МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ ВИЩИЙ НАВЧАЛЬНИЙ ЗАКЛАД
“ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ”
Кафедра прикладної математики та інформатики

Методичні вказівки до виконання розрахункової роботи з дисципліни “ Крос-платформне програмування“

для бакалаврів, що навчаються за спеціальностями 121 «Інженерія програмного забезпечення», 122 «Комп’ютерні науки»,125 «Кібербезпека», усіх форм навчання

ДРОГОБИЧ-2025

УДК 004.42(072)

М 54

Методичні вказівки до виконання розрахункової роботи з дисципліни «Крос-платформне програмування» для бакалаврів, що навчаються за спеціальностями 121 «Інженерія програмного забезпечення», 122 «Комп’ютерні науки»,125 «Кібербезпека» усіх форм навчання / уклад. М.О. Александров. – Дрогобич : ДонНТУ, 2025. – 28 с.

Наведено методичні рекомендації, контрольні питання й завдання для виконання індивідуальної розрахункової роботи.

Укладач: Микита АЛЕКСАНДРОВ, .ф., доцент кафедри ПМІ

Рецензент: Сергій КОВАЛЬОВ, к.т.н., доц., зав. каф. ЕТ і КІ

Відповідальний за випуск: Ярослав ДОРОГИЙ, д.т.н., проф., зав. каф. ПМІ

Затверджено навчально-методичним відділом ДВНЗ ДонНТУ,
протокол №__ від __.__.20__

Розглянуто на засіданні кафедри ПМІ, протокол №2 від 21.02.2025р.

ЗМІСТ

ВСТУП	4
Завдання	5
Варіанти завдань до розрахункової роботи	6
Приклад виконання для варіанту Х	8
Питання для самоконтролю	17
СПИСОК РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ	18
Додаток А. Лістинг основних файлів. REST API ASP.NET Core.	19
Додаток Б. Лістинг основних файлів веб-сервісу в Blazor	22

ВСТУП

Курс "Кросплатформне програмування" є невід'ємною частиною підготовки фахівців у галузі програмної інженерії, комп'ютерних наук та кібербезпеки. У сучасному світі розробка програмного забезпечення вимагає створення гнучких рішень, які можуть функціонувати на різних операційних системах і пристроях. Саме тому особливе значення набувають технології, що забезпечують крос-платформну розробку.

Метою виконання розрахункової роботи є набуття теоретичних знань та практичних навичок у створенні крос-платформних веб-додатків з використанням технологій ASP.NET Core та Blazor. Студенти зможуть розробити повноцінний RESTful API, реалізувати CRUD-операції та забезпечити інтеграцію веб-додатка для взаємодії з API через HTTP-запити. Це дозволить їм глибше зрозуміти принципи побудови сучасних веб-рішень та отримати досвід роботи з популярними інструментами та фреймворками.

У межах розрахункової роботи студенти повинні розробити веб-додаток, який буде взаємодіяти з базою даних через REST API. Важливо забезпечити належний рівень обробки помилок, валідацію введених даних та відповідне стилізування інтерфейсу. Крім того, студенти складуть звіт, у якому детально опишуть процес розробки, тестування та результати своєї роботи.

Методичні вказівки містять детальні інструкції щодо виконання розрахункової роботи, вимоги до реалізації проєкту, а також рекомендації щодо написання звіту. Вони спрямовані на допомогу студентам у засвоєнні основ крос-платформної розробки та набутті практичного досвіду у створенні сучасних веб-додатків.

Завдання

Розробити RESTful API засобами ASP.NET Core. Розробити веб-додаток, який буде взаємодіяти з API за допомогою REST методів (GET, POST, PUT, DELETE). Веб-додаток має відображати дані, отримані від API, а також дозволяти додавати, оновлювати та видаляти інформацію.

Загальні вимоги:

- API повинно бути реалізоване на основі ASP.NET Core.
- Для взаємодії з даними використовуйте базу даних (SQLite, SQL Server або PostgreSQL за вашим вибором).
- Веб-додаток має бути створений з використанням Blazor.
- Дані повинні бути пов’язані з конкретною тематикою (в залежності від варіанта завдання).
- Реалізуйте CRUD (Create, Read, Update, Delete) операції.
- Забезпечте відповідну обробку помилок та валідацію введених даних.
- Веб-додаток має бути стилізований засобами CSS.

Вимоги до звіту:

1. Мета, завдання та варіант.
2. Стислий опис предметної області та визначення архітектури бази даних.
3. Опис процесу проектування та розробки REST API включаючи опис створених класів і методів.
4. Опис процесу проектування та розробки веб-додатку включаючи опис створених класів і методів.
5. Опис процесу стилізації веб-додатку за допомогою CSS.
6. Опис процесу тестування роботи розробленого веб-додатку та REST API включаючи приклади HTTP-запитів до API.
7. Код основних файлів REST API та веб додатку.
8. Висновки.

