

**МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ**  
**КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМ. ТАРАСА ШЕВЧЕНКА**  
**ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ**

**Кафедра мережевих та інтернет технологій**

**СУЧАСНІ ІНТЕРНЕТ ТЕХНОЛОГІЇ**

**РОБОТА З ДАНИМИ В ASP.NET CORE.**

**РЕАЛІЗАЦІЯ ШАБЛОНУ REPOSITORY**

**Лабораторне заняття № 2**

**Черкун Єви Сергіївни**

**Хід виконання роботи:**

2.1 Аналіз можливої проблеми подвійного відстеження при оновленні даних.

Проблема подвійного відстеження (Double Tracking) виникає, коли Entity Framework Core одночасно відстежує дві копії однієї й тієї ж сутності з однаковим ключем. У такому випадку контекст (DbContext) не може визначити, яку версію сутності потрібно оновити, і видає помилку типу *“entity cannot be tracked because another instance with the same key is already being tracked”*.

Це часто трапляється, коли сутність повторно завантажують з бази або створюють новий об’єкт із тим самим ідентифікатором. Щоб уникнути проблеми, для операцій читання варто використовувати AsNoTracking(), а під час оновлення — працювати лише з одним екземпляром сутності або встановлювати її стан як Modified.

Таким чином забезпечується коректне оновлення даних без конфліктів у контексті.

2.2 Створення базового та конкретного інтерфейсів репозиторію

Створюємо папку Interfaces у бібліотеці даних.

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

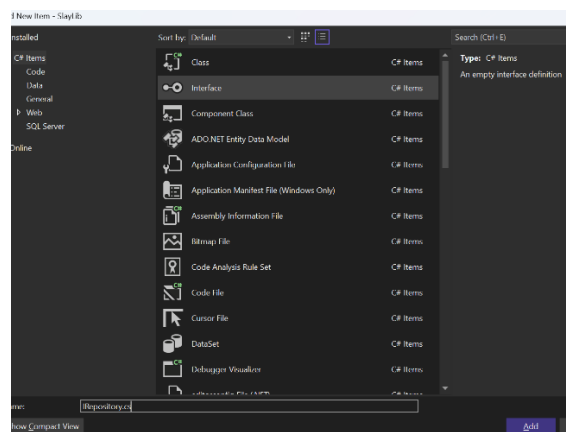


Рис. 2.2.1 - Створення базового інтерфейсу

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. 2.2.2 - Реалізація операцій для роботи з даними

