



Формирование архитектуры приложения (Часть 3)

Введение

В рамках данной работы вы завершите построение многослойной архитектуры и рассмотрите некоторые команды для работы с миграциями моделей базы данных.

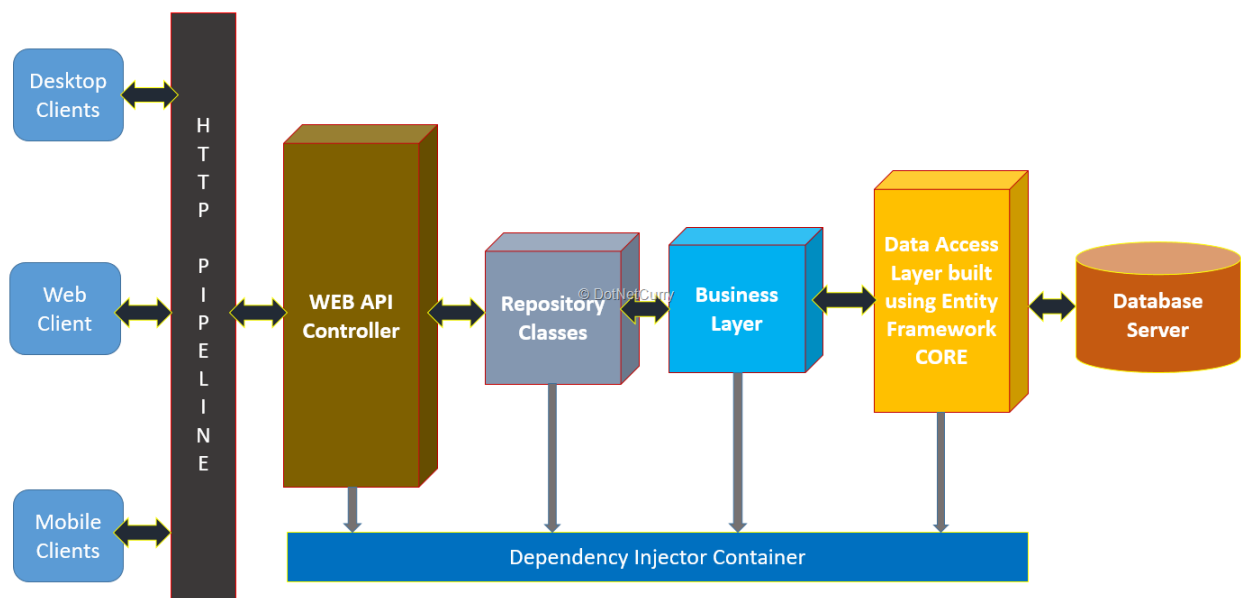
Проблема с сильной связанностью слоёв проекта

Рассмотрим ещё раз слои N-tier архитектуры:

1. Слой представления “Presentation Layer” - Слой, который предоставляет возможность пользователю взаимодействовать с вашим приложением
2. Слой бизнес логики “Business Logic Layer” - Слой, где будет написан ваш основной код, который выполняет определенные задачи.
3. Слой доступа к данным “Data Access Layer” - Слой, в котором написана реализация для получения данных из какого-либо источника данных

Посмотрите на схему ниже и обратите внимание на блок “Dependency Injector Container”.

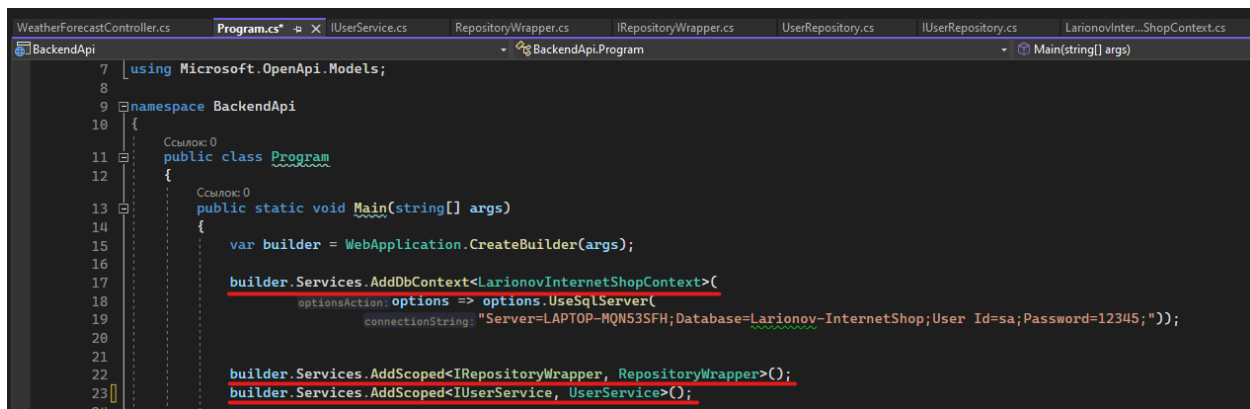
Какую функцию выполняет данный блок и почему на данный блок ссылаются все слои N-tier архитектуры?



