

МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ
КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМ. ТАРАСА
ШЕВЧЕНКА
ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
Кафедра мережеских та інтернет технологій
СУЧАСНІ ІНТЕРНЕТ ТЕХНОЛОГІЇ
РОБОТА З ДАНИМИ В ASP.NET CORE.
РЕАЛІЗАЦІЯ ШАБЛОНУ REPOSITORY
Лабораторне заняття №2
Заяць Діани Юріївни

Хід виконання роботи:

2.2 Створення інтерфейсів репозиторію

2.2.1 Базовий інтерфейс

Створюємо папку Interfaces у бібліотеці даних.

Далі додаємо новий item - Interface і називаємо його IRepository - це буде наш базовий інтерфейс

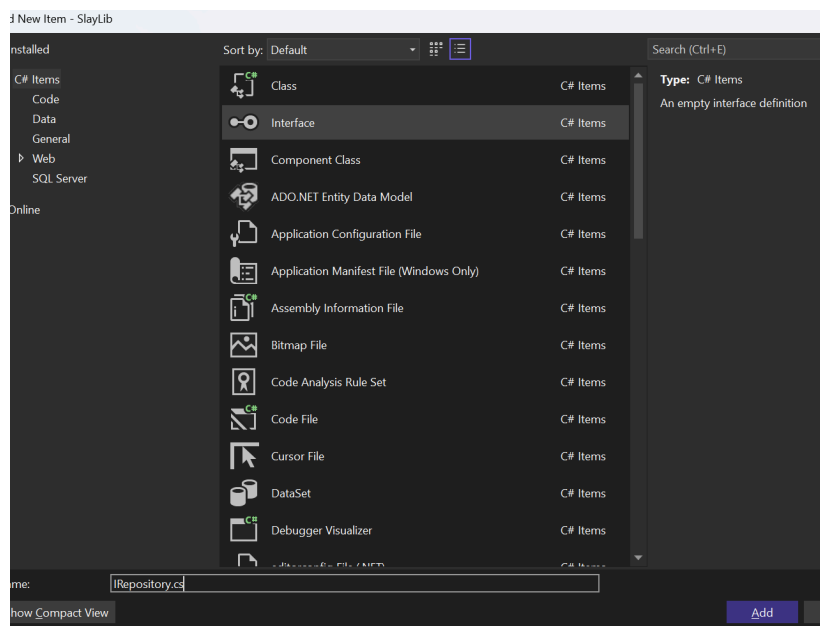


Рис. 2.2.1.1 - Створення базового інтерфейсу

Тепер реалізуємо сам інтерфейс. Нам потрібно додати базові операції для роботи з даними (отримання всіх записів, пошук за умовою, додавання, оновлення, видалення).

```

using System;
using System.Collections.Generic;
using System.Linq.Expressions;
using System.Threading.Tasks;

namespace SlayLib.Interfaces
{
    1 reference
    public interface IRepository
    {
        0 references
        Task<IEnumerable<T>> GetAllAsync<T>() where T : class;
        0 references
        Task<IEnumerable<T>> FindAsync<T>(Expression<Func<T, bool>> predicate) where T : class;
        0 references
        Task AddAsync<T>(T entity) where T : class;
        0 references
        Task UpdateAsync<T>(T entity) where T : class;
        0 references
        Task RemoveAsync<T>(T entity) where T : class;
        0 references
        Task SaveChangesAsync();
    }
}

```

Рис. 2.2.1.2 - Реалізація операцій для роботи з даними

2.2.2 Конкретний інтерфейс

Створюємо цей інтерфейс аналогічно до попереднього

Перейдемо до реалізації, додамо в нього метод пошуку за поштою, відповідно оновимо клас ApplicationUser

```

using System.Threading.Tasks;
using SlayLib.Models;

namespace SlayLib.Interfaces
{
    0 references
    public interface IMitRepository : IRepository
    {
        0 references
        Task<ApplicationUser> GetUserByEmailAsync(string email);
    }
}

```

Рис. 2.2.2.1 - Реалізація конкретного інтерфейсу

```

using Microsoft.AspNetCore.Identity;

namespace SlayLib.Models
{
    19 references
    public class ApplicationUser : IdentityUser
    {
        1 reference
        public string FirstName { get; set; }
        1 reference
        public string LastName { get; set; }
        0 references
        public string Email { get; set; } = string.Empty;
    }
}