9. Кожен скріншот повинен містити опис та підпис.

Варіанти завдань до розрахункової роботи

Таблиця 1 – Варіанти завдань до розрахункової роботи

В.	Опис завдання
1	**Управління персоналом.** API - Дані про співробітників (*приклад: ПІБ, посада, відділ, дата прийняття на роботу*). Фронтенд - Інтерфейс для перегляду списку співробітників, редагування інформації, пошуку за відділом.
2	**Система управління клієнтами.** API - Дані про клієнтів *(приклад: ім'я, прізвище, контактні дані, дата останньої взаємодії).* Фронтенд - Форма для перегляду, додавання, оновлення та видалення клієнтів.
3	**Управління замовленнями.** API - Список замовлень *(приклад: ідентифікатор замовлення, товар, кількість, ціна, статус).* Фронтенд - Панель для створення, оновлення та видалення замовлень, сортування за статусом.
4	**Система обліку витрат.** API - Дані про витрати *(приклад: категорія, сума, дата, опис).* Фронтенд - Інтерфейс для введення нових витрат, перегляду історії та пошуку за датою.
5	**Система управління товарами на складі.** API - Інформація про товари *(приклад: назва, кількість, постачальник, дата надходження).* Фронтенд - Форма для керування запасами, редагування кількості, фільтри за постачальником.
6	**Система бронювання місць.** API - Інформація про бронювання *(приклад: місце, клієнт, дата, статус).* Фронтенд - Панель для створення нових бронювань, редагування статусу та перегляду доступних місць.
7	**Управління студентами.** API - Дані про студентів *(приклад: ім'я, група, середній бал, спеціальність).* Фронтенд - Інтерфейс для перегляду списку студентів, додавання нових записів, редагування балів.

8	**Кінопрокат.** API - Дані про фільми *(приклад: назва, жанр, рік, прокатний статус).* Фронтенд - Форма для керування списком фільмів, пошуку за жанром, додавання нових.
9	**Система обліку задач.** API - Завдання *(приклад: назва, опис, дедлайн, статус виконання).* Фронтенд - Список задач із можливістю змінювати статус виконання, сортувати за дедлайном.
10	**Управління транспортом.** API - Дані про транспорт *(приклад: тип, модель, номер, статус обслуговування).* Фронтенд - Панель для додавання нових записів, редагування статусу, пошуку за типом транспорту.
X	**Система управління книгами.** API - Робота з базою даних книг (*приклад: назва, автор, рік видання, жанр).* Фронтенд - Візуалізація списку книг, можливість додавання, редагування, та видалення записів.

Приклад виконання для варіанту X

1. **Структура бази даних.** База даних складається з однієї таблиці Books:
   - Id (int) — первинний ключ.
   - Title (string) — назва книги.
   - Author (string) — автор книги.
   - Year (int) — рік видання.
   - Genre (string) — жанр.
2. **Розробка API**
   - Створіть проєкт ASP.NET Core Web API.
   - Налаштуйте базу даних за допомогою Entity Framework Core.

Entity Framework Core (EF Core) — це ORM (Object-Relational Mapping), що дозволяє працювати з базою даних через об'єкти C#. Ось покрокова інструкція налаштування бази даних для API:

1. ***Додайте пакети NuGet.*** Перейдіть до Package Manager Console або використовуйте NuGet Manager у Visual Studio та встановіть необхідні пакети:

```
dotnet add package Microsoft.EntityFrameworkCore
dotnet add package Microsoft.EntityFrameworkCore.Sqlite # Для SQLite
dotnet add package Microsoft.EntityFrameworkCore.SqlServer # Для SQL Server
dotnet add package Microsoft.EntityFrameworkCore.Tools
dotnet add package Microsoft.EntityFrameworkCore.Design
```

2. ***Створіть клас моделі даних.*** Клас моделі відповідає структурі таблиці в базі даних. Наприклад, для таблиці Books:

```
public class Book
{
    public int Id { get; set; } // Первинний ключ
    public string Title { get; set; } // Назва книги
    public string Author { get; set; } // Автор
    public int Year { get; set; } // Рік видання
    public string Genre { get; set; } // Жанр
}
```

