МИНОБРНАУКИ РОССИИ

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ

ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)

Кафедра вычислительной техники

ОТЧЕТ

по лабораторной работе №3

по дисциплине «Основы разработки корпоративных систем на платформе .NET»

Тема: «Реализация слоя доступа к данным приложения»

Студент гр. 6305 _____ Стрельников В.Е.

Преподаватель _____ Пешехонов К.А.

Санкт-Петербург

2020

Содержание

Цель работы

Реализовать слой доступа к данным веб-API приложения с помощью ASP.NET Core 3.1 в среде JetBrains Rider.

Основные положения

В процессе создания веб-API приложения «Библиотека книг» с помощью ASP.NET Core будет сделано следующее:

1. Создание проекта, состоящего из слоёв (Core layer, Business logic layer, Data Access layer), что позволит сделать их независимыми друг от друга и легко дополняемыми\изменяемыми;
2. Реализация паттернов Repository и Unit of Work;
3. Реализация использования Entity Framework для работы с базой данных SQL Server Express;
4. Реализация использования Automapper для маппаинга моделей в ресурсы веб-API;
5. Реализация использования Swagger для формирования удобного веб-интерфейса тестирования приложения с помощью запросов;
6. Будут реализованы Unit тесты слоя бизнес логики приложения.

В данной лабораторной работе рассматривается реализация слоя доступа к данным приложения.

Выполнение

Слой доступа к данным очень важен, потому что именно так мы общаемся с нашей базой данных. И для этого .NET Core предоставляет надежную основу, Entity Framework Core. Entity Framework - это модель объектных отношений (ORM), которая отображает все таблицы и столбцы базы данных в объекты C#, что упрощает управление данными и выполнение запросов.

В первую очередь нужно определить, как поведение моделей, их ограничения и отношения. Мы могли бы определить все конфигурации непосредственно в

DbContext, но для лучше выделить для этого отдельные файлы. Создадим BookConfiguration и AuthorConfiguration.

```
namespace Book.DAL.Configurations
{
    public class BookConfiguration : IEntityTypeConfiguration<BookModel>
    {
        public void Configure(EntityTypeBuilder<BookModel> builder)
        {
            builder
                .HasKey(model => model.Id);

            builder
                .Property(model => model.Id)
                .UseIdentityColumn();

            builder
                .Property(model => model.Name)
                .IsRequired()
                .HasMaxLength(50);

            builder
                .HasOne(model => model.Author)
                .WithMany(artmodel => artmodel.Books)
                .HasForeignKey(model => model.AuthorId);

            builder
                .ToTable("Books");

        }
    }
}
```

```
namespace Book.DAL.Configurations
{
    public class AuthorConfiguration : IEntityTypeConfiguration<AuthorModel>
    {
        public void Configure(EntityTypeBuilder<AuthorModel> builder)
        {
            builder
                .HasKey(artmodel => artmodel.Id);

            builder
                .Property(model => model.Id)
                .UseIdentityColumn();

            builder
                .Property(model => model.Name)
                .IsRequired()
                .HasMaxLength(50);

            builder
                .ToTable("Authors");
        }
    }
}
```

Теперь можно приступить к работе с DbContext. Создадим файл BookDbContext.cs и опишем поведение, которое дает нам доступ к соответствующей таблице из базы.

```
namespace Book.DAL
{
    public class BookDbContext : DbContext
    {
        public DbSet<BookModel> Books { get; set; }
        public DbSet<AuthorModel> Authors { get; set; }

        public BookDbContext(DbContextOptions<BookDbContext> options)
            : base(options)
        {}

        protected override void OnModelCreating(ModelBuilder builder)
        {
            builder
                .ApplyConfiguration(new BookConfiguration());

            builder
                .ApplyConfiguration(new AuthorConfiguration());
        }
    }
}
```

Отметим, что если мы не планируем управлять или извлекать данные по отдельности из таблицы, нам необязательно добавлять для нее сеты, Entity Framework будет создавать эту таблицу, если модель имеет какое-либо отношение к другим моделям.

Теперь необходимо реализовать интерфейс взаимодействия с DbContext. Для этого реализуем интерфейсы созданные ранее в ядре проекта (Core layer).

