МИНОБРНАУКИ РОССИИ

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ

ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)

Кафедра вычислительной техники

ОТЧЕТ

по лабораторной работе №2

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

Тема: «Реализация слоя бизнес логики приложения»

Студент гр. 6305 _____ Стрельников В.Е.

Преподаватель _____ Пешехонов К.А.

Санкт-Петербург

2020

Содержание

Цель работы

Реализовать слой бизнес логики 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 ядра приложения и слоя бизнес логики приложения.

Выполнение

Перед тем как реализовывать бизнес логику приложения необходимо создать Core layer (ядро приложения). Основными сущностями будут BookModel & AuthorModel (книга и автор книги).

Определим модель Book. Мы используем отношение один ко многим с моделью Author.

```
namespace Book.Core.Models
{
    public class BookModel
```

```
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public int AuthorId { get; set; }
        public AuthorModel Author { get; set; }
    }
}
```

Модель автора в свою очередь имеет следующий вид:

```
namespace Book.Core.Models
{
    public class AuthorModel
    {
        public AuthorModel()
        {
            Books = new Collection<BookModel>();
        }

        public int Id { get; set; }
        public string Name { get; set; }
        public ICollection<BookModel> Books { get; set; }
    }
}
```

После определение моделей можно перейти к реализации репозиториев и шаблона Repository. Шаблон Repository, простыми словами, инкапсулирует операции с базой данных (ORM или другие методы доступа к базе данных) в одном месте. Происходит абстрагирование уровня доступа к данным. Это имеет много преимуществ, таких как:

1. Возможность повторного использования - не нужно переписывать код для использования доступа к базе данных;
2. Тестируемость - можем анализировать свои действия с базой данных без логики доступа к данным;
3. Разделение проблем - доступ к данным является обязанностью только хранилища;
4. Разъединенный код - если мы захотим изменить среду хранения, это потребует гораздо меньше усилий.

Использования паттерна Unit of work отвечает за отслеживание списка изменений во время любой транзакции и его фиксацию.

Несмотря на то, что Entity Framework внутренне реализует шаблон Repository и Unit of work, хорошей практикой является создание собственного репозитория для отделения проекта от Entity Framework. Таким образом, мы не сильно привязаны к нему, и если мы хотим перейти с этой версии Entity Framework на другую или на другую среду ORM, нам потребуется гораздо меньше усилий для этого. Также это помогает при использовании команд Mock при тестировании.

Вначале будет реализован интерфейс репозитория, который будет базовым, с базовыми операциями доступа к данным.

```
namespace Book.Core.Repositories
{
    public interface IRepository<TEntity> where TEntity : class
    {
        ValueTask<TEntity> GetByIdAsync(int id);
        Task<IEnumerable<TEntity>> GetAllAsync();
        IEnumerable<TEntity> Find(Expression<Func<TEntity, bool>> predicate);
        Task<TEntity> SingleOrDefaultAsync(Expression<Func<TEntity, bool>> predicate);
        Task AddAsync(TEntity entity);
        Task AddRangeAsync(IEnumerable<TEntity> entities);
        void Remove(TEntity entity);
        void RemoveRange(IEnumerable<TEntity> entities);
    }
}
```

Аналогичные интерфейсы создадим для моделей книг и автора книг:

```
namespace Book.Core.Repositories
{
    public interface IBookRepository : IRepository<BookModel>
    {
        Task<IEnumerable<BookModel>> GetAllWithAuthorAsync();
        Task<BookModel> GetWithAuthorByIdAsync(int id);
        Task<IEnumerable<BookModel>> GetAllWithAuthorByAuthorIdAsync(int authorId);
    }
}
```

```
namespace Book.Core.Repositories
{
    public interface IAuthorRepository : IRepository<AuthorModel>
    {
        Task<IEnumerable<AuthorModel>> GetAllWithBooksAsync();
        Task<AuthorModel> GetWithBooksByIdAsync(int id);
    }
}
```

Также создадим интерфейс для Unit of Work:

```
namespace Book.Core
{
    public interface IUnitOfWork : IDisposable
    {
        IBookRepository Books { get; }
        IAuthorRepository Authors { get; }
        Task<int> CommitAsync();
    }
}
```