Dependency Injection (DI) [рус. Внедрение Зависимостей] - это механизм, который позволяет сделать компоненты программы слабосвязанными, а всю программу в целом более гибкой, более адаптируемой и расширяемой. (подробнее [ТУТ](#), а с примером [ТУТ](#))

Вы уже успели использовать **DI** при выполнении 2-й части данной практической работы.

Когда в файле Program.cs вызывается метод расширения `.AddScoped<T>`, происходит связывание интерфейса (Описание методов которые класс должен реализовать) и класса (Реализация методов, которые описал интерфейс). Таким образом в контейнер зависимостей добавляется реализация для интерфейса `IUserService` и интерфейса `IRepositoryWrapper`. Помимо этого, в контейнер зависимостей также можно добавлять сервисы или классы (В данном случае это контекст базы данных).



```
7 using Microsoft.OpenApi.Models;
8
9 namespace BackendApi
10 {
11     Ссылка: 0
12     public class Program
13     {
14         Ссылка: 0
15         public static void Main(string[] args)
16         {
17             var builder = WebApplication.CreateBuilder(args);
18
19             builder.Services.AddDbContext<LarionovInternetShopContext>(  
                optionsAction: options => options.UseSqlServer(  
                    connectionString: "Server=LAPTOP-MQNS3SFH;Database=Larionov-InternetShop;User Id=sa;Password=12345;");  
            );
20
21             builder.Services.AddScoped<IRepositoryWrapper, RepositoryWrapper>();
22             builder.Services.AddScoped<IUserService, UserService>();
23 }
```

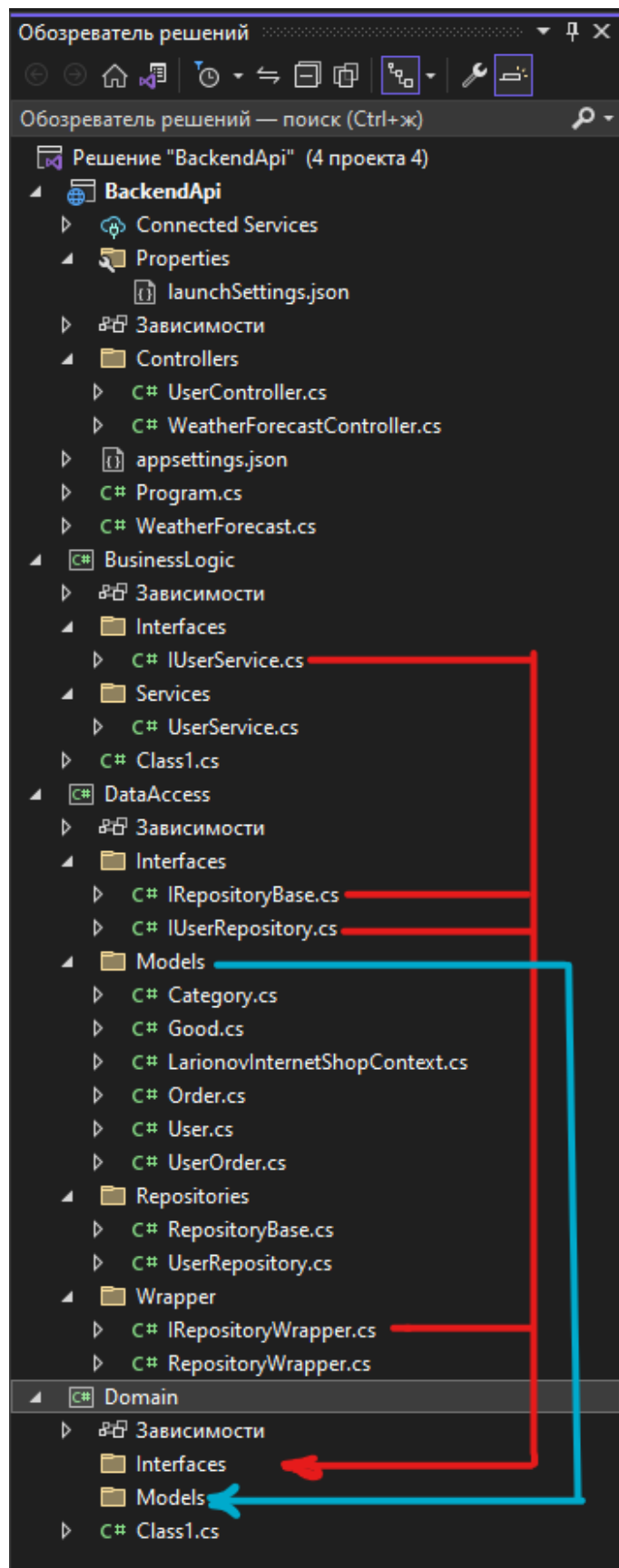
Однако, даже при том, что мы создали несколько слоев и разместили для каждого из слоев соответствующий код - **мы не добились слабой связанности между проектами!**