В первую очередь реализуем базовый репозиторий.

```
namespace Book.DAL.Repositories
{
    public class Repository<TEntity> : IRepository<TEntity> where TEntity : class
    {
        protected readonly DbContext Context;

        public Repository(DbContext context)
        {
            this.Context = context;
        }

        public async Task AddAsync(TEntity entity)
        {
```

```
            await Context.Set<TEntity>().AddAsync(entity);
        }

        public async Task AddRangeAsync(IEnumerable<TEntity> entities)
        {
            await Context.Set<TEntity>().AddRangeAsync(entities);
        }

        public IEnumerable<TEntity> Find(Expression<Func<TEntity, bool>> predicate)
        {
            return Context.Set<TEntity>().Where(predicate);
        }

        public async Task<IEnumerable<TEntity>> GetAllAsync()
        {
            return await Context.Set<TEntity>().ToListAsync();
        }

        public ValueTask<TEntity> GetByIdAsync(int id)
        {
            return Context.Set<TEntity>().FindAsync(id);
        }

        public void Remove(TEntity entity)
        {
            Context.Set<TEntity>().Remove(entity);
        }

        public void RemoveRange(IEnumerable<TEntity> entities)
        {
            Context.Set<TEntity>().RemoveRange(entities);
        }

        public Task<TEntity> SingleOrDefaultAsync(Expression<Func<TEntity, bool>>
predicate)
        {
            return Context.Set<TEntity>().SingleOrDefaultAsync(predicate);
        }
    }
}
```

После определения всех основных операций мы готовы реализовать оставшиеся два репозитория.

BookRepository

```
namespace Book.DAL.Repositories
{
    public class BookRepository : Repository<BookModel>, IBookRepository
    {
        public BookRepository(DbContext context)
            : base(context)
        { }

        private BookDbContext BookDbContext => Context as BookDbContext;

        public async Task<IEnumerable<BookModel>> GetAllWithAuthorAsync()
        {
```

```
            return await BookDbContext.Books
                .Include(model => model.Author)
                .ToListAsync();
        }

        public async Task<BookModel> GetWithAuthorByIdAsync(int id)
        {
            return await BookDbContext.Books
                .Include(model => model.Author)
                .SingleOrDefaultAsync(model => model.Id == id);
        }

        public async Task<IEnumerable<BookModel>> GetAllWithAuthorByAuthorIdAsync(int
authorId)
        {
            return await BookDbContext.Books
                .Include(model => model.Author)
                .Where(model => model.AuthorId == authorId)
                .ToListAsync();
        }
    }
}
```

AuthorRepository

```
namespace Book.DAL.Repositories
{
    public class AuthorRepository : Repository<AuthorModel>, IAuthorRepository
    {
        public AuthorRepository(DbContext context)
            : base(context)
        { }

        private BookDbContext BookDbContext => Context as BookDbContext;

        public async Task<IEnumerable<AuthorModel>> GetAllWithBooksAsync()
        {
            return await BookDbContext.Authors
                .Include(artmodel => artmodel.Books)
                .ToListAsync();
        }

        public Task<AuthorModel> GetWithBooksByIdAsync(int id)
        {
            return BookDbContext.Authors
                .Include(artmodel => artmodel.Books)
                .SingleOrDefaultAsync(artmodel => artmodel.Id == id);
        }
    }
}
```

И также реализуем Unit of work, который позволит одновременно взаимодействовать с обеими репозиториями.

```
namespace Book.DAL
{
    public class UnitOfWork : IUnitOfWork
    {
        private readonly BookDbContext _context;
```

```
        private BookRepository _bookRepository;
        private AuthorRepository _authorRepository;

        public UnitOfWork(BookDbContext context)
        {
            this._context = context;
        }

        public IBookRepository Books => _bookRepository ??= new
BookRepository(_context);
        public IAuthorRepository Authors => _authorRepository ??= new
AuthorRepository(_context);

        public async Task<int> CommitAsync()
        {
            return await _context.SaveChangesAsync();
        }

        public void Dispose()
        {
            _context.Dispose();
        }
    }
}
```

Как и ранее, нужно добавить внедрение зависимостей (Dependency Injection), чтобы наше приложение знало, что, когда мы используем интерфейсы репозитория. В файле Startup API-слоя укажем:

services.AddScoped<IUnitOfWork, UnitOfWork>();

Где *Scoped*: для каждого запроса создается свой объект сервиса. То есть если в течение одного запроса есть несколько обращений к одному сервису, то при всех этих обращениях будет использоваться один и тот же объект сервиса.

Теперь нам нужно добавить строки подключения и указать нашему API, как и откуда брать данные. При сборке .NET Core проверяется каждое свойство в appsettings.json, которое совпадает с текущим окружением и переопределяет эти свойства.

```
{
  "ConnectionStrings": {
    "Default": "server=.\\SQLEXPRESS; database=Library; user id=sa; password=12345"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
```

Здесь ключевые слова:

1. `server` это сервер базы данных;
2. `database` это имя вашей данных;
3. `user` это пользователь с правами администратора;
4. `password` это пароль пользователя.

