выборке:

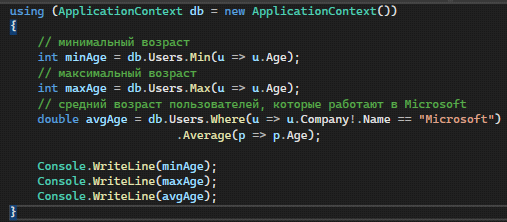


В результате будут выполняться варажения SQL наподобие:



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:

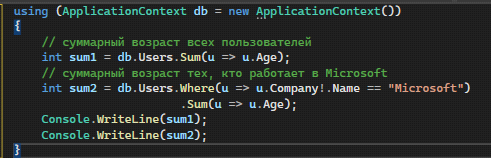


Эти запросы трансформируются в SQLite в следующие SQL-выражения:

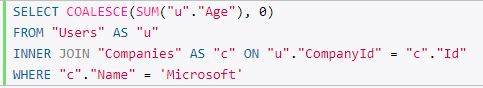


Сумма значений

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



Во втором случае будет выполняться следующее SQL-выражение:

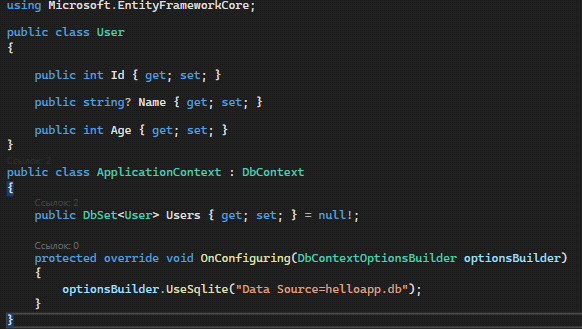


Отслеживание объектов и AsNoTracking

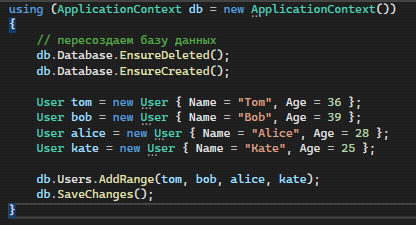
Запросы могут быть отслеживаемыми и неотслеживаемыми. По умолчанию все запросы, которые возвращают объекты классов моделей являются отслеживаемыми. Когда контекст данных извлекает данные из базы данных, Entity Framework Core помещает извлеченные объекты в кэш и отслеживает изменения, которые происходят 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.

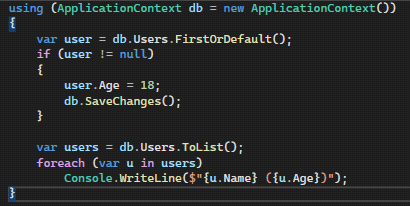
Небольшой пример. Допустим, у нас есть следующие модели и контекст данных:



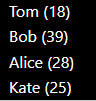
Пусть в базе данных есть есть несколько объектов User:



При обычном выполнении:



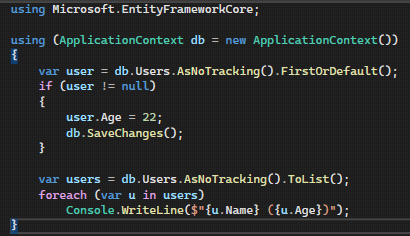
Консольный вывод:



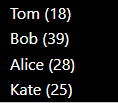
Мы видим, что в наборе users первый элемент имеет у свойства Age значение 18.

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:



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



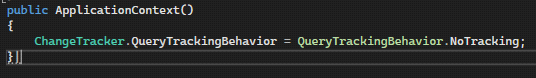
Так как при получении первого элемента используется AsNoTracking, он не будет отслеживаться, и поэтому вызов db.SaveChanges() никак не повлияет на базу данных, а первый элемент сохранит свое первоначальное значение у свойства Age.