Причиной этому является то, что Слой “BusinessLogic” напрямую зависит от слоя DataAccess.

Для решение этой проблемы мы и создали слой Domain, который будет хранить внутри себя абстракции (абстрактные классы и интерфейсы). А детали (реализация этих интерфейсов) будут непосредственно лежать в отдельных слоях.

Решение проблемы

Откройте ваш проект. Переместите (Перетащите файлы) следующие интерфейсы и модели в проект Domain.



Измените пространство имён **для всех моделей БД** из DataAccess в Domain, которые лежат в проекте Domain:

До

```
1 using System;
2 using System.Collections.Generic;
3
4 namespace DataAccess.Models
5 {
6     Ссылка: 4
7     public partial class Category
8     {
9         Ссылка: 0
10        public Category()
11        {
12            Goods = new HashSet<Good>();
13        }
14
15        Ссылка: 0
16        public int Id { get; set; }
17        Ссылка: 1
18        public string Name { get; set; } = null!;
19        Ссылка: 1
20        public string Description { get; set; } = null!;
21
22        Ссылка: 2
23        public virtual ICollection<Good> Goods { get; set; }
24    }
25 }
```

После

```
Domain
Domain.Models.Category

1 namespace Domain.Models
2 {
3     Ссылка: 4
4     public class Category
5     {
6         Ссылка: 0
7         public Category()
8         {
9             Goods = new HashSet<Good>();
10        }
11
12        Ссылка: 0
13        public int Id { get; set; }
14        Ссылка: 1
15        public string Name { get; set; } = null!;
16        Ссылка: 1
17        public string Description { get; set; } = null!;
18
19        Ссылка: 2
20        public virtual ICollection<Good> Goods { get; set; }
21    }
22 }
```

