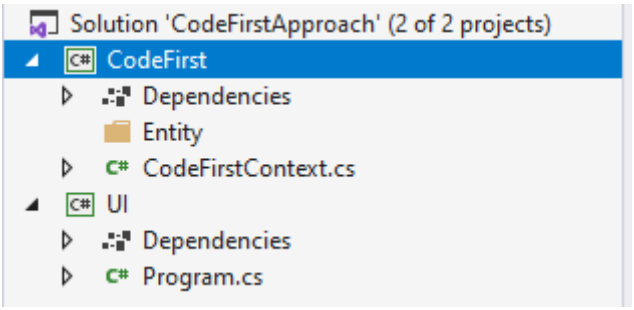
Упражнение: Комуникация с база данни (Code First)

Какво ще направим?

Ще създадем приложение, което може да комуникира с база от данни, като я създава използвайки Code First метод, т.е. ще използваме Entity Framework Core, за да генерираме таблици в база от данни по тяхно описание чрез класове.

Архитектура на проекта

Проектът, ще се състои от един слой – библиотека, която ще служи за мост между приложението и базата от данни. Моделите ще поставяме в папка Entity под проекта CodeFirst.



Нужни пакети

За целта на проекта ще са ни нужни следните пакети, които можете да намерите в NuGet мениджъра (десен бутон върху Dependencies -> Manage NuGet packages...):

- **Microsoft.EntityFrameworkCore**
- **Microsoft.EntityFrameworkCore.SqlServer (защото ще използваме MS SQL Server за база от данни)**
- **Microsoft.EntityFrameworkCore.Tools**
- **Microsoft.EntityFrameworkCore.Design**
- **Microsoft.EntityFrameworkCore.Proxies**

Последните два пакета са ни нужни за добавяне на миграции и генериране на таблиците следвайки описания с класове модел.

Слой за комуникация с БД

Ще започнем с изграждането на слоя за комуникация с БД. За да използваме Code First метода е нужно да опишем таблиците си с класове. За по-добра структура на проекта ще групираме тези класове под папката Entity. Тези класове трябва да се използват в контекст (чрез контекста се установява самата връзка с БД, както и операциите по извличане, изтриване, обновяване, добавяне). Нека започнем с нашия контекст.

Контекстът трябва да е клас, който задължително наследява DbContext. В него като DbSet<T> ще добавим моделите си. От там EF Core ще ги вземе и създаде съответните таблици. Това е класът, в който ще конфигурираме връзките между таблиците си.

```
namespace CodeFirst
{
    0 references
    public class CodeFirstContext : DbContext
    {
        //TODO: ...
    }
}
```

Нека за сега оставим контекста празен. Ще се върнем обратно на него, когато създадем нашите модели.

Ще преминем през създаването на основните типове връзки между две таблици – един-към-един, един-към-много, много-към-много.

Връзка един-към-един:

Нека започнем с един-към-един. За целта ще създадем две таблици – автори и биографии. Съответно всеки автор има само една биография и всяка биография се отнася само за един автор.

Авторът ще има следните свойства: първо име, фамилия, биография, а биографията ще съдържа самата биография (текст), дата на раждане, място на раждане, националност, съответния автор. Всеки модел трябва да има и основен ключ (primary key), който ще отделим в базов клас.

В папката Entity ще създадем трите класа – BaseEntity, Author и Biography.

Базовият клас BaseEntity:

```
namespace CodeFirst.Entity
{
    2 references
    public abstract class BaseEntity
    {
        0 references
        public int Id { get; set; }
    }
}
```

EF Core е достатъчно умен, за да разбере по именуването на свойството (Id), че за него трябва да бъде създадена съответна колона, която да служи като основен ключ. Освен това колоната ще бъде конфигурирана и за автоматично инкрементиране.

Класът Author:

```
1 reference
public class Author : BaseEntity
{
    0 references
    public string FirstName { get; set; }

    0 references
    public string LastName { get; set; }

    0 references
    public int BiographyId { get; set; }

    0 references
    public virtual Biography Biography { get; set; }
}
```

За EF Core е достатъчно да се добави свойство от тип, който също описва таблица в базата (в случая Biography), за да разбере типа на връзка между двете таблици. Подобно на Id, тук EF Core ще проследи, че BiographyId съвпада със свойството Biography, но завършва на "Id" и ще го използва за външен ключ към таблицата. Biography е виртуално свойство, за да се включи опцията за lazy-loading на EF Core. Най-просто обяснено, това означава, че при зареждане на даден запис от БД няма да се заредят излишни данни преди тяхното поискване. За да конфигурираме EF Core да използва lazy-loading е нужно да пренапишем метода OnConfiguring в нашият контекст-клас. Използвайки параметъра optionsBuilder, можем да достъпим различни опции за конфигурация на контекста.

```
0 references
public class CodeFirstContext : DbContext
{
    0 references
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseLazyLoadingProxies();
    }
}
```

Нека се върнем към нашият клас Author. За всяко свойство в класа EF ще създаде колона в таблицата за нашите автори. Типа на колоната ще бъде съответстващ на типа, който използваме в класа си.