Конкретний інтерфейс створюємо аналогічно до попереднього. 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. 2.2.3 - 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;
    }
}
```

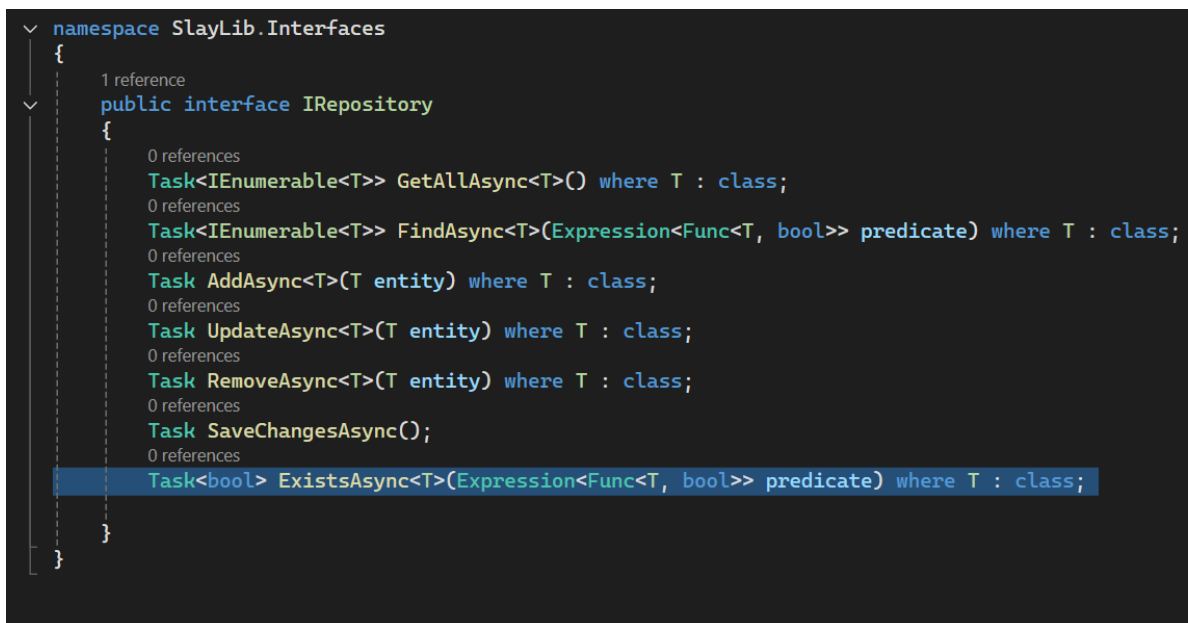
Рис. 2.2.4 - Оновлення класу ApplicationUser

### 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.

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. 2.3.1 - Додавання методу ExistsAsync()

Метод ExistsAsync() перевіряє, чи існує у базі даних сутність, що відповідає певній умові, і повертає true або false.

Він корисний, коли потрібно швидко дізнатися про наявність запису без завантаження всіх даних.

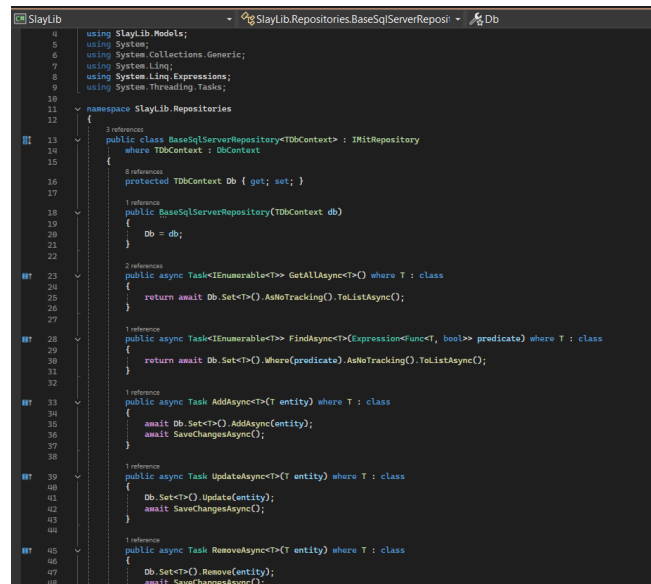
Найчастіше застосовується перед створенням нового запису, щоб уникнути дублювання (наприклад, перевірка, чи існує користувач з таким email), або перед оновленням чи видаленням, щоб переконатися, що сутність є в базі. Таким чином, ExistsAsync() робить роботу з даними ефективнішою та безпечнішою.

### 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 SlayLib.Models;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Linq.Expressions;
using System.Threading.Tasks;

namespace SlayLib.Repositories
{
    public class BaseSqlServerRepository<T> : IMitRepository
        where T : class
    {
        protected DbContext Db { get; set; }

        public BaseSqlServerRepository(DbContext db)
        {
            Db = db;
        }

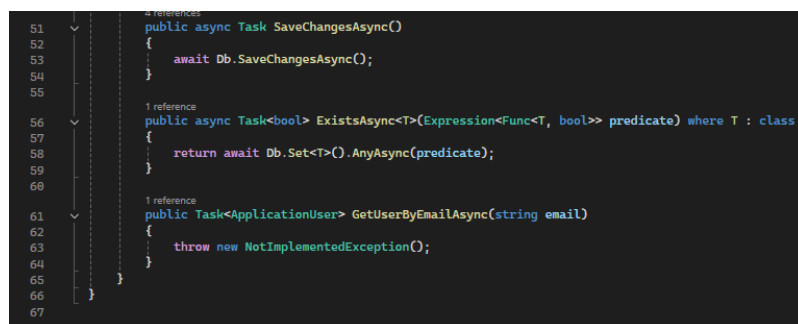
        public async Task<IEnumerable<T>> GetAllAsync() where T : class
        {
            return await Db.Set<T>().AsNoTracking().ToListAsync();
        }

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

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

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

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