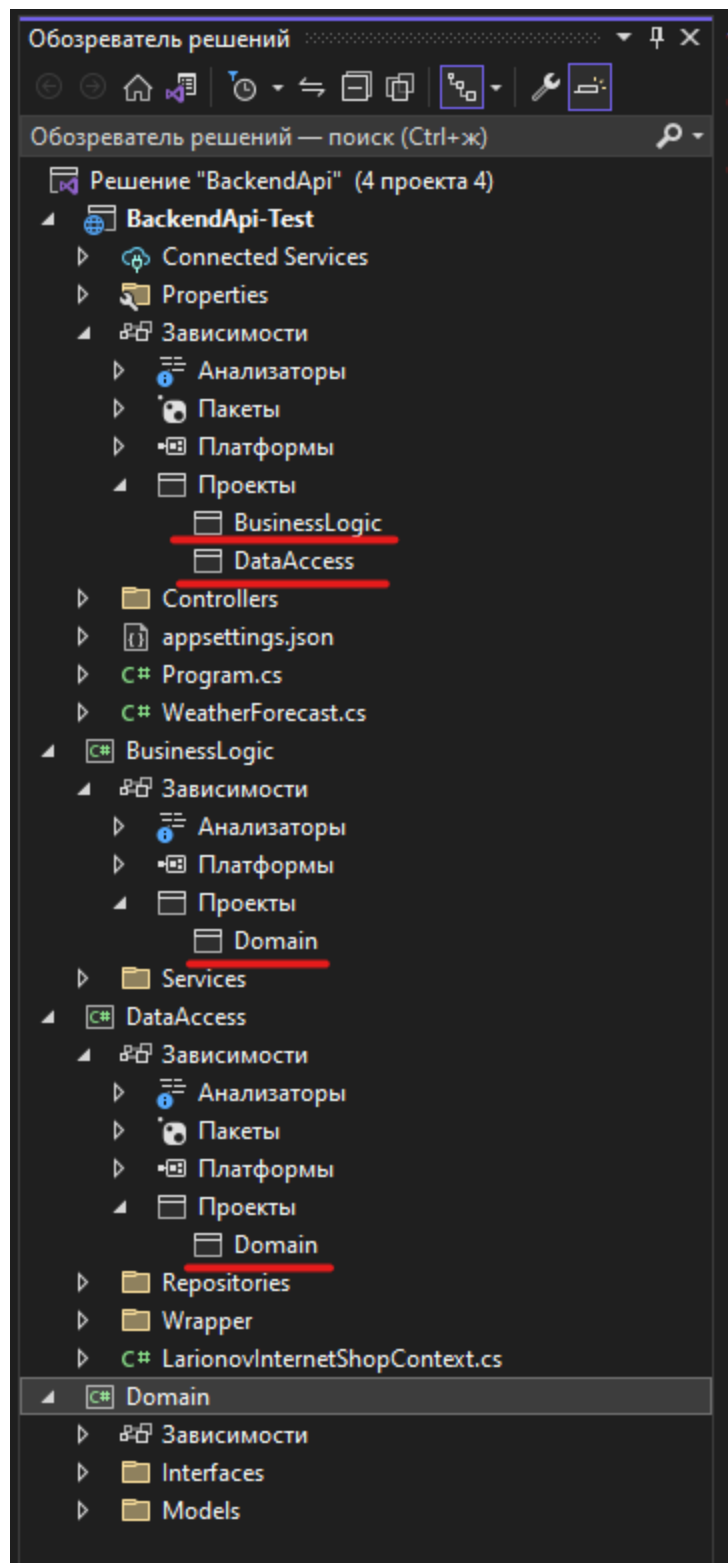
Таким же образом измените пространство имён и для всех интерфейсов в проекте Domain:

```
Domain
Domain.Interfaces.IUserService

1 using Domain.Models;
2
3 namespace Domain.Interfaces
4 {
5     Ссылка: 4
6     public interface IUserService
7     {
8         Ссылка: 2
9         Task<List<User>> GetAll();
10        Ссылка: 2
11        Task<User> GetById(int id);
12        Ссылка: 2
13        Task Create(User model);
14        Ссылка: 2
15        Task Update(User model);
16        Ссылка: 2
17        Task Delete(int id);
18    }
19 }
```