Класът Biography:

```
1 reference
public class Biography : BaseEntity
{
    0 references
    public string BiographyData { get; set; }

    0 references
    public DateTime DateOfBirth { get; set; }

    0 references
    public string PlaceOfBirth { get; set; }

    0 references
    public string Nationality { get; set; }

    0 references
    public int AuthorId { get; set; }

    0 references
    public virtual Author Author { get; set; }
}
```

Аналогично има връзка към Author. Това, че сме добавили и в двата класа връзка към другия е достатъчно за EF Core, но ние ще използваме OnModelCreating() метода на контекста, за да определим изрично типа на връзката.

Забележка: За nullable типовете се създава съответно незадължителна колона в таблицата, докато при типовете като DateTime винаги ще създадат задължителна колона. Ако искаме да промени това трябва да направим свойството си от тип DateTime? (нулиращ се DateTime).

Добавяне на класовете в контекста:

```
0 references
public class CodeFirstContext : DbContext
{
    0 references
    public virtual DbSet<Author> Authors { get; set; }

    0 references
    public virtual DbSet<Biography> Biographies { get; set; }

    0 references
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)...
}
```

Класовете, за които искаме да бъде създадена таблица в БД се добавят като свойства на контекст класа ни. Таблиците ще носят имената на свойствата.

Конфигуриране на връзките между таблиците:

За изричната конфигурация на връзките между таблиците, както и задаването на ограничения за различните колони се пренаписва OnModelCreating() метода на контекста. В него конфигурациите стават чрез методи с много себеописателни имена.

```
0 references
public class CodeFirstContext : DbContext
{
    0 references
    public virtual DbSet<Author> Authors { get; set; }

    0 references
    public virtual DbSet<Biography> Biographies { get; set; }

    0 references
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Author>()
            .HasOne( navigationExpression: a => a.Biography)
            .WithOne( navigationExpression: b => b.Author)
            .HasForeignKey<Biography>(b => b.AuthorId);
    }

    0 references
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)...
}
```

Няма значение дали конфигурирането става от страна на автора или от страна на биографията.

С това завършва имплементацията на един-към-един връзката.

Връзка един-към-много:

За целта ще създадем две таблици – компании и служители. Съответно една компания може да има много служители, а всеки служител работи само за една компания.

Компанията ще има следните свойства: име, списък със служители. Служителят, от друга страна, ще има следните свойства: име, компания, в която работи. Отново ще наследяват базовия клас BaseEntity.

Класът Company:

```
1 reference
public class Company : BaseEntity
{
    0 references
    public string Name { get; set; }

    0 references
    public virtual ICollection<Employee> Employees { get; set; }
}
```

Класът Employee:

```
1 reference
public class Employee : BaseEntity
{
    0 references
    public string Name { get; set; }

    0 references
    public int CompanyId { get; set; }

    0 references
    public virtual Company Company { get; set; }
}
```

Отново добавяме класовете като DbSet<T> в нашия контекст:

```
0 references
public class CodeFirstContext : DbContext
{
    0 references
    public virtual DbSet<Author> Authors { get; set; }

    0 references
    public virtual DbSet<Biography> Biographies { get; set; }

    0 references
    public virtual DbSet<Company> Companies { get; set; }

    0 references
    public virtual DbSet<Employee> Employees { get; set; }

    0 references
    protected override void OnModelCreating(ModelBuilder modelBuilder)...

    0 references
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)...
}
```

Отново на този етап EF Core може сам да определи връзката между двете таблици, но случаят не винаги е такъв, затова ще използваме FluentAPI-то (ModeBuilder), за да конфигурираме връзката. И тук няма значение от коя страна ще започне конфигурацията.

```
0 references
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Author>()
        .HasOne( navigationExpression: a => a.Biography)
        .WithOne( navigationExpression: b => b.Author)
        .HasForeignKey<Biography>(b => b.AuthorId);

    modelBuilder.Entity<Company>()
        .HasMany( navigationExpression: c => c.Employees)
        .WithOne( navigationExpression: e => e.Company)
        .HasForeignKey(e => e.CompanyId);
}
```

Към този момент нашата един-към-много връзка е имплементирана.

