# Глава 1

# Что такое Entity Framework Core

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

# 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("Объекты успешно сохранены");

// получаем объекты из бд и выводим на консоль

var users = db.Users.ToList();

Console.WriteLine("Список объектов:");

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);

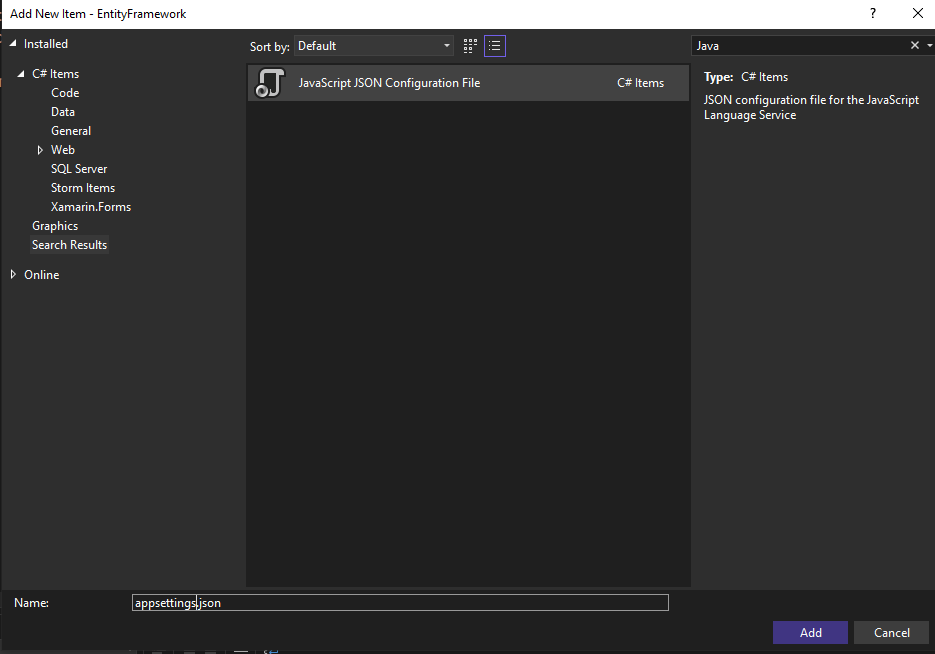
db.SaveChanges();

}

## 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": {

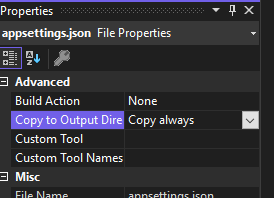
"DefaultConnection": "Data Source=helloapp.db"

}

}

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:");

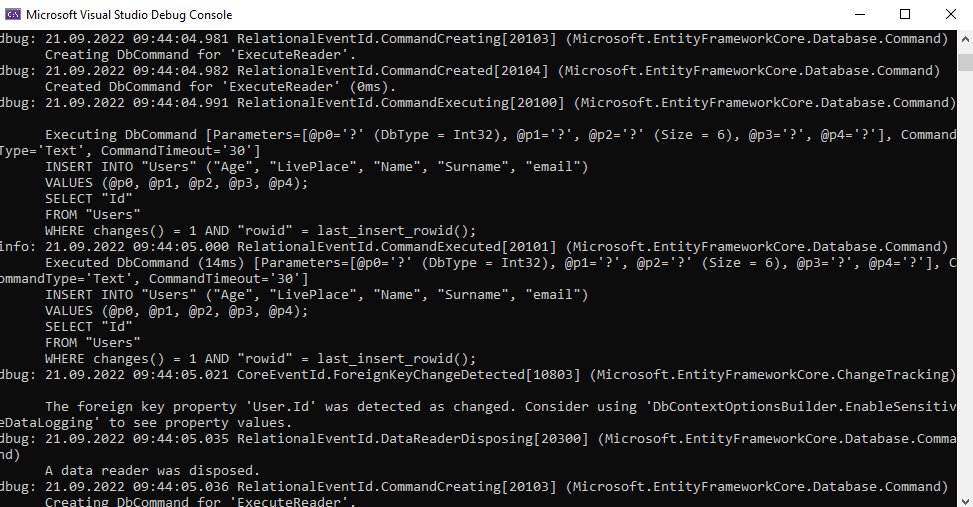
foreach (User u in users)

{

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()

{

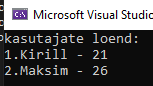
await base.DisposeAsync();

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)