```
public async Task SaveChangesAsync()
{
    await Db.SaveChangesAsync();
}

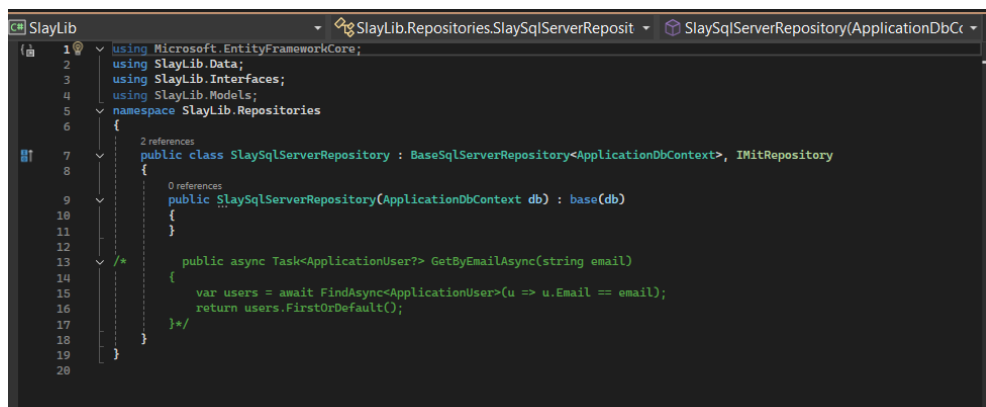
public async Task<bool> ExistsAsync<T>(Expression<Func<T, bool>> predicate) where T : class
{
    return await Db.Set<T>().AnyAsync(predicate);
}

public Task<ApplicationUser> GetUserByEmailAsync(string email)
{
    throw new NotImplementedException();
}
```

Рис 2.4.1 – Реалізація базового класу репозиторію

## 2.5 Створення конкретного репозиторію для веб-застосунку

Розширюємо його функціональність методом, що виконує пошук користувача за унікальною властивістю (email):



```
using Microsoft.EntityFrameworkCore;
using SlayLib.Data;
using SlayLib.Interfaces;
using SlayLib.Models;

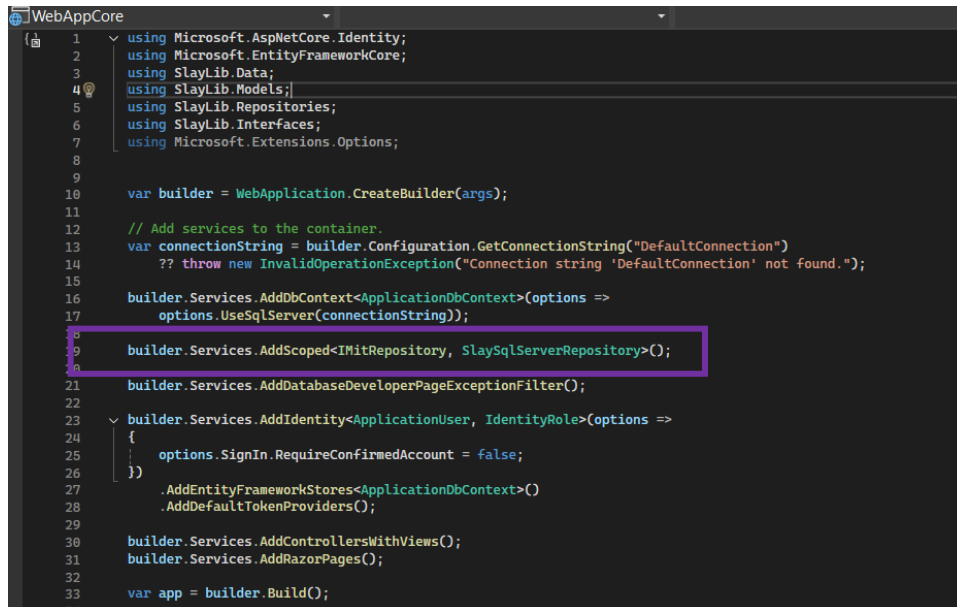
namespace SlayLib.Repositories
{
    public class SlaySqlServerRepository : BaseSqlServerRepository<ApplicationDbContext>, IMitRepository
    {
        public SlaySqlServerRepository(ApplicationDbContext db) : base(db)
        {
        }