Връзка много-към-много:

За целта ще създадем класове за книги и за категории, като всяка книга може да попадне в много категории и всяка категория може да има много книги под нея. В случая обаче ще създадем и трети клас, който ще е връзката между книгите и категориите.

Книгата ще има следните свойства: заглавие, автор, списък с категории, а категорията ще съдържа име и списък с книги. Отново двата модела ще наследяват базовия клас BaseEntity.

Класът Book:

```
2 references
public class Book : BaseEntity
{
    0 references
    public string Title { get; set; }

    0 references
    public int AuthorId { get; set; }

    0 references
    public virtual Author Author { get; set; }

    1 reference
    public virtual ICollection<BookCategory> BookCategories { get; set; }
}
```

Класът Category:

```
2 references
public class Category : BaseEntity
{
    0 references
    public string Name { get; set; }

    1 reference
    public virtual ICollection<BookCategory> BookCategories { get; set; }
}
```

Забележка: Вместо връзка един към друг двата класа пазят връзка към помощния клас BookCategory

Класът, правещ връзка между категориите и книгите – BookCategory:

Забележка: Не наследява базовият клас BaseEntity, тъй като в този случай основният ключ ще е комбинация от две колони.

```
0 references
public class BookCategory
{
    0 references
    public int BookId { get; set; }
    0 references
    public virtual Book Book { get; set; }
    0 references
    public int CategoryId { get; set; }
    0 references
    public virtual Category Category { get; set; }
}
```

Отново добавяме класовете като DbSet<T> в контекста ни, като в този случай добавяме и помощния клас.

```
0 references
public virtual DbSet<Employee> Employees { get; set; }

0 references
public virtual DbSet<Book> Books { get; set; }

0 references
public virtual DbSet<Category> Categories { get; set; }

0 references
public virtual DbSet<BookCategory> BookCategories { get; set; }

0 references
protected override void OnModelCreating(ModelBuilder modelBuilder)...
```

За да се осъществи много-към-много връзката, тя трябва задължително да бъде изрично конфигурирана през FluentApi-то, както следва:

```
modelBuilder.Entity<BookCategory>()
    .HasKey(bc => new { bc.BookId, bc.CategoryId });

modelBuilder.Entity<BookCategory>()
    .HasOne( navigationExpression: bc => bc.Book)
    .WithMany( navigationExpression: b => b.BookCategories)
    .HasForeignKey(bc => bc.BookId);

modelBuilder.Entity<BookCategory>()
    .HasOne( navigationExpression: bc => bc.Category)
    .WithMany( navigationExpression: c => c.BookCategories)
    .HasForeignKey(bc => bc.CategoryId);
```

, където първо конфигурираме основния ключ на помощната таблица – уникална комбинация от две колони (2 външни ключа).

С това завършва имплементацията на нашата много-към-много връзка.

Стринг за свързване (connection string):

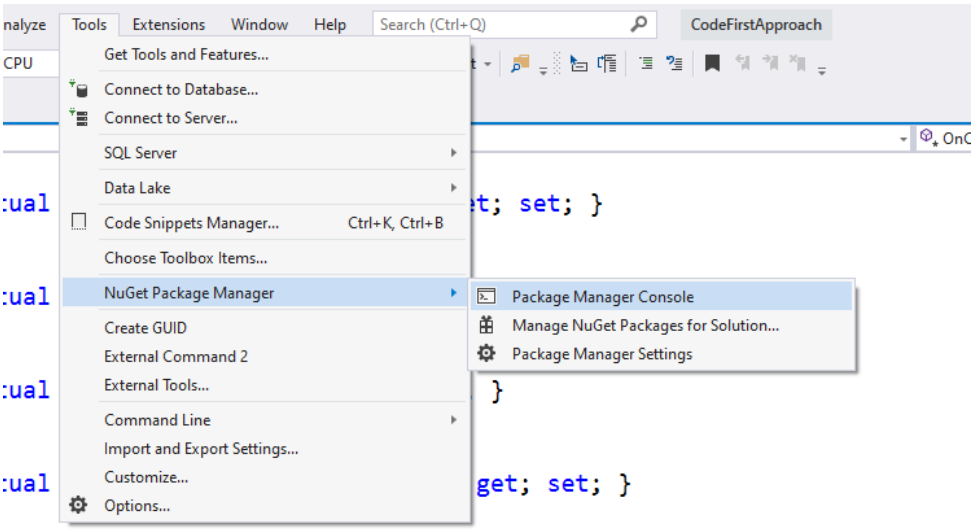
Ще конфигурираме EF Core да използва MS SQL Server, като ще му подадем нужната информация за свързването със сървъра във формат на стринг. Това става отново през пренаписания метод OnConfiguring.

```
0 references
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    optionsBuilder.UseSqlServer( connectionString: @"Data Source=(localdb)\ProjectsV13;Initial Catalog=CodeFirstDB;");
    optionsBuilder.UseLazyLoadingProxies();
}
```

Използваме метода UseSqlServer(), като му подадем параметър – стринга за свързване.