Свойство ChangeTracker

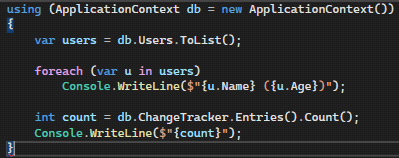
Кроме использования метода AsNoTracking, можно отключить отслеживание в целом для объекта контекста. Для этого надо установить значение **QueryTrackingBehavior.NoTracking** для свойства **db.ChangeTracker.QueryTrackingBehavior**:



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



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



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.

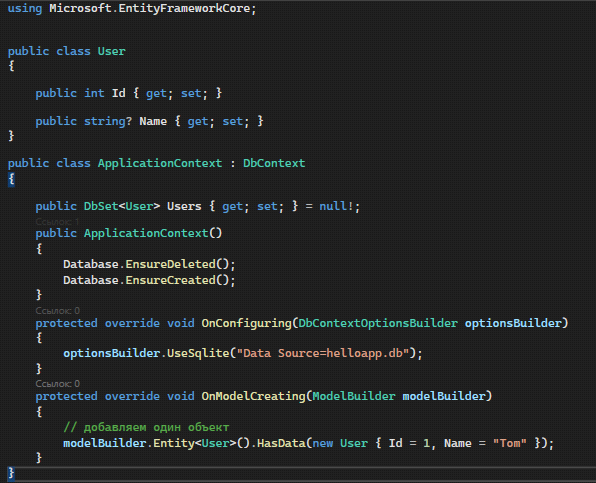
В процессе выполнения запросов LINQ to Entities мы может получать два объекта, которые предоставляют наборы данных: **IEnumerable** и **IQueryable**. С одной стороны, интерфейс IQueryable наследуется от IEnumerable, поэтому по идее объект IQueryable это и есть также объект IEnumerable. Но реальность несколько сложнее. Между объектами этих интерфейсов есть разница в плане функциональности, поэтому они не взаимозаменяемы.

Интерфейс IEnumerable находится в пространстве имен **System.Collections** и **System.Collections.Generic** (обобщенная версия). Объект IEnumerable представляет набор данных в памяти и может перемещаться по этим данным только вперед. Запрос, представленный объектом IEnumerable, выполняется немедленно и полностью, поэтому получение данных приложением происходит быстро.

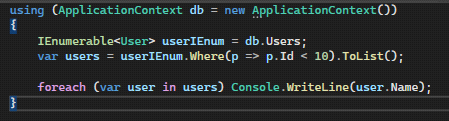
При выполнении запроса IEnumerable загружает все данные, и если нам надо выполнить их фильтрацию, то сама фильтрация происходит на стороне клиента.

Интерфейс IQueryable располагается в пространстве имен **System.Linq**. Объект IQueryable предоставляет удаленный доступ к базе данных и позволяет перемещаться по данным как в прямом порядке от начала до конца, так и в обратном порядке. В процессе создания запроса, возвращаемым объектом которого является IQueryable, происходит оптимизация запроса. В итоге в процессе его выполнения тратится меньше памяти, меньше пропускной способности сети, но в то же время он может обрабатываться чуть медленнее, чем запрос, возвращающий объект IEnumerable.

Для примера используем следующую модель и контекст данных:



Возьмем два вроде бы идентичных выражения. Объект IEnumerable:



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

Здесь запрос будет иметь следующий вид:



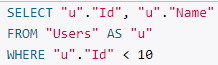
Фильтрация результата, обозначенная с помощью метода Where(p => p.Id > id) будет идти уже после выборки из бд в самом приложении.

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

Объект IQueryable:

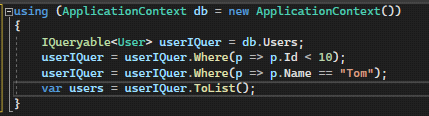


Здесь запрос будет иметь следующий вид:

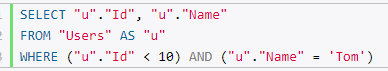


Таким образом, все методы суммируются, запрос оптимизируется, и только потом, когда идет обращеие к методу **ToList()**, происходит выборка из базы данных.