Привязка проектов

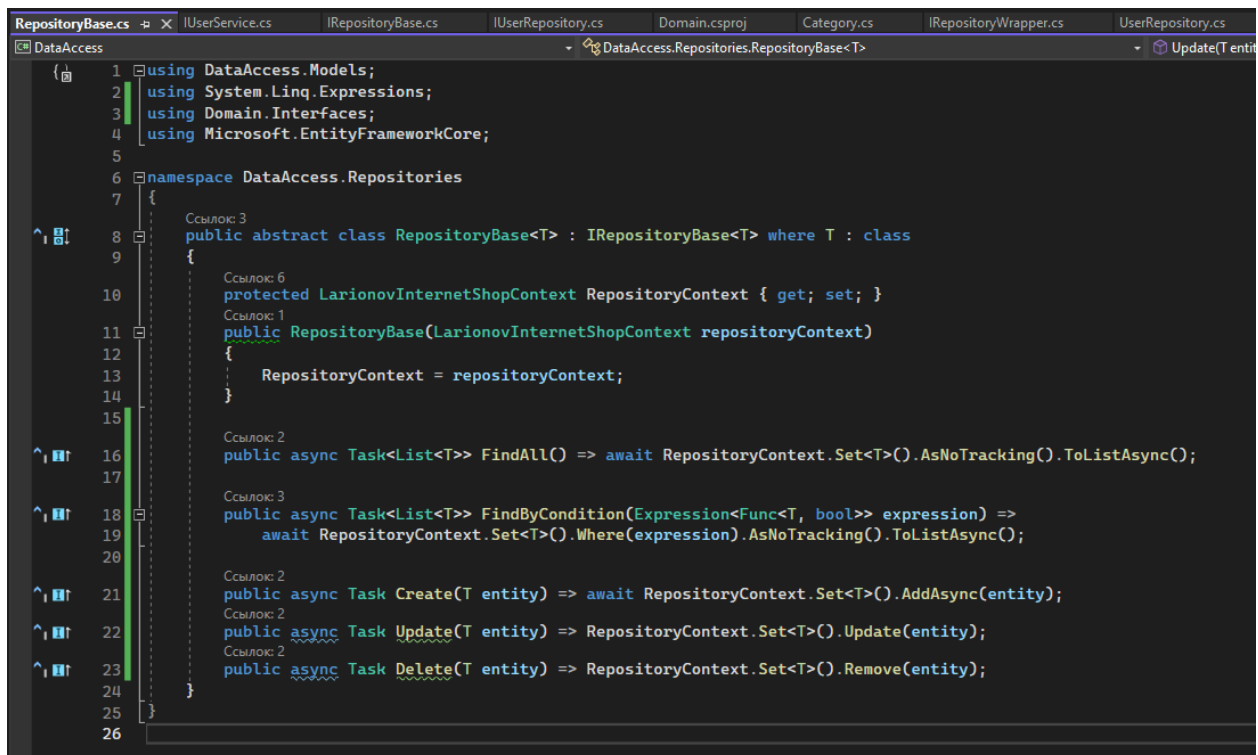
Так как все интерфейсы расположены в проекте Domain - потребуется изменить зависимости проектов:



Переход на асинхронность

Для того, чтобы приложение работало более эффективно (распределяло серьезные задачи на отдельные потоки), изменим некоторые методы для классов RepositoryBase, UserService, RepositoryWrapper.

Перейдите в класс RepositoryBase и преобразуйте его следующим образом:



```
1 using DataAccess.Models;
2 using System.Linq.Expressions;
3 using Domain.Interfaces;
4 using Microsoft.EntityFrameworkCore;
5
6 namespace DataAccess.Repositories
7 {
8     public abstract class RepositoryBase<T> : IRepositoryBase<T> where T : class
9     {
10         protected LarionovInternetShopContext RepositoryContext { get; set; }
11         public RepositoryBase(LarionovInternetShopContext repositoryContext)
12         {
13             RepositoryContext = repositoryContext;
14         }
15
16         public async Task<List<T>> FindAll() => await RepositoryContext.Set<T>().AsNoTracking().ToListAsync();
17
18         public async Task<List<T>> FindByCondition(Expression<Func<T, bool>> expression) =>
19             await RepositoryContext.Set<T>().Where(expression).AsNoTracking().ToListAsync();
20
21         public async Task Create(T entity) => await RepositoryContext.Set<T>().AddAsync(entity);
22         public async Task Update(T entity) => RepositoryContext.Set<T>().Update(entity);
23         public async Task Delete(T entity) => RepositoryContext.Set<T>().Remove(entity);
24     }
25 }
26
```

Перейдите в класс RepositoryWrapper и измените метод Save()

```

RepositoryWrapper.cs  RepositoryBase.cs  IUserService.cs  IRepositoryBase.cs  IUserRepository.cs  Domain.csproj  Category.cs
DataAccess
1  using DataAccess.Models;
2  using DataAccess.Repositories;
3  using Domain.Interfaces;
4
5  namespace DataAccess.Wrapper
6  {
7      Ссылка: 2
8      public class RepositoryWrapper : IRepositoryWrapper
9      {
10         private LarionovInternetShopContext _repoContext;
11
12         private IUserRepository _user;
13         Ссылка: 7
14         public IUserRepository User
15         {
16             get
17             {
18                 if (_user == null)
19                 {
20                     _user = new UserRepository(_repoContext);
21                 }
22                 return _user;
23             }
24         }
25         Ссылка: 0
26         public RepositoryWrapper(LarionovInternetShopContext repositoryContext)
27         {
28             _repoContext = repositoryContext;
29         }
30         Ссылка: 4
31         public async Task Save()
32         {
33             await _repoContext.SaveChangesAsync();
34         }
35     }