Създаване на базата:

Остава ни само да създадем БД по описаните от нас модели. Вече имаме инсталирани нужните за това пакети. Ще започнем като добавим нова миграция, която ще кръстим "Initial", понеже е първоначална за базата ни. За целта ще използваме команди в Package Manager конзолата, която се отваря по следния начин:



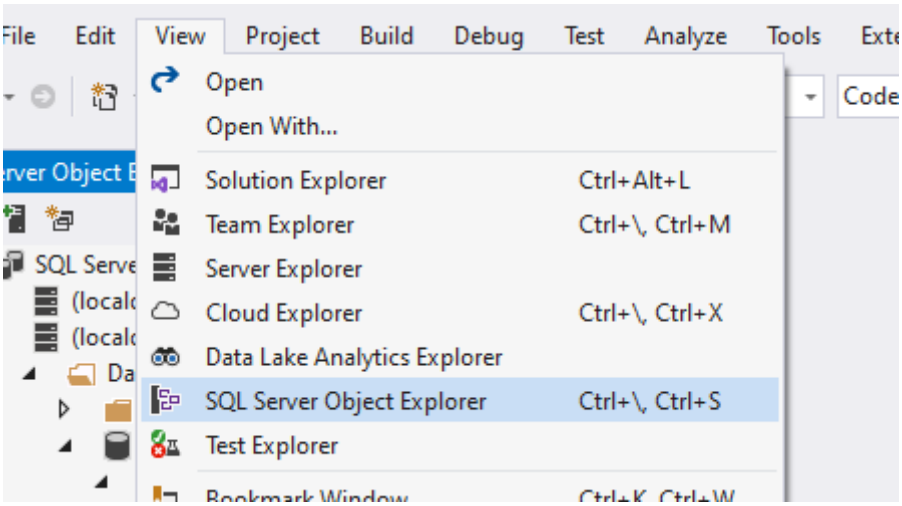
За създаване на първата ни миграция ще използваме командата Add-Migration <име на миграцията>:

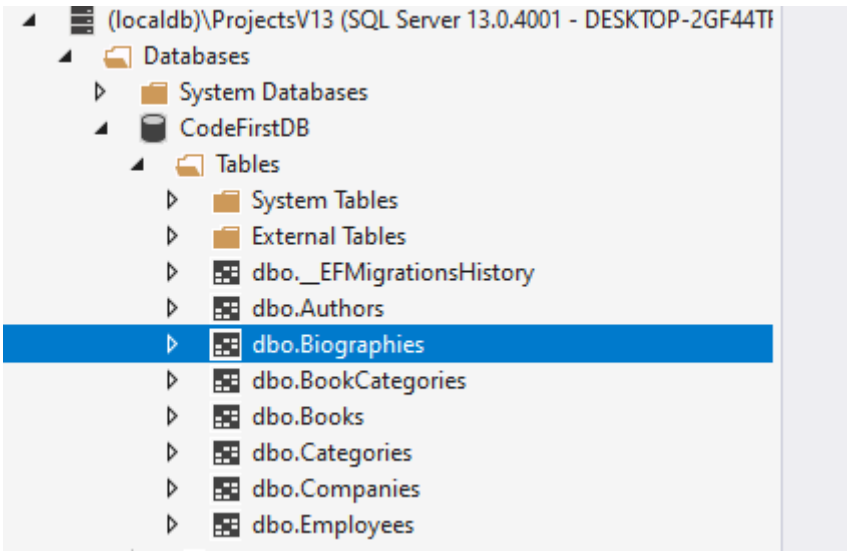
```
PM> Add-Migration Initial
To undo this action, use Remove-Migration.
PM>
```

Забележка: Командата ще създаде таблица Migrations, която съдържа класове описващи миграцията и позволяващи ни специфични конфигурации, върху които няма да се спрем сега.

Ще използваме и командата Update-Database, която преди да приложи промените от миграцията ще създаде нашата база. Ако всичко е наред ще се изпише заявката по създаване на таблицата, както и съобщение "Done."

За да валидираме, че всичко в базата ни от данни е създадено по точен начин можем да използваме вградения във Visual Studio SQL Server Object Explorer:





След като открием базата си и установим, че всичко в таблиците е наред, можем да кажем, че с това нашият Code First метод за работа с база приключи. Данните се достъпват по същият начин, както и при Database First метода, чрез контекст класа ни.