        public async Task<ApplicationUser> GetUserByEmailAsync(string email)
        {
            var users = await FindAsync<ApplicationUser>(u => u.Email == email);
            return users.FirstOrDefault();
        }
    }
}
```

Рис 2.5.1 – Створення конкретного репозиторію та його розширення

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

Після створення інтерфейсу репозиторію та його конкретної реалізації наступним кроком є реєстрація цієї залежності у контейнері впровадження залежностей. Це дозволяє бізнес-логіці працювати не з конкретним класом, а з абстракцією, що відповідає принципу інверсії залежностей.



```
1 using Microsoft.AspNetCore.Identity;
2 using Microsoft.EntityFrameworkCore;
3 using SlayLib.Data;
4 using SlayLib.Models;
5 using SlayLib.Repositories;
6 using SlayLib.Interfaces;
7 using Microsoft.Extensions.Options;
8
9
10 var builder = WebApplication.CreateBuilder(args);
11
12 // Add services to the container.
13 var connectionString = builder.Configuration.GetConnectionString("DefaultConnection")
14     ?? throw new InvalidOperationException("Connection string 'DefaultConnection' not found.");
15
16 builder.Services.AddDbContext<ApplicationDbContext>(options =>
17     options.UseSqlServer(connectionString));
18 builder.Services.AddScoped<IMitRepository, SlaySqlServerRepository>();
19
20 builder.Services.AddDatabaseDeveloperPageExceptionFilter();
21
22 builder.Services.AddIdentity<ApplicationUser, IdentityRole>(options =>
23 {
24     options.SignIn.RequireConfirmedAccount = false;
25 })
26     .AddEntityFrameworkStores<ApplicationDbContext>()
27     .AddDefaultTokenProviders();
28
29 builder.Services.AddControllersWithViews();
30 builder.Services.AddRazorPages();
31
32 var app = builder.Build();
33
```

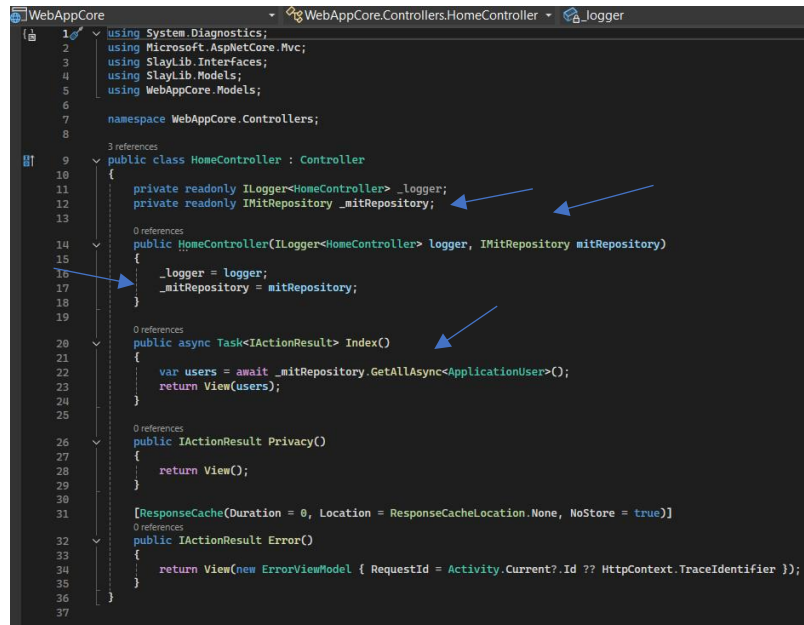
Рис 2.6.1 - Реєстрація залежності (інтерфейсу та його реалізації) у контейнері впровадження залежностей

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

Після створення інтерфейсу, реалізації та реєстрації репозиторію в контейнері залежностей, наступним кроком є його використання у прикладному коді. У прикладі нижче (рис. 2.7.1) HomeController в ASP.NET Core отримує абстракцію репозиторію IMitRepository через DI.

Завдяки цьому контролер не залежить від конкретної реалізації доступу до даних і працює лише з інтерфейсом. У методі Index викликається метод All(), який повертає колекцію користувачів. Контролер отримує дані для відображення, залишаючись ізольованим від роботи з базою даних. Це забезпечує ізоляцію бізнес-логіки від деталей збереження даних — головну перевагу цього підходу.

Таким чином, патерн Repository дозволяє зосередити бізнес-логіку на обробці запитів, а технічні аспекти доступу до даних залишити в інфраструктурному шарі.