```

Адаптируйте класс UserService под текущие изменения, например

```

using Domain.Interfaces;
using Domain.Models;

namespace BusinessLogic.Services
{
    public class UserService : IUserService
    {
        private IRepositoryWrapper _repositoryWrapper;

        public UserService(IRepositoryWrapper repositoryWrapper)
        {
            _repositoryWrapper = repositoryWrapper;
        }

        public async Task<List<User>> GetAll()
        {
            return await _repositoryWrapper.User.FindAll();
        }
    }

```

```

    public async Task<User> GetById(int id)
    {
        var user = await _repositoryWrapper.User
            .FindByCondition(x => x.Id == id);
        return user.First();
    }

    public async Task Create(User model)
    {
        await _repositoryWrapper.User.Create(model);
        _repositoryWrapper.Save();
    }

    public async Task Update(User model)
    {
        _repositoryWrapper.User.Update(model);
        _repositoryWrapper.Save();
    }

    public async Task Delete(int id)
    {
        var user = await _repositoryWrapper.User
            .FindByCondition(x => x.Id == id);

        _repositoryWrapper.User.Delete(user.First());
        _repositoryWrapper.Save();
    }
}

```

Так как мы изменили возвращаемый тип данных для некоторых методов - нам потребуется изменить интерфейс для них:

Интерфейс IRepositoryBase

Интерфейс IRepositoryWrapper

Интерфейс IUserService

```
1 using Domain.Models;
2
3 namespace Domain.Interfaces
4 {
5     Ссылка: 4
6     public interface IUserService
7     {
8         Ссылка: 2
9         Task<List<User>> GetAll();
10        Ссылка: 2
11        Task<User> GetById(int id);
12        Ссылка: 2
13        Task Create(User model);
14        Ссылка: 2
15        Task Update(User model);
16        Ссылка: 2
17        Task Delete(int id);
18    }
19 }
```