Здесь же, в ядре (Core layer) опишем интерфейсы-сервисы, которые в дальнейшем будут являться основой бизнес логики приложения – связующим звеном между API и Data layer. Сделаем это для обоих моделей.

```
namespace Book.Core.Services
{
    public interface IBookService
    {
        Task<IEnumerable<BookModel>> GetAllWithAuthor();
        Task<BookModel> GetBookById(int id);
        Task<IEnumerable<BookModel>> GetBooksByAuthorId(int authorId);
        Task<BookModel> CreateBook(BookModel newBook);
        Task UpdateBook(BookModel bookToBeUpdated, BookModel book);
        Task DeleteBook(BookModel book);
    }
}
```

```
   public interface IAuthorService
   {
        Task<IEnumerable<AuthorModel>> GetAllAuthors();
        Task<AuthorModel> GetAuthorById(int id);
        Task<AuthorModel> CreateAuthor(AuthorModel newAuthor);
        Task UpdateAuthor(AuthorModel authorToBeUpdated, AuthorModel author);
        Task DeleteAuthor(AuthorModel author);
    }
}
```

Таким образом бизнес логика в будущем будет реализовывать стандартный набор CRUD функций, что означает использование четырёх базовых функций, используемых при работе с базами данных: создание (create), чтение (read), модификация (update), удаление (delete).

Основной проект будет содержать всю структуру бизнес-логики приложения, все, что должно рассказать, как оно должно работать. Так что, если нам нужно значительное изменение технологии или даже логики, мы можем просто

отключить старый модуль и подключить новый, который соответствует нашим основным проектным моделям.

Перейдём непосредственно к Business logic layer, который отвечает за бизнес-логику и взаимодействует с уровнем доступа к данным. Ключевым моментом здесь является то, что мы будем использовать шаблон Unit Of Work для управления этим интерфейсом, поэтому нам не нужно напрямую добавлять сюда используемый в будущем на слове доступа к данным DbContext.

В слое понадобится только реализация двух сервисных интерфейсов, определенных в Core, а именно IBookService и IAuthorService.

```
amespace Book.BLL
{
    public class BookService : IBookService
    {
        private readonly IUnitOfWork _unitOfWork;

        public BookService(IUnitOfWork unitOfWork)
        {
            this._unitOfWork = unitOfWork;
        }

        public async Task<BookModel> CreateBook(BookModel newBook)
        {
            if (newBook == null)
            {
                throw new ArgumentNullException(nameof(newBook));
            }

            await _unitOfWork.Books.AddAsync(newBook);
            await _unitOfWork.CommitAsync();
            return newBook;
        }

        public async Task DeleteBook(BookModel book)
        {
            _unitOfWork.Books.Remove(book);
            await _unitOfWork.CommitAsync();
        }

        public async Task<IEnumerable<BookModel>> GetAllWithAuthor()
        {
            return await _unitOfWork.Books
                .GetAllWithAuthorAsync();
        }

        public async Task<BookModel> GetBookById(int id)
        {
            return await _unitOfWork.Books
                .GetWithAuthorByIdAsync(id);
```

```
        }

        public async Task<IEnumerable<BookModel>> GetBooksByAuthorId(int authorId)
        {
            return await _unitOfWork.Books
                .GetAllWithAuthorByAuthorIdAsync(authorId);
        }

        public async Task UpdateBook(BookModel bookToBeUpdated, BookModel book)
        {
            if ((bookToBeUpdated == null) || (book == null))
            {
                throw new ArgumentNullException(nameof(bookToBeUpdated));
            }

            bookToBeUpdated.Name = book.Name;
            bookToBeUpdated.AuthorId = book.AuthorId;

            await _unitOfWork.CommitAsync();
        }
    }
}


namespace Book.BLL
{
    public class AuthorService : IAuthorService
    {
        private readonly IUnitOfWork _unitOfWork;

        public AuthorService(IUnitOfWork unitOfWork)
        {
            this._unitOfWork = unitOfWork;
        }

        public async Task<AuthorModel> CreateAuthor(AuthorModel newAuthor)
        {
            await _unitOfWork.Authors
                .AddAsync(newAuthor);

            return newAuthor;
        }

        public async Task DeleteAuthor(AuthorModel author)
        {
            _unitOfWork.Authors.Remove(author);

            await _unitOfWork.CommitAsync();
        }

        public async Task<IEnumerable<AuthorModel>> GetAllAuthors()
        {
            return await _unitOfWork.Authors.GetAllAsync();
        }

        public async Task<AuthorModel> GetAuthorById(int id)
        {
            return await _unitOfWork.Authors.GetByIdAsync(id);
        }
```

```
        public async Task UpdateAuthor(AuthorModel authorToBeUpdated, AuthorModel
author)
        {
            if ((authorToBeUpdated == null) || (author == null))
            {
                throw new ArgumentNullException(nameof(authorToBeUpdated));
            }

            authorToBeUpdated.Name = author.Name;

            await _unitOfWork.CommitAsync();
        }
    }
}
```