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



В данном случае будет создаваться следующий SQL-запрос:

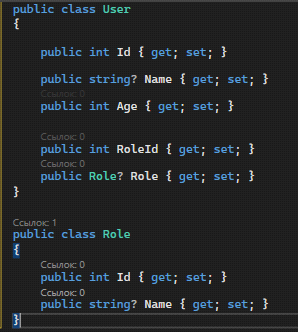


Что же лучше использовать? Все зависит от конкретной ситуации. Если разработчику нужен весь набор возвращаемых данных, то лучше использовать IEnumerable, предоставляющий максимальную скорость. Если же нам не нужен весь набор, а то только некоторые отфильтрованные данные, то лучше применять IQueryable.

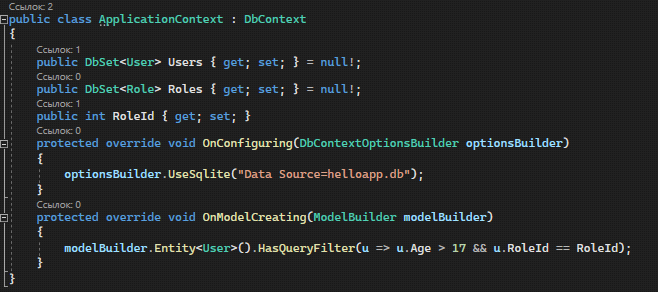
Фильтры запросов уровня модели

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:

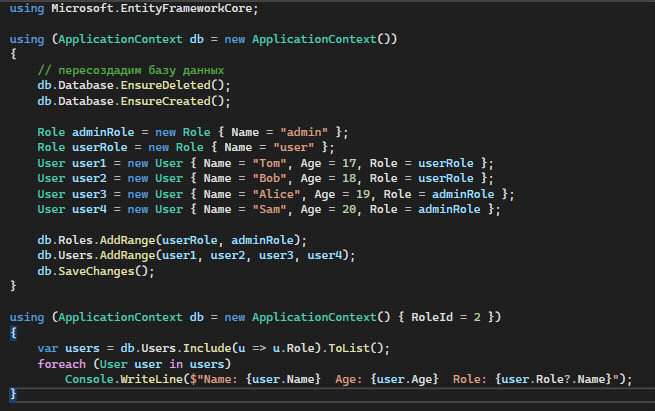


Здесь класс пользователя имеет ссылку на класс роли, к которой он принадлежит. И также определим контекст данных:



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:



Результатом будет следующий консольный вывод:

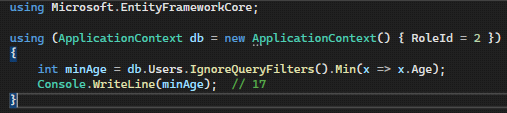


То есть только два объекта из добавленных четырех соответствуют тому предикату, который был передан в HasQueryFilter. И данный фильтр будет действовать для всех запросов к базе данных, которые извлекают данные из таблицы Users. Например, нахождение минимального возраста:



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()**:



SQL в Entity Framework Core

Выполнение SQL-запросов

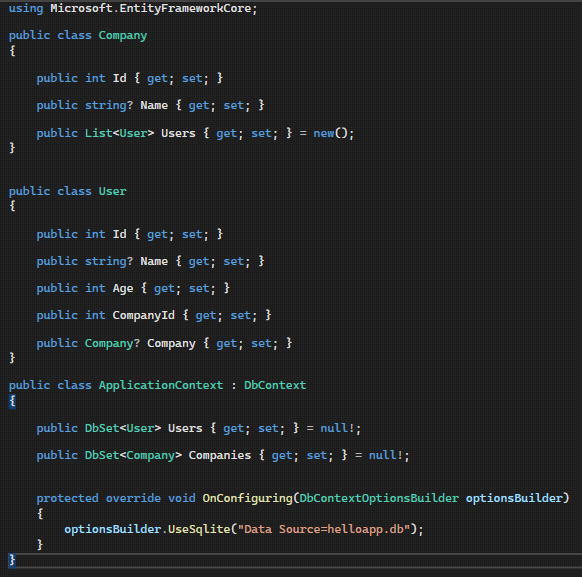
Кроме использования инфраструктуры LINQ to Entities для создания запросов Entity Framework Core также позволяет писать запросы к базе данных сразу на языке SQL. Это может быть удобно, если запрос очень сложный по своей структуре или если Entity Framework Core на основе Linq to Entities создает не очень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