```
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.

Мерджимо зміни з усіх гілок команди в main і лабораторна, відповідно, вважається готовою до здачі.

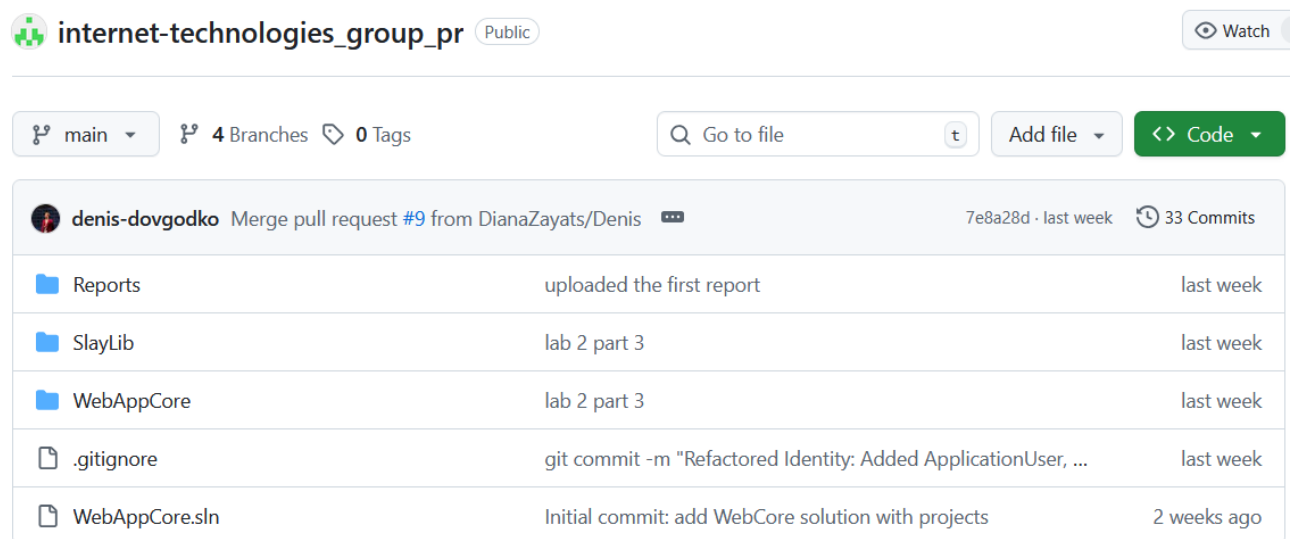


Рис 2.8.1 – Фіксування змін проєкту на Github в гілці main

Висновок:

У лабораторній роботі реалізовано шаблон Repository у веб-застосунку ASP.NET Core. Створено інтерфейси, класи репозиторіїв і налаштовано Dependency Injection для виконання CRUD-операцій. Такий підхід забезпечив відокремлення бізнес-логіки від доступу до даних, підвищив гнучкість, тестованість і масштабованість застосунку. Завдяки інверсії залежностей (DIP) контролери працюють із абстракціями, що відповідає принципам SOLID.

Впровадження патерну Repository сприяло створенню чистої архітектури та спрощенню підтримки проєкту