Лабораторна робота №2

Звіт

З дисципліни “Сучасні інтернет технології”

на тему: “Робота з даними в asp.net core. реалізація шаблону repository”.

Студента 4 курсу: Групи МІТ-41
Демиденко Андрій

Київ - 2025р.

Хід виконання роботи

Тема: Робота з даними в asp.net core. реалізація шаблону repository.
Мета: Набути практичних навичок зі створення вебзастосунків ASP.NET Core з використанням архітектурного шаблону MVC, реалізувати автентифікацію, розширити стандартні моделі даних, виконати рефакторинг проєкту шляхом поділу моноліту на окремі логічні шари, а також опанувати механізм міграцій Entity Framework Core та принципи роботи з репозиторіями.

Створення базового та конкретного репозиторію

Створення інтерфейсу репозиторію як шару абстракції. У модулі, що зосереджує роботу з даними, у папці Interfaces необхідно створити інтерфейс IRepository та описати сигнатури методів для базової взаємодії з даними (рис.1**)**.

```
using System.Linq.Expressions;

namespace WebApplication.Data.Interfaces
{
    2 references
    public interface IRepository
    {
        5 references
        IQueryable<T> All<T>() where T : class;

        5 references
        IQueryable<T> ReadAll<T>() where T : class;

        1 reference
        IQueryable<T> ReadWhere<T>(Expression<Func<T, bool>> expression) where T : class;

        1 reference
        Task<T?> FirstOrDefaultAsync<T>(Expression<Func<T, bool>> expression) where T : class;

        1 reference
        Task<T?> SingleAsync<T>(Expression<Func<T, bool>> expression) where T : class;

        1 reference
        Task<T?> ReadSingleAsync<T>(Expression<Func<T, bool>> expression) where T : class;

        1 reference
        Task<int> AddAsync<T>(T item) where T : class;

        1 reference
        Task<int> UpdateAsync<T>(T item) where T : class;

        1 reference
        Task<int> RemoveAsync<T>(T item) where T : class;

        1 reference
        Task<bool> ExistsAsync<T>(Expression<Func<T, bool>> expression) where T : class;
    }
}
```

Рис. 1 – Створення інтерфейсу репозиторію як шару абстракції.

Створення інтерфейсу конкретного репозиторію ASP.NET Core. Такий інтерфейс має успадковувати всі методи базового інтерфейсу IRepository та може також включати специфічні методи маніпулацій даними, що притаманні цьому веб застосунку (рис. 2).

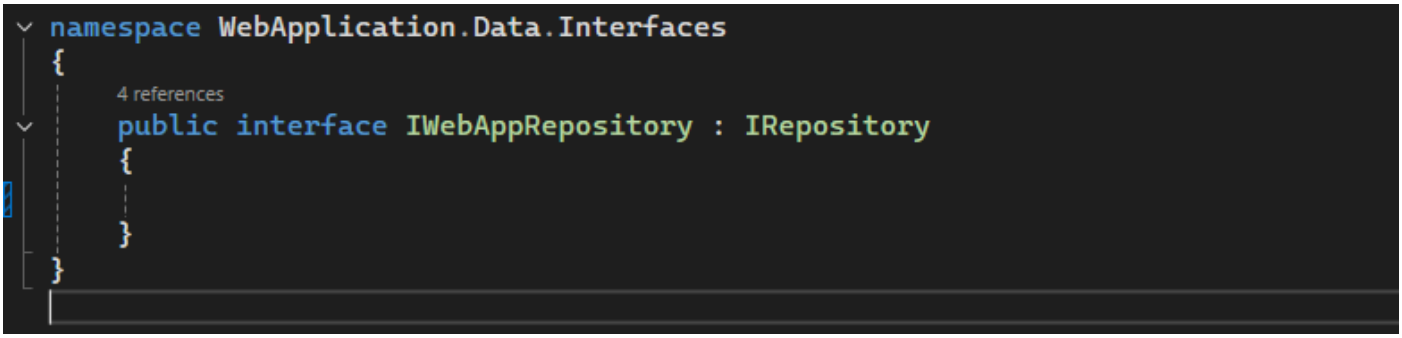

Рис. 2 – Створення інтерфейсу конкретного репозиторію ASP.NET Core.

Додавання базового методу

Додати базовий метод для перевірки існування сутності за умовою **ExistsAsync()** (рис. 3)**.**

Рис. 3 – ExistsAsync().

Цей метод може використовуватися для наступних цілей:

1. Реєстрація користувача: Перед створенням нового користувача перевірити: "Чи існує вже користувач з такою електронною поштою?"
2. Унікальні імена: Перед тим, як дозволити користувачу створити "Категорію" або "Тег", перевірити: "Чи існує вже запис з такою назвою?"
3. Перевірка ID: Перед тим, як додати товар у кошик, перевірити: "Чи існує взагалі товар з ID, який надіслав клієнт?"