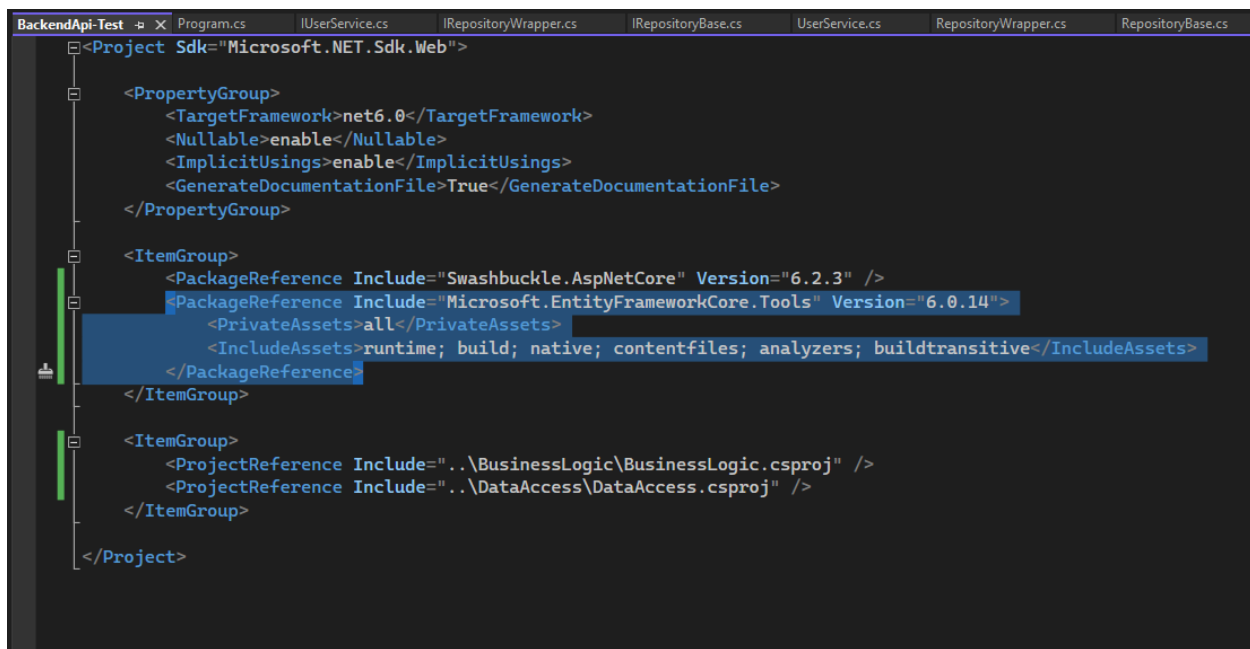
Запустите проект и проверьте работу

Работа с Миграциями в EF Core

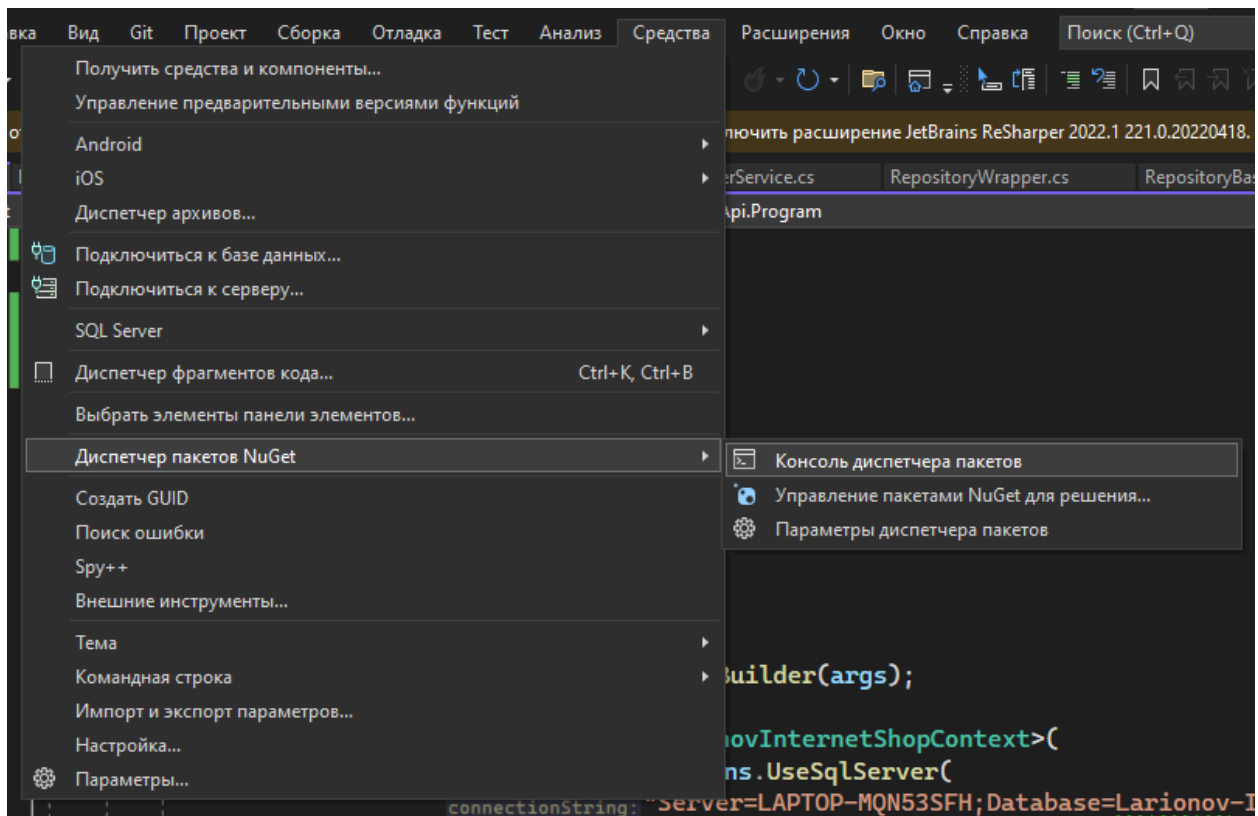
Приходилось ли вам работать с вашим проектом, который взаимодействует с базой данных? И как часто приходится сохранять скрипты для формирования бд, чтобы сформировать вашу базу данных на разных рабочих машинах?

При помощи миграций EF Core вы можете упростить разворачивание вашей базы данных.