FromSqlRaw

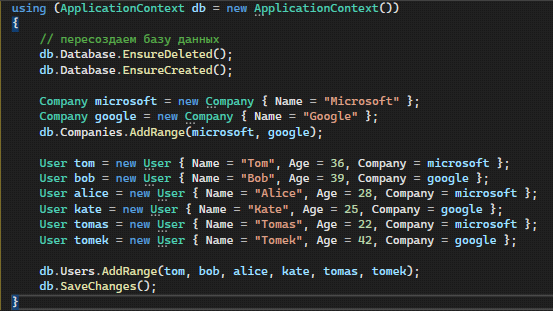
Для получения данных из БД у объектов DbSet определен метод **FromSqlRaw()**, который принимает в качестве параметра sql-выражение и набор параметров. В качестве результата FromSqlRaw возвращает набор полученных из бд объектов.

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.

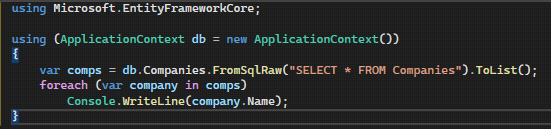
Для примера возьмем следующие модели и контекст данных:



Пусть ранее у нас были добавлены следующие данные:



Например, получим все объекты из таблицы Companies:

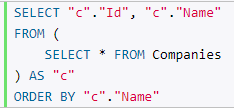


Выражение SELECT извлекает данные из таблицы. Так как эта таблица сопоставляется с моделью Company и хранит объекты этой модели, то в результате мы получим набор объектов класса Company.

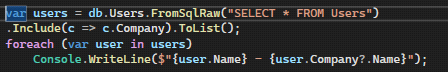
При этом мы можем добавлять к методу другие методы LINQ, которые все вместе будут конкатенироваться в одно общее SQL-выражение:



В итоге будет выполняться следующее SQL-выражение:

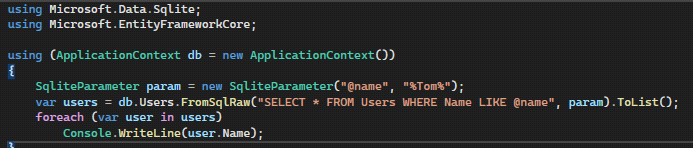


Также можно использовать метод Include для подгрузки связанных данных:



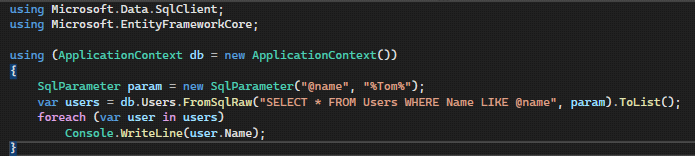
Параметры

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

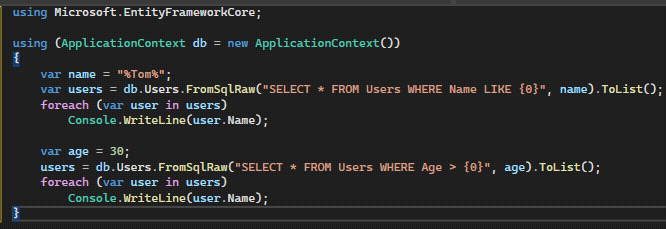


Поскольку выше приведенный контекст данных подключается к бд SQLite, то для представления параметров применяется тип **SqliteParameter** из пространства имен **Microsoft.Data.Sqlite**. Конструктор SqliteParameter имеет ряд версий. В данном случае в конструктор передается два параметра. Первый параметр - 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**:

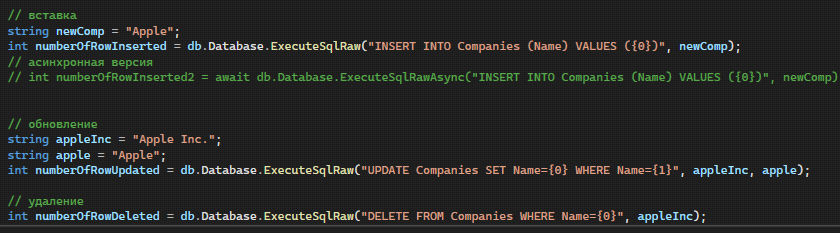


Также мы можем определять параметры как простые переменные:



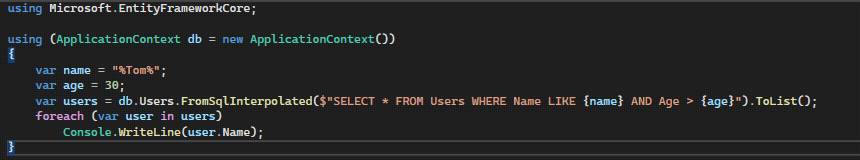
ExecuteSqlRaw

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:

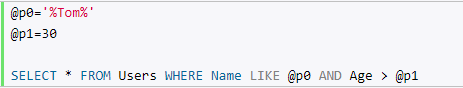


Интерполяция строк

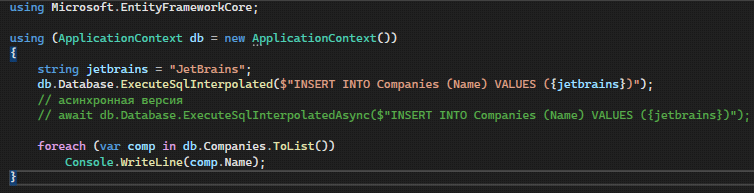
Для методов FromSqlRaw и ExecuteSqlRaw в EF Core определены их двойники - методы **FromSqlInterpolated()** и **ExecuteSqlInterpolated()** (асинхронная версия - **ExecuteSqlInterpolatedAsync()**), которые позволяют использовать интерполяцию строк для передачи параметров. Пример с **FromSqlInterpolated**:



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



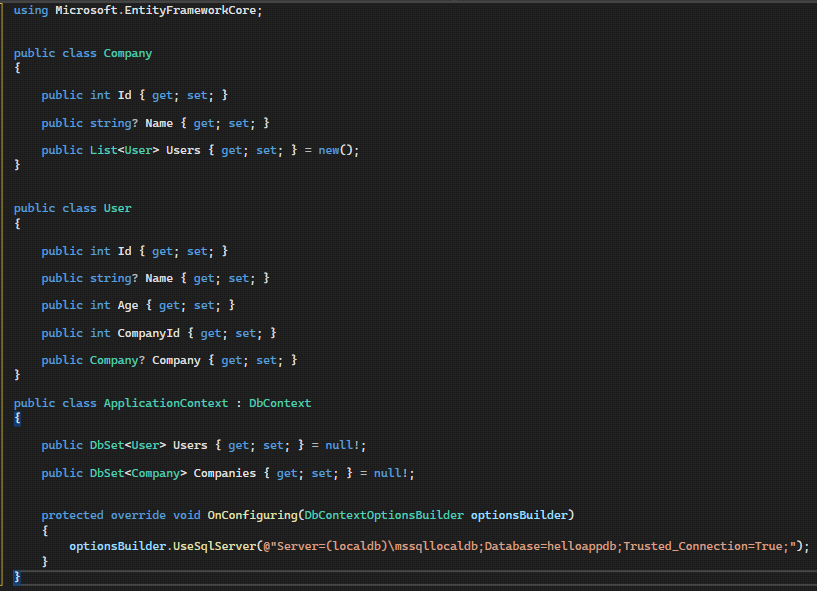
Использование **ExecuteSqlInterpolated()/ExecuteSqlInterpolatedAsync()**:



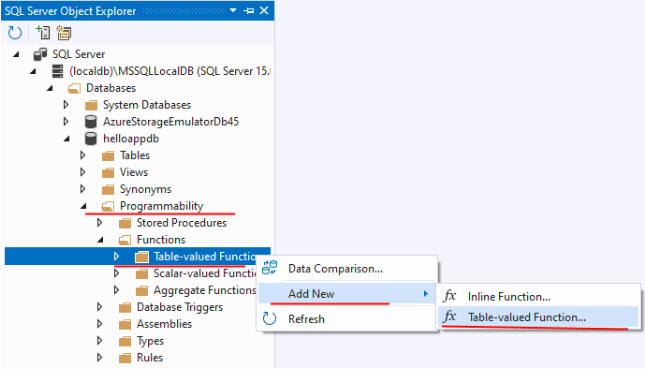
Хранимые функции

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:

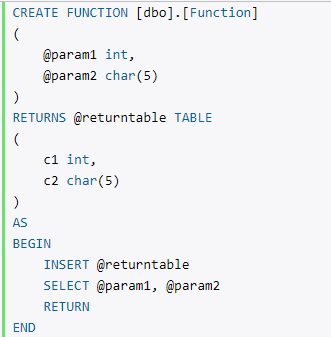


Вначале создадим саму хранимую функцию, к которой мы будем обращаться. Для этого в Visual Studio в окне **SQL Server Object Explorer** откроем узел базы данных. В узле базы данных откроем подузел **Programmability -> Functions -> Table-valued Functions**:

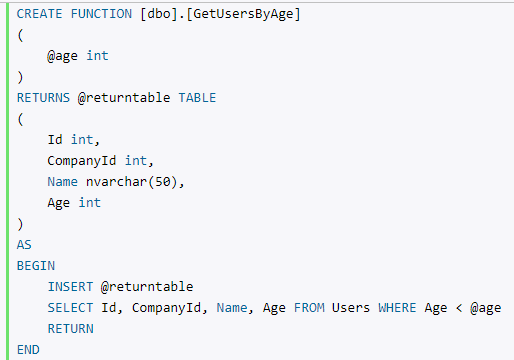


Далее нажмем на этот узел правой кнопкой мыши и выберем **Add New -> Table-valued Function**:

После этого Visual Studio генерирует и открывает файл скрипта со следующим содержимым:



Изменим скрипт следующим образом:

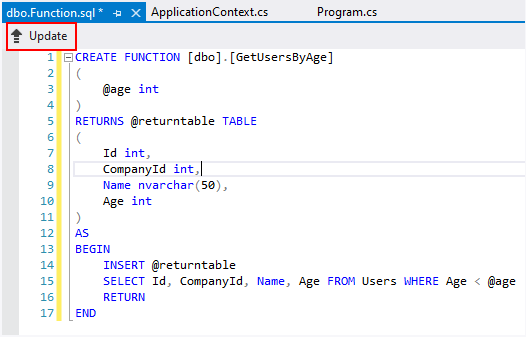


Выражение CREATE FUNCTION [dbo].[GetUsersByAge] создае новую функцию GetUsersByAge. После ее названия определяется список параметров. Пусть наша функция принимает только один параметр @age, который имеет тип int, то есть целочисленное значение.

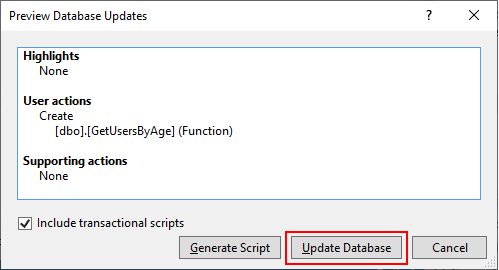
Затем идет определение возвращаемого объекта-таблицы в выражении RETURNS @returntable TABLE(...). В скобках идет перечисление столбцов возвращаемой таблицы. В данном случае они совпадают с определением таблицы Users. То есть таблица будет содержать объекты класса User.

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.