При этом мы абстрагируем нашу бизнес-логику от уровня представления, который является нашим API. Осталось добавить Dependency Injection для сервисов, добавив следующие строки в Startup.cs, расположенном в Presentation layer (непосредственно веб-API):

services.AddTransient<IBookService, BookService>();

services.AddTransient<IAuthorService, AuthorService>();

Отметим, что означает ключевое слово *Transient*: при каждом обращении к сервису создается новый объект сервиса. В течение одного запроса может быть несколько обращений к сервису, соответственно при каждом обращении будет создаваться новый объект. Подобная модель жизненного цикла наиболее подходит для легковесных сервисов, которые не хранят данных о состоянии.

Unit testing — процесс в программировании, позволяющий проверить на корректность отдельные модули исходного кода программы, наборы из одного или более программных модулей вместе с соответствующими управляющими данными, процедурами использования и обработки. Идея состоит в том, чтобы писать тесты для каждой нетривиальной функции или метода. Это позволяет достаточно быстро проверить, не привело ли очередное изменение кода к регрессии, то есть к появлению ошибок в уже оттестированных местах программы, а также облегчает обнаружение и устранение таких ошибок.

Для слоя бизнес логики были реализованы следующие тесты с использованием NuGet package XUnit.

```
namespace Book.BLL.Unit
{
    public class BookServiceTests
    {
        [Fact]
        public async Task UpdateBook_ArgumentNullException()
        {
            // Arrange
            var book = new BookModel();
            var bookToBeUpdated = new BookModel();
            var unitOfWork = new Mock<IUnitOfWork>();

            var service = new BookService(unitOfWork.Object);

            // Act
            book = null;

            // Assert
            await Assert.ThrowsAsync<ArgumentNullException>(() =>
service.UpdateBook(bookToBeUpdated, book));
        }

        [Fact]
        public void UpdateBook_Success()
        {
            // Arrange
            var book = new BookModel
            {
                AuthorId = 1,
                Name = "Mathematics"
            };
            var bookToBeUpdated = new BookModel
            {
                AuthorId = 2,
                Name = "History"
            };
            var unitOfWork = new Mock<IUnitOfWork>();
            unitOfWork.Setup(x => x.CommitAsync());

            var bookData = new BookService(unitOfWork.Object);

            // Act
            var result = bookData.UpdateBook(bookToBeUpdated, book);

            // Assert
            Assert.Equal(bookToBeUpdated.Name, book.Name);
            Assert.Equal(bookToBeUpdated.AuthorId, book.AuthorId);
        }
    }
}
```

```
namespace Book.BLL.Unit
{
```

```
    public class AuthorServiceTests
    {
        [Fact]
        public async Task UpdateAuthor_ArgumentNullException()
        {
            // Arrange
            var author = new AuthorModel();
            var authorToBeUpdated = new AuthorModel();
            var unitOfWork = new Mock<IUnitOfWork>();

            var service = new AuthorService(unitOfWork.Object);

            // Act
            author = null;

            // Assert
            await Assert.ThrowsAsync<ArgumentNullException>(() =>
service.UpdateAuthor(authorToBeUpdated, author));
        }

        [Fact]
        public void UpdateAuthor_Success()
        {
            // Arrange
            var author = new AuthorModel
            {
                Name = "Ivan"
            };
            var authorToBeUpdated = new AuthorModel
            {
                Name = "Pyotr"
            };
            var unitOfWork = new Mock<IUnitOfWork>();
            unitOfWork.Setup(x => x.CommitAsync());

            var bookData = new AuthorService(unitOfWork.Object);

            // Act
            var result = bookData.UpdateAuthor(authorToBeUpdated, author);

            // Assert
            Assert.Equal(authorToBeUpdated.Name, author.Name);
        }
    }
}
```

Все тесты проходят успешно.

Вывод

В процессе выполнения данной лабораторной работы были реализованы ядро и слой бизнес логики веб-API приложения «Библиотека книг» с помощью ASP.NET Core 3.1 в среде JetBrains Rider.