Осталось добавить связь базы с API добавив в Startup строку:

```
services.AddDbContext<BookDbContext>
    (options =>
    options.UseSqlServer(
        Configuration.GetConnectionString(
            "Default"),
        x => x.MigrationsAssembly("Book.DAL")));
```

Здесь мы добавляем BookContext, говорим использовать SqlServer, используя строки подключения по умолчанию в appsettings.json, и что наши миграции должны выполняться в Book.DAL.

После ввода соответствующей команды сгенерируем миграции, которые появятся в слове автоматически.

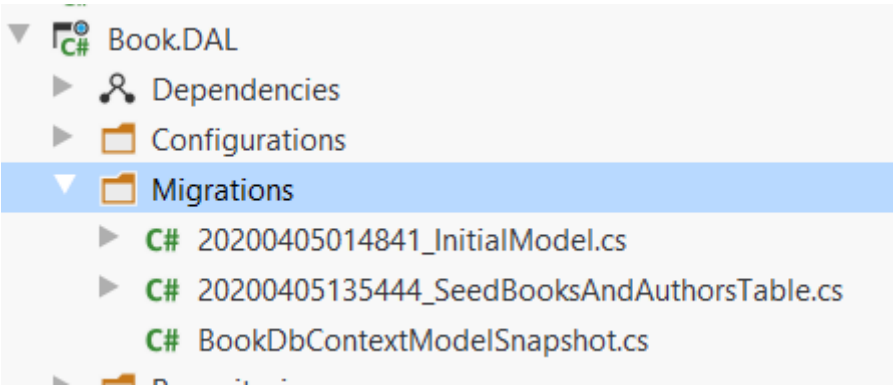

Рисунок 1. Миграции DAL слоя

Чтобы проинициализировать базу некоторыми данными, в файле SeedBooksAndAuthorsTable внесём следующие изменения (SQL команды), добавляющие авторов и книги.

```
namespace Book.DAL.Migrations
{
    public partial class SeedBooksAndAuthorsTable : Migration
    {
        protected override void Up(MigrationBuilder migrationBuilder)
```

```
        {
            migrationBuilder
                .Sql("INSERT INTO Authors (Name) VALUES ('Erich Maria Remarque')");
            migrationBuilder
                .Sql("INSERT INTO Authors (Name) VALUES ('Leo Tolstoy')");
            migrationBuilder
                .Sql("INSERT INTO Authors (Name) VALUES ('Friedrich Nietzsche')");
            migrationBuilder
                .Sql("INSERT INTO Authors (Name) VALUES ('Anton Chekhov')");


            migrationBuilder
                .Sql(@"INSERT INTO Books (Name, AuthorId) Values
                    ('Three Comrades', (SELECT Id FROM Authors WHERE Name = 'Erich
Maria Remarque'))");
            migrationBuilder
                .Sql(@"INSERT INTO Books (Name, AuthorId) Values
                    ('Arch of Triumph', (SELECT Id FROM Authors WHERE Name = 'Erich
Maria Remarque'))");
            migrationBuilder
                .Sql(@"INSERT INTO Books (Name, AuthorId) Values
                    ('The Black Obelisk', (SELECT Id FROM Authors WHERE Name = 'Erich
Maria Remarque'))");

            migrationBuilder
                .Sql(@"INSERT INTO Books (Name, AuthorId) Values
                    ('War and Peace', (SELECT Id FROM Authors WHERE Name = 'Leo
Tolstoy'))");
            migrationBuilder
                .Sql(@"INSERT INTO Books (Name, AuthorId) Values
                    ('Anna Karenina', (SELECT Id FROM Authors WHERE Name = 'Leo
Tolstoy'))");
            migrationBuilder
                .Sql(@"INSERT INTO Books (Name, AuthorId) Values
                    ('The Death of Ivan Ilyich', (SELECT Id FROM Authors WHERE Name =
'Leo Tolstoy'))");

            migrationBuilder
                .Sql(@"INSERT INTO Books (Name, AuthorId) Values
                    ('Thus Spoke Zarathustra', (SELECT Id FROM Authors WHERE Name =
'Friedrich Nietzsche'))");
            migrationBuilder
                .Sql(@"INSERT INTO Books (Name, AuthorId) Values
                    ('Twilight of the Idols', (SELECT Id FROM Authors WHERE Name =
'Friedrich Nietzsche'))");
            migrationBuilder
                .Sql(@"INSERT INTO Books (Name, AuthorId) Values
                    ('Ecce Homo', (SELECT Id FROM Authors WHERE Name = 'Friedrich
Nietzsche'))");

            migrationBuilder
                .Sql(@"INSERT INTO Books (Name, AuthorId) Values
                    ('Misery', (SELECT Id FROM Authors WHERE Name = 'Anton
Chekhov'))");
            migrationBuilder
                .Sql(@"INSERT INTO Books (Name, AuthorId) Values
                    ('The Chameleon', (SELECT Id FROM Authors WHERE Name = 'Anton
Chekhov'))");
            migrationBuilder
                .Sql(@"INSERT INTO Books (Name, AuthorId) Values
```

```
                    ('Fat and Thin ', (SELECT Id FROM Authors WHERE Name = 'Anton
Chekhov'))");


        }

        protected override void Down(MigrationBuilder migrationBuilder)
        {
            migrationBuilder
                .Sql("DELETE FROM Books");

            migrationBuilder
                .Sql("DELETE FROM Authors");
        }
    }
}
```