```

Рис. 2.2.2.2 - Оновлення класу ApplicationUser

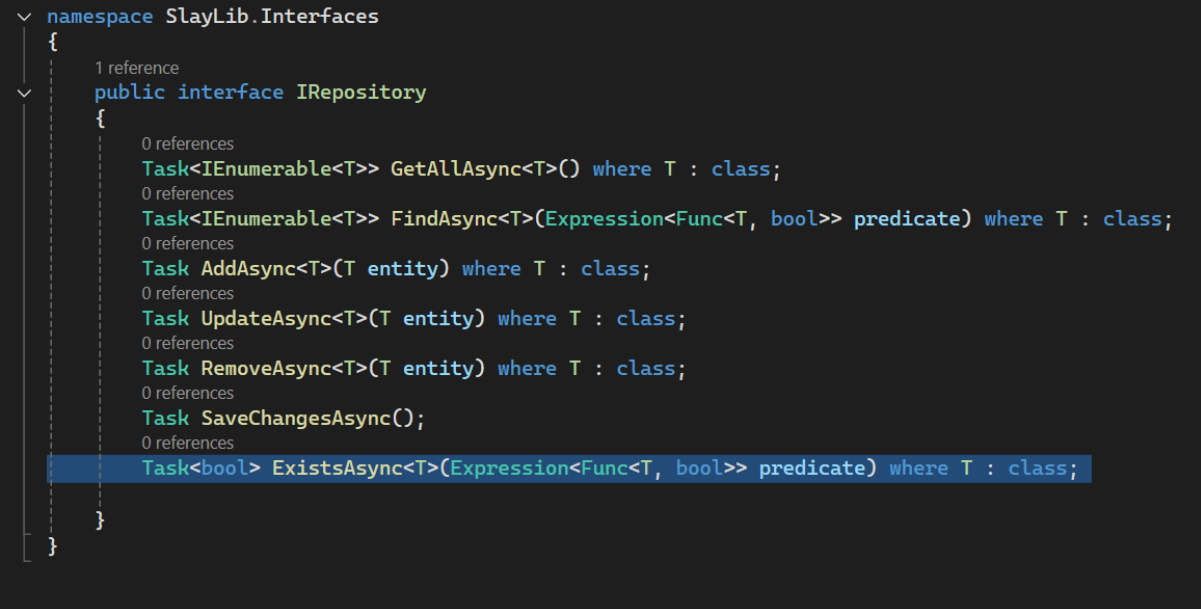
2.3 Додавання базового методу для перевірки існування сутності за умовою

На попередньому кроці ми створили базовий інтерфейс IRepository, який вже містить основні методи для CRUD-операцій.

Наступним кроком потрібно розширити цей інтерфейс методом, що дозволяє перевіряти існування сутності в базі даних за певною умовою.

Такий метод є важливим, тому що у реальних сценаріях часто потрібно перевірити, чи існує запис у таблиці (наприклад, користувач із певним email), перш ніж виконувати подальші дії (додавання, оновлення чи видалення)

Відкриваємо файл з створеним інтерфейсом і у кінець додаємо новий метод ExistsAsync()



```
namespace SlayLib.Interfaces
{
    1 reference
    public interface IRepository
    {
        0 references
        Task<IEnumerable<T>> GetAllAsync<T>() where T : class;
        0 references
        Task<IEnumerable<T>> FindAsync<T>(Expression<Func<T, bool>> predicate) where T : class;
        0 references
        Task AddAsync<T>(T entity) where T : class;
        0 references
        Task UpdateAsync<T>(T entity) where T : class;
        0 references
        Task RemoveAsync<T>(T entity) where T : class;
        0 references
        Task SaveChangesAsync();
        0 references
        Task<bool> ExistsAsync<T>(Expression<Func<T, bool>> predicate) where T : class;
    }
}
```

Рис. 2.3 - Додавання методу ExistsAsync()

Метод ExistsAsync() потрібен для того, щоб швидко перевірити, чи існує певний запис у базі даних, не завантажуючи його повністю.

Він повертає лише true або false, тому працює швидко й ефективно.

Це корисно, коли потрібно:

- перевірити, чи вже існує користувач з таким email перед реєстрацією;

- переконатись, що об'єкт справді є в базі перед видаленням або оновленням;
- уникнути помилок і зайвих запитів при роботі з даними.