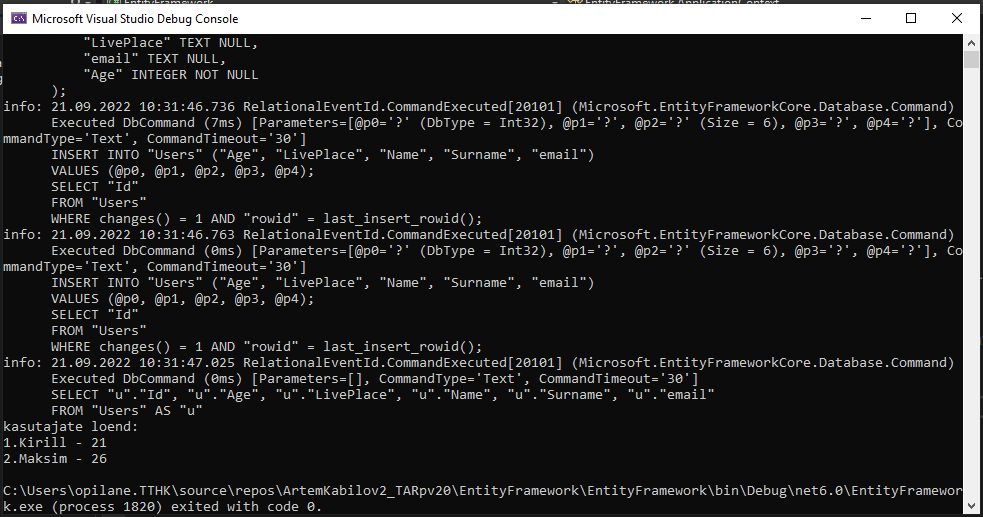
{

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();

}

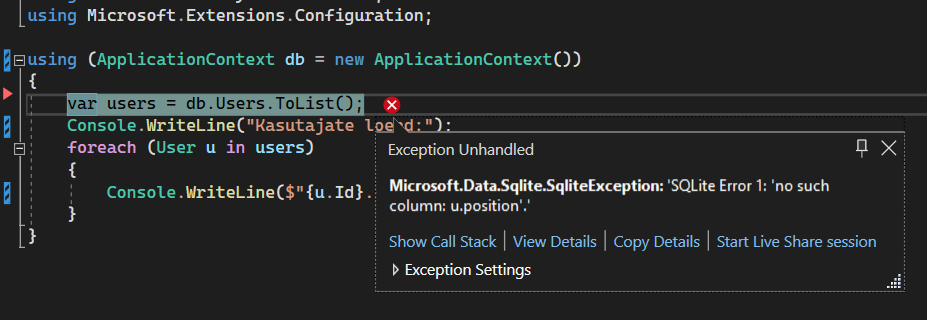
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