Реалізація базового класу репозиторію

Розглянемо базовий репозиторій, який інкапсулює роботу з Entity Framework Core і надає універсальні методи для виконання CRUD-операцій. Кожен метод реалізує окремий аспект взаємодії з даними: додавання нових сутностей, отримання всіх записів або вибірки за умовою, читання без відстеження, оновлення та видалення. Такий підхід дозволяє винести логіку доступу до бази даних в окремий шар і зробити бізнес-код незалежним від деталей реалізації.
При імплементації важливо дотримуватися кількох принципів. Методи репозиторію мають залишатися максимально простими й відповідати лише за роботу з даними, не змішуючи бізнес-правила. Для операцій читання доцільно використовувати AsNoTracking, щоб зменшити навантаження на контекст. Для оновлення варто контролювати стан сутностей, щоб уникати конфліктів відстеження. Також рекомендується підтримувати асинхронність у всіх методах, що взаємодіють із базою даних, і передбачати можливість розширення базового класу спеціалізованими репозиторіями для складніших сценаріїв. Така структура робить код більш гнучким, тестованим і відповідним принципам чистої архітектури (рис. 4 - 5).

```
using Microsoft.EntityFrameworkCore;
using System.Linq.Expressions;
using WebApplication.Data.Interfaces;

namespace WebApplication.Data.Repositories
{
    3 references
    public class BaseSqlServerRepository<TDbContext> : IRepository
        where TDbContext : DbContext
    {
        13 references
        protected TDbContext Db { get; set; }

        1 reference
        public BaseSqlServerRepository(TDbContext db)
        {
            Db = db;
        }
        1 reference
        public async Task<int> AddAsync<T>(T item) where T : class
        {
            Db.Entry(item).State = EntityState.Detached;
            await Db.AddAsync(item);
            return await Db.SaveChangesAsync();
        }

        5 references
        public IQueryable<T> All<T>() where T : class
        {
            return Db.Model.FindEntityType(typeof(T)) != null
                ? Db.Set<T>().AsQueryable()
                : new List<T>().AsQueryable();
        }
public async Task<T?> FirstOrDefaultAsync<T>(Expression<Func<T, bool>> expression) where T : class
{
    return await All<T>().FirstOrDefaultAsync(expression);
}

5 references
public IQueryable<T> ReadAll<T>() where T : class
{
    return All<T>().AsNoTracking();
}

1 reference
public async Task<T?> ReadSingleAsync<T>(Expression<Func<T, bool>> expression) where T : class
{
    return await ReadAll<T>().SingleOrDefaultAsync(expression);
}

1 reference
public IQueryable<T> ReadWhere<T>(Expression<Func<T, bool>> expression) where T : class
{
    return ReadAll<T>().Where(expression);
}

1 reference
public async Task<int> RemoveAsync<T>(T item) where T : class
{
    Db.Remove(item);
    return await Db.SaveChangesAsync();
}

1 reference
public async Task<T?> SingleAsync<T>(Expression<Func<T, bool>> expression) where T : class
{
    return await All<T>().SingleOrDefaultAsync(expression);
}
```

Рис. 4 – AddAsync(), All(), FirstOrDefaultAsync(), ReadAll(),ReadSingleAsync(), ReadWhere(), RemoveAsync(), SingleAsync().

```
1 reference
public async Task<int> UpdateAsync<T>(T item) where T : class
{
    var local = Db.Set<T>()
        .Local
        .FirstOrDefault(entry => entry.Equals(item));

    if (local != null)
    {
        Db.Entry(local).State = EntityState.Detached;
    }

    Db.Entry(item).State = EntityState.Modified;
    Db.Update(item);
    return await Db.SaveChangesAsync();
}

1 reference
public async Task<bool> ExistsAsync<T>(Expression<Func<T, bool>> expression) where T : class
{
    return await ReadAll<T>().AnyAsync(expression);
}
```

Рис. 5 – ExistsAsync(), UpdateAsync().

Створення конкретного класу репозиторію для веб-застосунку

Створення конкретного класу репозиторію для веб-застосунку та розширення його функціональність методом, що виконує пошук користувача за унікальною властивістю (рис. 6 - 7).

```
using WebApplication.Data.Models;

namespace WebApplication.Data.Interfaces
{
    4 references
    public interface IWebAppRepository : IRepository
    {
        1 reference
        Task<ApplicationUser?> GetUserByEmailAsync(string email);
    }
```

Рис. 6 – IWebAppRepository.cs.

```
namespace WebApplication.Data.Repositories
{

    2 references
    public class WebAppRepository : BaseSqlServerRepository<ApplicationDbContext>, IWebAppRepository
    {

        0 references
        public WebAppRepository(ApplicationDbContext db) : base(db)
        {

        }

        1 reference
        public async Task<ApplicationUser?> GetUserByEmailAsync(string email)
        {

            return await ReadAll<ApplicationUser>()
                .FirstOrDefaultAsync(u => u.Email == email);
        }
```

Рис. 7 – WebAppRepository.cs.

Реєстрація залежностей в контейнері

Після створення інтерфейсу репозиторію та його конкретної реалізації наступним кроком є реєстрація цієї залежності у контейнері впровадження залежностей (рис. 8). Це дозволяє бізнес-логіці працювати не з конкретним класом, а з абстракцією, що відповідає принципу інверсії залежностей.