Завдяки цьому методу код стає чистішим, швидшим і безпечнішим у плані перевірок

2.4 Реалізація базового класу репозиторію

Створюємо нову папку в бібліотеці даних - назовемо її Repositories. В ній створюємо клас BaseSqlServerRepository

Він буде реалізовувати всі методи, оголошені в інтерфейсі IRepository. Цей клас стане основою для всіх інших конкретних репозиторіїв (наприклад, для користувачів, замовлень тощо)

Його основна мета - інкапсулювати всю роботу з базою даних, щоб інші частини програми (контролери, сервіси) не взаємодіяли безпосередньо з DbContext, а працювали через готові методи доступу до даних. Таким чином, ми відокремлюємо бізнес-логіку від інфраструктурного шару, як вимагає патерн Repository.

```
using Microsoft.EntityFrameworkCore;
using SlayLib.Interfaces;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Linq.Expressions;
using System.Threading.Tasks;

namespace SlayLib.Repositories
{
    [reference]
    public class BaseSqlServerRepository<TDbContext> : IRepository
        where TDbContext : DbContext
    {
        protected readonly TDbContext Db;

        [reference]
        public BaseSqlServerRepository(TDbContext db)
        {
            Db = db;
        }

        [reference]
        public async Task<IEnumerable<T>> GetAllAsync<T>() where T : class
        {
            return await Db.Set<T>().AsNoTracking().ToListAsync();
        }

        [reference]
        public async Task<IEnumerable<T>> FindAsync<T>(Expression<Func<T, bool>> predicate) where T : class
        {
            return await Db.Set<T>().Where(predicate).AsNoTracking().ToListAsync();
        }

        [reference]
        public async Task AddAsync<T>(T entity) where T : class
        {
            await Db.Set<T>().AddAsync(entity);
            await SaveChangesAsync();
        }

        [reference]
        public async Task UpdateAsync<T>(T entity) where T : class
        {
            Db.Set<T>().Update(entity);
            await SaveChangesAsync();
        }

        [reference]
        public async Task RemoveAsync<T>(T entity) where T : class
        {
            Db.Set<T>().Remove(entity);
            await SaveChangesAsync();
        }

        [reference]
        public async Task SaveChangesAsync()
        {
            await Db.SaveChangesAsync();
        }

        [reference]
        public async Task<bool> ExistsAsync<T>(Expression<Func<T, bool>> predicate) where T : class
        {
            return await Db.Set<T>().AnyAsync(predicate);
        }
    }
}
```

Рис. 2.4 - Реалізація класу репозиторію

2.5 Створення конкретного репозиторію для веб-застосунку

Розширюємо функціональність репозиторію методом, який дозволяє виконувати пошук користувача за унікальною властивістю - email.

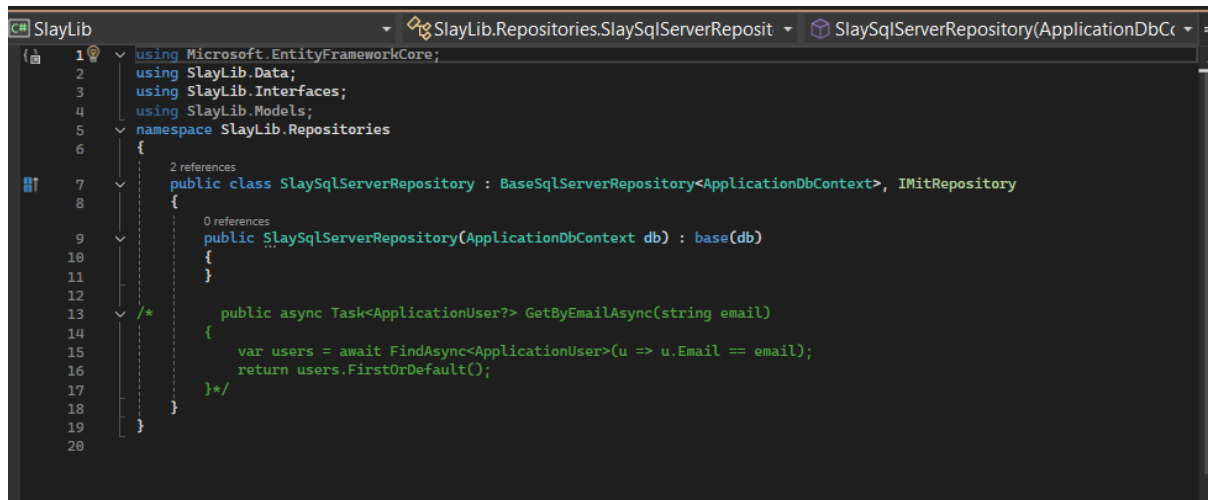


Рис 2.5.1 – Створення конкретного репозиторію та додавання методу пошуку користувача за email

2.6 Реєстрація залежностей (інтерфейсу репозиторію та його реалізації) у контейнері впровадження залежностей.