}

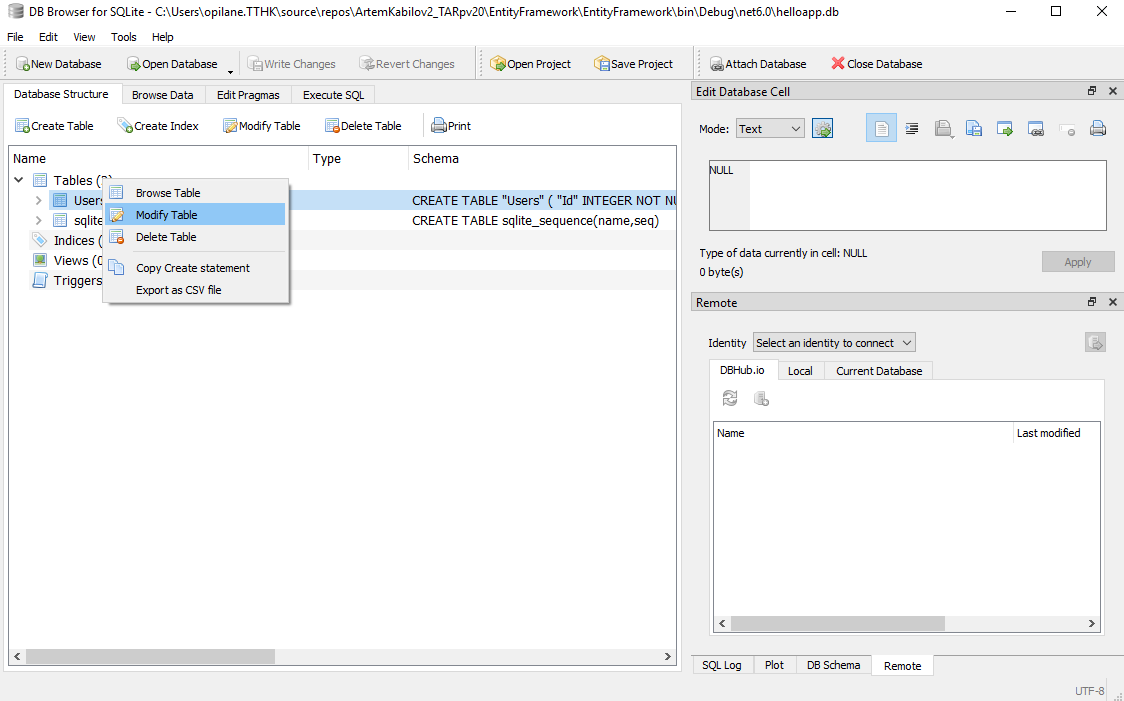
}



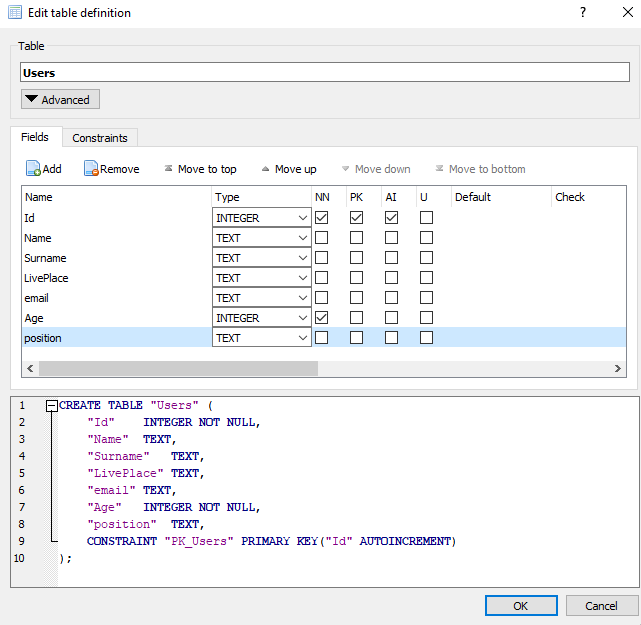
## Ручное изменение базы данных

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 for SQLite](https://sqlitebrowser.org/). Так, откроем базу данных в этой программе. Нажмем на таблицу Users правой кнопкой мыши и в появившемся контекстном меню выберем Modify Table:



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



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

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

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

## Database.EnsureCreated и Database.EnsureDeleted

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

public class ApplicationContext : DbContext

{

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

public ApplicationContext()

{

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

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

}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

}

}

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

## Миграция

Миграция по сути предствляет план перехода базы данных от старой схемы к новой.

Для создания миграции в окне Package Manager Console вводится команда

Add-Migration название миграции

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

Если планируется использовать миграции, то лучше их использовать сразу при создании базы данных. Для использования миграций в Visual Stuido необходимо добавить в проект через менеджер Nuget пакет **Microsoft.EntityFrameworkCore.Tools**.

public class ApplicationContext : DbContext

{

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

public ApplicationContext()

{

// Database.EnsureCreated();

}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

}

}

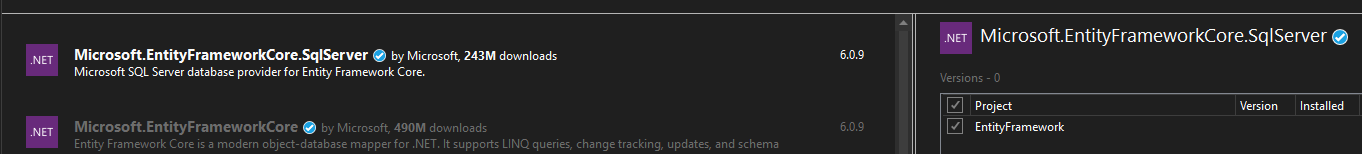
Обратите внимание, что в конструкторе контекста закомментирован метод Database.EnsureCreated(). В данном случае он не нужен. Более того при выполнении миграции этот метод вызывает ошибку. Этот момент следует учитывать.

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

# Глава 2

# MS SQL Server

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



## Мнеяем ApplicationContext

public class ApplicationContext : DbContext

{

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

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

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

}

}

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

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

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

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

using (ApplicationContext db = new ApplicationContext())

{

db.Database.EnsureDeleted();

db.Database.EnsureCreated();

// создаем два объекта User

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

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

// добавляем их в бд

db.Users.AddRange(user1, user2);

db.SaveChanges();

}

// получение данных

using (ApplicationContext db = new ApplicationContext())

{

// получаем объекты из бд и выводим на консоль

var users = db.Users.ToList();

Console.WriteLine("Users list:");

foreach (User u in users)

{

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

}}