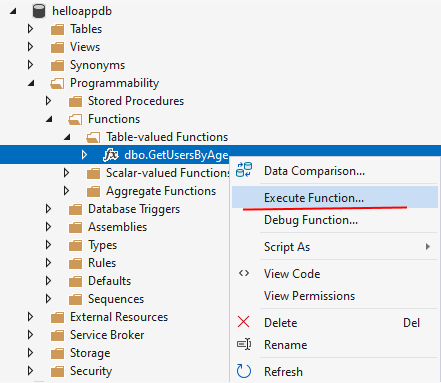
Теперь добавим функцию в базу данных. Для этого нажмем на кнопку Update:



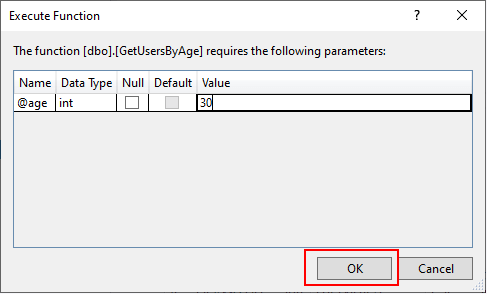
Затем в появившемся окне нажмем на кнопку **Update Database**



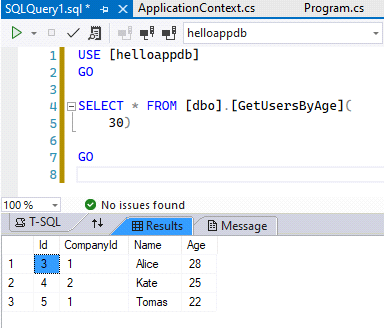
После этого в базу данных будет добавлена функция, и ее можно будет использовать. Но перед обращением к ней из кода c# мы ее протестируем, чтобы убедиться, что она работает как надо. Для этого нажмем на функцию правой кнопкой мыши и в появившемся меню выберем пункт **Execute Function**:



После этого откроется окно для установки входных параметров функции. Введем в поле Value какое-нибудь число, которое будет передаваться в функцию в качестве параметра:



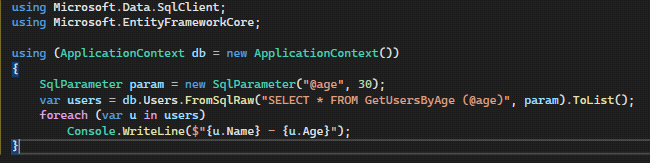
И Visual Studio сгенерирует и сразу же выполнит скрипт с функцией и переданным в нее параметром:



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#:



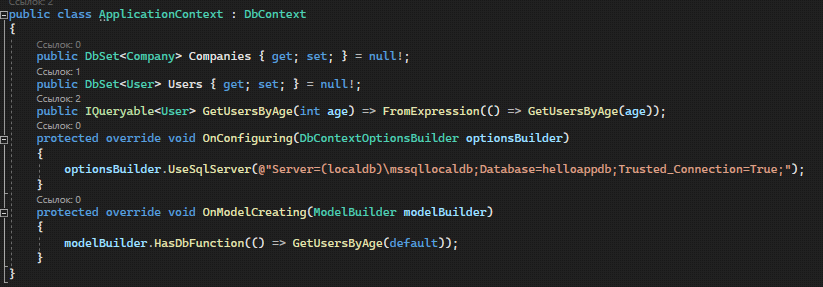
В данном случае в запросе вместо таблицы указываем имя вызов функции с переданными ей параметрами: GetUsersByAge (@age)

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:



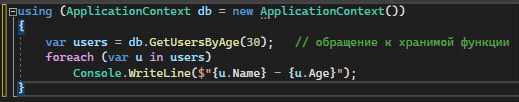
Здесь добавлен метод GetUsersByAge(), который соответствует хранимой функции в БД. Он принимает параметр типа int и возвращает объект IQueryable<User>. Этот метод с помощью встроенного в классе DbContext метода FromExpression вызывает GetUsersByAge(age).