Убедимся в том, что созданная нами база данных существует, запустив SQL Server Management Studio.

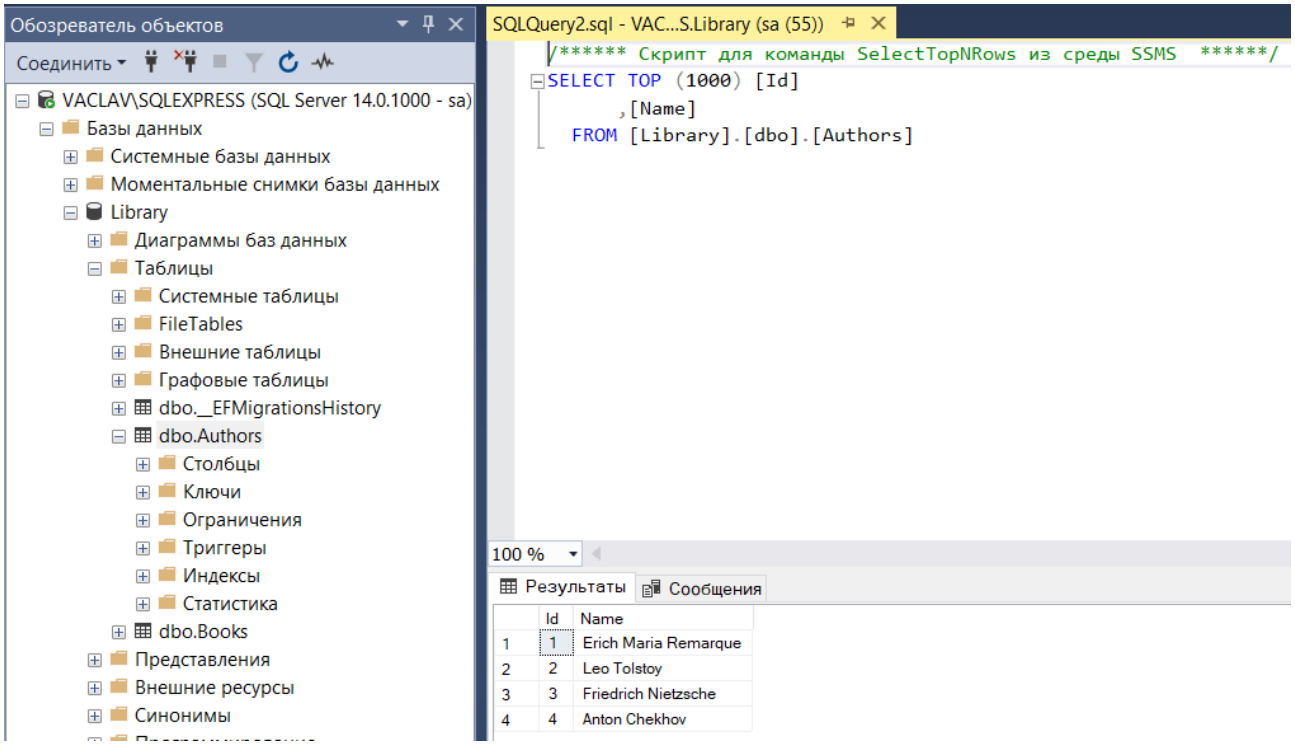

Рисунок 2. Вывод авторов книг из соответствующей таблицы базы Library.

Итак, теперь мы готовы начать отправлять запросы в нашу базу данных.

Вывод

В процессе выполнения данной лабораторной работы был реализован слой доступа к данным веб-API приложения «Библиотека книг» с помощью ASP.NET Core 3.1 в среде JetBrains Rider.