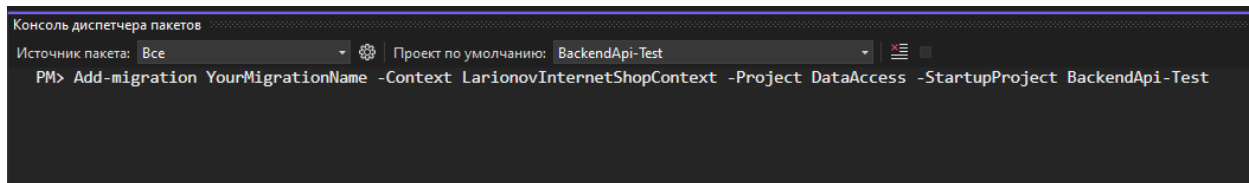
Добавьте библиотеку Microsoft.EntityFrameworkCore.Tools в проект API



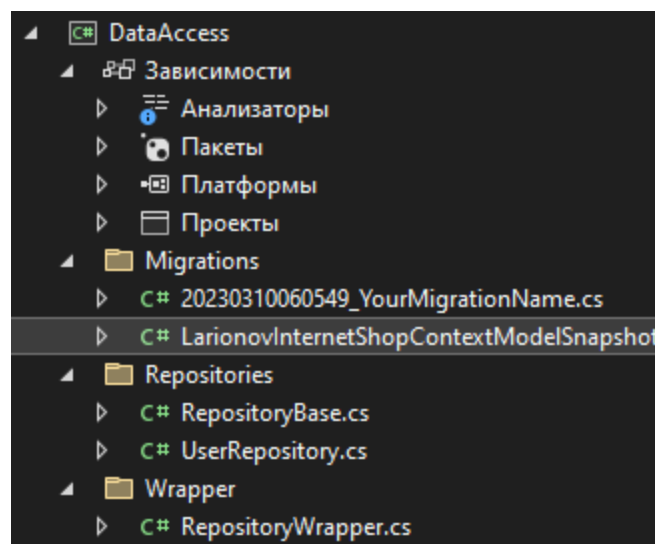
Перейдите в консоль диспетчера пакетов



Напишите команду **Add-migration MigrationName -Context LarionovInternetShopContext -Project DataAccess -StartupProject BackendApi**

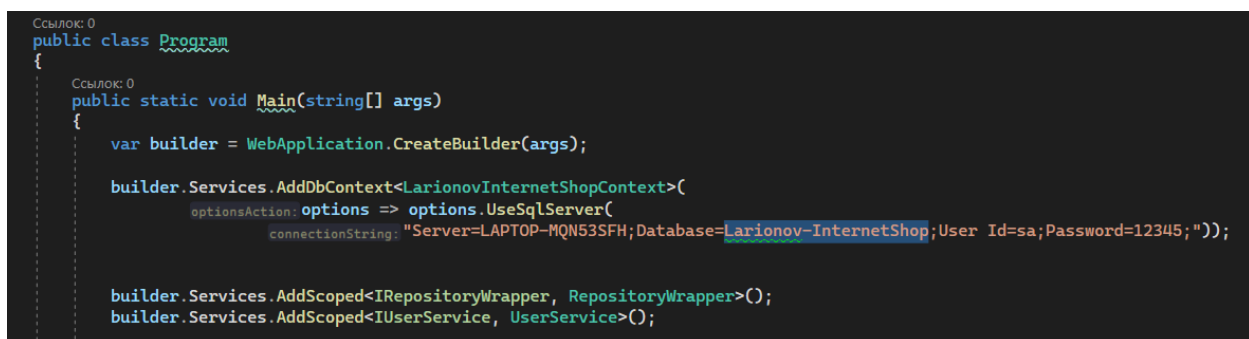


После этого у вас сформируются файлы для миграции в папке Migration



Для проверки работы, сформируем новую бд при помощи миграций

Измените имя бд в строке подключения на любое другое



И выполните в консоли диспетчера пакетов Nuget команду Update-Database

```
Консоль диспетчера пакетов
Источник пакета: Все
PM> Update-Database
Build started...
Build succeeded.
Microsoft.EntityFrameworkCore.Infrastructure[10403]
    Entity Framework Core 6.0.14 initialized 'Larionov-InternetShop'
Microsoft.EntityFrameworkCore.Database.Command[20101]
```

Проверьте, что база данных сформировалась со всеми вашими таблицами