Після створення інтерфейсу та його реалізації необхідно зареєструвати цю пару в контейнері залежностей. Це дає змогу працювати з абстракцією, а не з конкретною реалізацією, що відповідає принципу інверсії залежностей. Такий підхід підвищує гнучкість та тестованість застосунку.

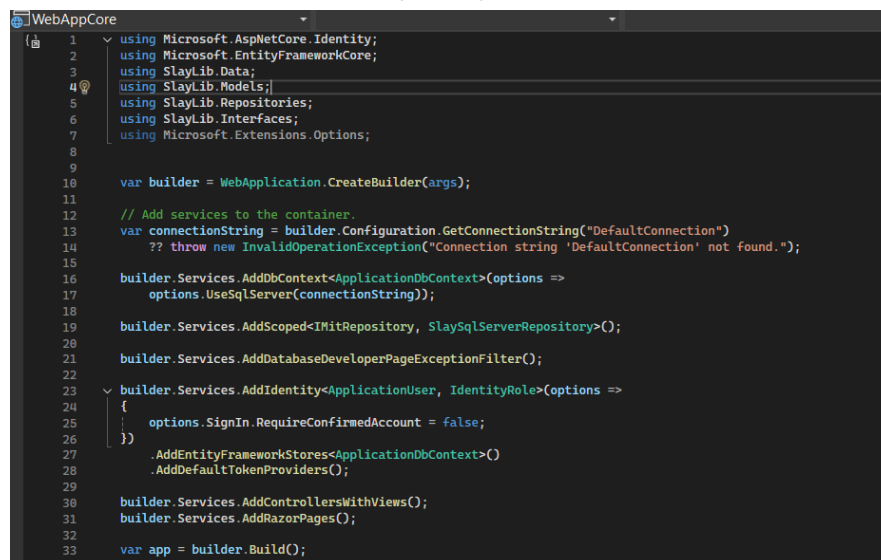
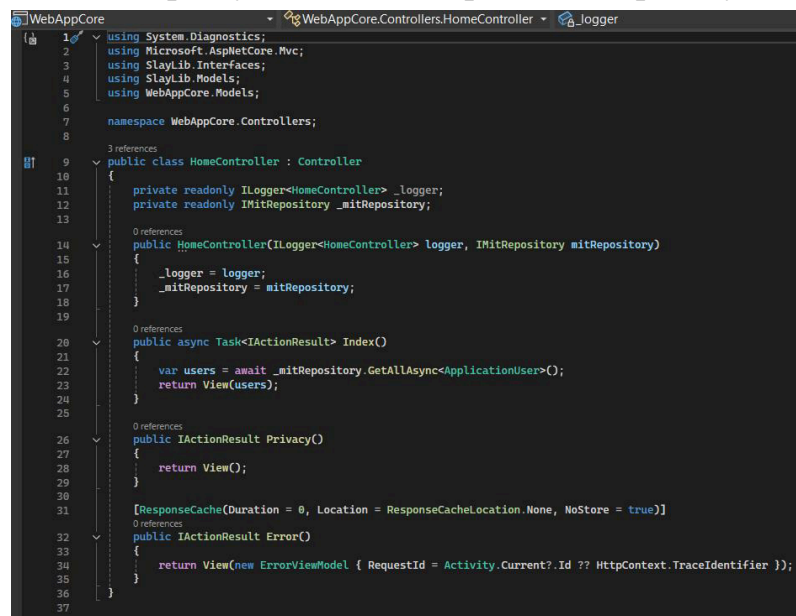


Рис 2.6.1 - Реєстрація інтерфейсу та реалізації в контейнері залежностей

2.7 Інтегрування репозиторію у контролер: реалізування методу, який отримує дані, використовуючи репозиторій.

Після створення і реєстрації репозиторію його можна використовувати у контролерах. У прикладі (рис. 2.7.1) контролер HomeController отримує екземпляр IMitRepository через механізм Dependency Injection. Таким чином, контролер працює з абстракцією, а не з конкретним класом, що ізолює бізнес-логіку від реалізації доступу до даних.