Таким чином ми повідомляємо контейнеру, що кожного разу, коли застосунку знадобиться об’єкт типу IMitRepository, він має створити екземпляр класу MitSqlServerRepository з життєвим циклом Scoped. Це означає, що в межах одного HTTP-запиту всі залежності цього типу будуть спільними, а для нового запиту створюватиметься новий екземпляр.

Такий підхід забезпечує правильне керування життєвим циклом репозиторію, ізоляцію бізнес-логіки від інфраструктури та відкриває можливість легко замінювати реалізацію (наприклад, на InMemoryRepository для тестування) без змін у коді контролерів.

```
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using WebApplication.Data.Data;
using WebApplication.Data.Interfaces;
using WebApplication.Data.Repositories;

var builder = Microsoft.AspNetCore.Builder.WebApplication.CreateBuilder(args);

// Add services to the container.
var connectionString = builder.Configuration.GetConnectionString("DefaultConnection") ?? throw new InvalidOperationException("Con
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString, b => b.MigrationsAssembly("WebApplication")));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(options => options.SignIn.RequireConfirmedAccount = true)
    .AddEntityFrameworkStores<ApplicationDbContext>();
builder.Services.AddControllersWithViews();

builder.Services.AddScoped<IWebAppRepository, WebAppRepository>();
```

Рис. 8 – Реєстрація залежностей в контейнері.

Інтеграція репозиторію у контролер

Наступним кроком після створення інтерфейсу репозиторію, його реалізації та реєстрації у контейнері залежностей є безпосереднє використання цього репозиторію у прикладному коді. Зазвичай це відбувається у сервісах або контролерах, які отримують абстракцію репозиторію через механізм Dependency Injection (рис. 9).

Контролер HomeController отримує залежність через механізм Dependency Injection: абстракцію репозиторію IMitRepository. Завдяки цьому контролер не створює об’єкти самостійно й не залежить від конкретної реалізації доступу до даних – він працює лише з інтерфейсом. У методі Index виконується звернення до репозиторію через метод All(), який повертає колекцію користувачів. Таким чином, контролер отримує необхідні дані для подальшої передачі у відображення, залишаючись ізольованим від деталей роботи з базою даних.

Це демонструє ключову перевагу патерну Repository: бізнес-логіка контролера зосереджена на обробці запиту та формуванні відповіді, тоді як усі технічні аспекти доступу до даних інкапсульовані в інфраструктурному шарі реалізації репозиторію.

Такий підхід дозволяє бізнес-логіці працювати виключно з інтерфейсом, не знаючи деталей реалізації доступу до даних. У результаті контролер або сервіс може виконувати необхідні дії – наприклад, отримувати список користувачів чи зберігати новий об’єкт – використовуючи методи репозиторію, а вся робота з базою даних залишається прихованою в інфраструктурному шарі.

```
using WebApplication.Data.Interfaces;

namespace WebApplication.Controllers
{
    1 reference
    public class HomeController : Controller
    {

        private readonly IWebAppRepository _repository;

        0 references
        public HomeController(IWebAppRepository repository)
        {
            _repository = repository;
        }

        0 references
        public async Task<IActionResult> Index()
        {
            var users = _repository.All<ApplicationUser>();

            return View(users);
        }
```

Рис. 9 – Інтеграція репозиторію у контролер.

Висновок:

У ході виконання роботи було розглянуто та реалізовано шаблон Repository як один із ключових підходів до організації доступу до даних в ASP.NET Core. Спочатку було створено базовий інтерфейс репозиторію, який виконує роль шару абстракції та визначає основний набір операцій для роботи з даними. Далі був реалізований конкретний інтерфейс репозиторію для вебзастосунку, що розширює базову функціональність специфічними методами. Було розроблено базовий клас репозиторію, який інкапсулює логіку взаємодії з Entity Framework Core і надає універсальні CRUD-операції, включаючи оптимізацію шляхом використання AsNoTracking та повну підтримку асинхронності.

На основі базової реалізації створено конкретний репозиторій для застосунку, що дозволяє виконувати цільові операції з користувачами та іншими сутностями. Далі реалізовано процес реєстрації залежностей у контейнері DI, що забезпечує інверсію залежностей та дозволяє працювати із застосунком через абстракції, а не конкретні реалізації. Завдяки життєвому циклу Scoped репозиторії створюються окремо для кожного HTTP-запиту, що гарантує коректність роботи з контекстом даних.

На завершальному етапі репозиторій було інтегровано в контролер. Це продемонструвало ключові переваги архітектури: контролер не залежить від конкретної

реалізації доступу до даних, а взаємодіє лише з інтерфейсом, що робить код гнучким, тестованим і простим у розширенні. Використання репозиторію дозволило ізолювати бізнес-логіку від інфраструктурних деталей і забезпечити чисту архітектуру проєкту.
Таким чином, у ході роботи було досягнуто поставленої мети: опановано роботу з даними в ASP.NET Core, реалізовано шаблон Repository, організовано багаторівневу структуру застосунку, впроваджено механізми DI та міграцій, що формує ґрунтовні навички побудови надійних і масштабованих вебзастосунків.