Далее в переопрепределенном методе OnModelCreating() класса контекста нам надо зарегистрировать метод GetUsersByAge с помощью вызова **HasDbFunction()**:



Далее мы можем обратиться к хранимой функции GetUsersByAge в коде следующим образом:



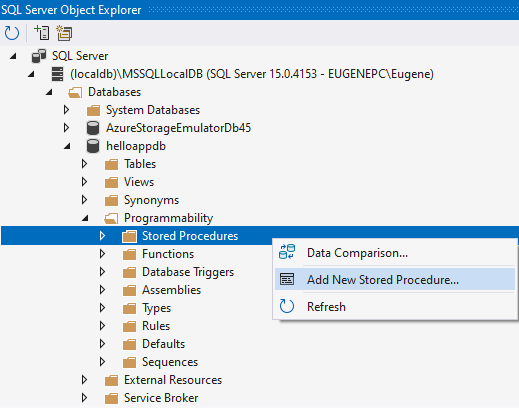
Хранимые процедуры

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:

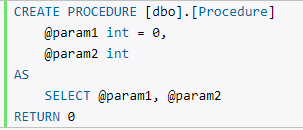


Вначале создадим саму хранимую процедуру, к которой мы будем обращаться. Для этого в Visual Studio в окне **SQL Server Object Explorer** откроем узел базы данных. В узле базы данных откроем подузел Programmability -> Stored Procedures:

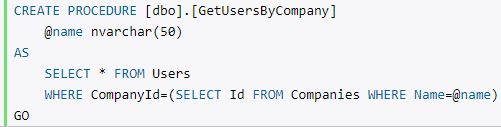


Далее нажмем на этот узел правой кнопкой мыши и выберем **Add New Stored Procedure**:

После этого Visual Studio генерирует и октрывает файл скрипта хранимой процедуры со следующим содержимым:

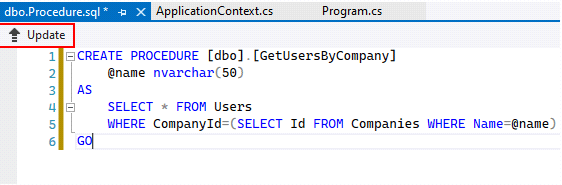


Изменим скрипт следующим образом:

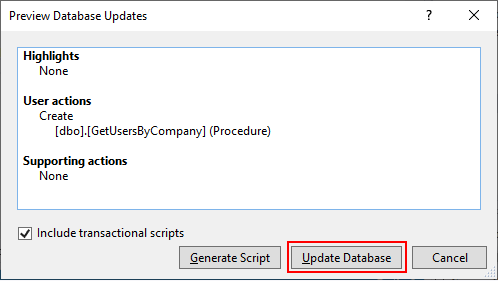


Данная процедура ищет все строки, где значение столбца название компании равно строке, переданной через параметр @name.

Далее нажмем на кнопку Update для добавления хранимой процедуры:

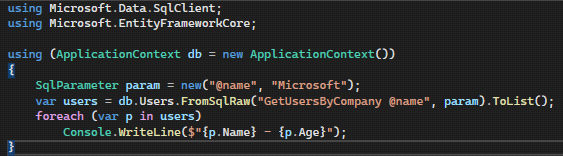


И затем в появившемся окошке нажимаем кнопку Update Database:



После этого в узле **Stored Procedures** появится новая хранимая процедура, и мы сможем ее использовать.

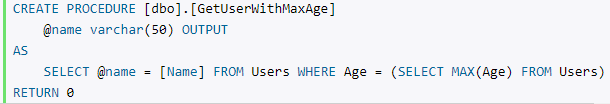
Теперь обратимся к процедуре в коде C#:



Параметр в методе FromSqlRaw принимает название процедуры, после которого идет перечисление параметров: GetUsersByCompany @name

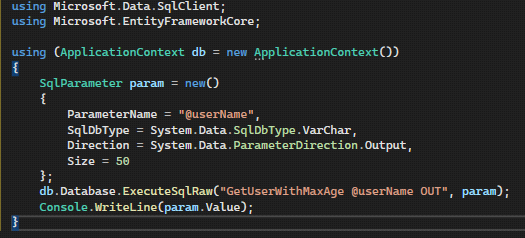
Выходные параметры процедуры

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:



Параметр name здесь определен как выходной с ключевым словом **OUTPUT**. Через него будет передаваться имя пользователя.

Обратимся к этой процедуре в коде C#:



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.