У методі Index() викликається метод All(), який повертає список користувачів. Це дозволяє отримати дані для відображення без прямої взаємодії з базою даних. Такий підхід підсилює розділення відповідальностей і спрощує подальший розвиток проекту.



```
1 using System.Diagnostics;
2 using Microsoft.AspNetCore.Mvc;
3 using SlayLib.Interfaces;
4 using SlayLib.Models;
5 using WebAppCore.Models;
6
7 namespace WebAppCore.Controllers;
8
9 public class HomeController : Controller
10 {
11     private readonly ILogger<HomeController> _logger;
12     private readonly IMitRepository _mitRepository;
13
14     public HomeController(ILogger<HomeController> logger, IMitRepository mitRepository)
15     {
16         _logger = logger;
17         _mitRepository = mitRepository;
18     }
19
20     public async Task<IActionResult> Index()
21     {
22         var users = await _mitRepository.GetAllAsync<ApplicationUser>();
23         return View(users);
24     }
25
26     public IActionResult Privacy()
27     {
28         return View();
29     }
30
31     [ResponseCache(Duration = 0, Location = ResponseCacheLocation.None, NoStore = true)]
32     public IActionResult Error()
33     {
34         return View(new ErrorViewModel { RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier });
35     }
36 }
37
```

Рис 2.7.1 - Використання репозиторію у контролері

2.8. Фіксування змін у проєкті на GitHub.

Після завершення роботи всі зміни з гілок команди об'єднуються в основну гілку main. Після цього лабораторна робота вважається завершеною та готовою до здачі.

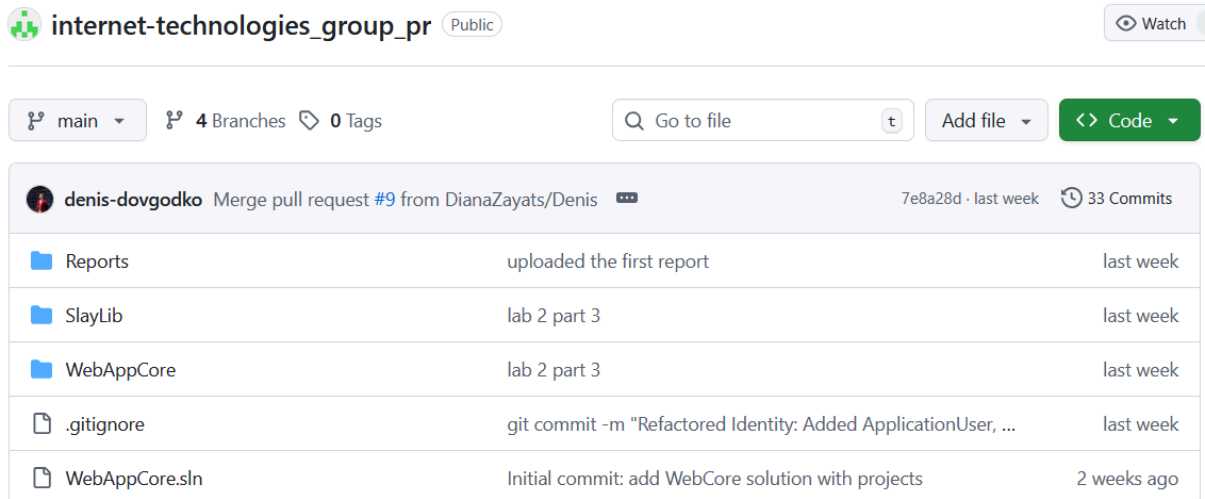


Рис 2.8.1 – Фіксування змін проєкту на Github в гілці main

Висновок:

У цій лабораторній роботі реалізовано шаблон Repository у веб-застосунку ASP.NET Core. Створено інтерфейси, конкретні класи репозиторіїв та налаштовано Dependency Injection для виконання CRUD-операцій. Така архітектура забезпечує чітке відокремлення бізнес-логіки від рівня доступу до даних, що підвищує гнучкість, тестованість і масштабованість системи.

Використання принципу інверсії залежностей (DIP) дозволяє контролерам працювати з абстракціями, дотримуючись принципів SOLID. Реалізація патерну Repository сприяла формуванню чистої архітектури та полегшила підтримку і розвиток застосунку.