3. ***Створіть клас контексту бази даних.*** Контекст бази даних — це клас, що визначає, як Entity Framework Core працює з базою даних. Створіть новий клас LibraryContext:

```
public class LibraryContext : DbContext
{
    public LibraryContext(DbContextOptions<LibraryContext> options) :
base(options) { }

    // Визначення таблиць у базі даних
    public DbSet<Book> Books { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        // Налаштування таблиці Books
        modelBuilder.Entity<Book>().ToTable("Books");
```

```
        modelBuilder.Entity<Book>().HasKey(b => b.Id); // Вказуємо первинний
ключ
        modelBuilder.Entity<Book>().Property(b => b.Title).IsRequired(); //
Поле Title є обов'язковим
        modelBuilder.Entity<Book>().Property(b => b.Author).IsRequired();
    }
}
```

4. ***Налаштуйте підключення до бази даних.*** У файлі appsettings.json додайте рядок підключення до бази даних. Наприклад, для SQLite:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "DefaultConnection": "Data Source=library.db"
  }
}
```

5. ***Налаштуйте контекст у Program.cs.*** Додайте контекст бази даних до контейнера служб у Program.cs:

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

// Додаємо сервіс для EF Core
builder.Services.AddDbContext<LibraryContext>(options =>

options.UseSqlite(builder.Configuration.GetConnectionString("DefaultConnectio
n")));
// Якщо використовуєте SQL Server:
//
options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnec
tion"))
```

6. ***Створіть і застосуйте міграції.*** Міграції дозволяють створювати або оновлювати структуру бази даних на основі класів моделей.

- У терміналі виконайте команду для створення міграції. Це створить директорію Migrations, у якій буде описана структура таблиці Books.

```
dotnet ef migrations add InitialCreate
```

- Застосуйте міграцію для створення бази даних. Після виконання цієї команди файл бази даних library.db (для SQLite) або таблиці в базі даних (для SQL Server) будуть створені.

```
dotnet ef database update
```

7. Тестування підключення.

```
using (var scope = app.Services.CreateScope())
{
```

```
    var context = scope.ServiceProvider.GetRequiredService<LibraryContext>();
    context.Database.EnsureCreated(); // Перевіряє та створює базу даних,
якщо її немає
    context.Books.Add(new Book { Title = "Test Book", Author = "Author", Year
= 2024, Genre = "Fiction" });
    context.SaveChanges();
}
```

8. ***Реалізація методів CRUD.*** Використовуючи контекст LibraryContext, реалізуйте CRUD-операції в контролері BooksController.

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;

namespace Library_API.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class BooksController : ControllerBase
    {
        private readonly LibraryContext _context;

        public BooksController(LibraryContext context)
        {
            _context = context;
        }

        [HttpGet]
        public async Task<ActionResult<IEnumerable<Book>>> GetBooks()
        {
            return await _context.Books.ToListAsync();
        }

        [HttpGet("{id}")]
        public async Task<ActionResult<Book>> GetBook(int id)
        {
            var book = await _context.Books.FindAsync(id);
            if (book == null) return NotFound();
            return book;
        }

        [HttpPost]
        public async Task<ActionResult<Book>> PostBook(Book book)
        {
            _context.Books.Add(book);
            await _context.SaveChangesAsync();
            return CreatedAtAction("GetBook", new { id = book.Id }, book);
        }

        [HttpPut("{id}")]
        public async Task<IActionResult> PutBook(int id, Book book)
        {
            if (id != book.Id) return BadRequest();
            _context.Entry(book).State = EntityState.Modified;
            await _context.SaveChangesAsync();
            return NoContent();
        }

        [HttpDelete("{id}")]
        public async Task<IActionResult> DeleteBook(int id)
        {
            var book = await _context.Books.FindAsync(id);
            if (book == null) return NotFound();
            _context.Books.Remove(book);
```

```
            await _context.SaveChangesAsync();
            return NoContent();
        }
    }
}
```

9. ***Запуск та перевірка.*** Оберіть режим запуску – IIS Express. Та запустіть додаток.

10.

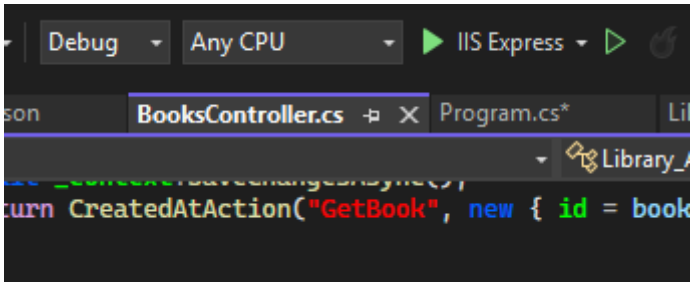

Рисунок 1 – Режим запуску IIS Express

Після запуску відкриється Swagger, де ви зможете подивитись на подробиці API.

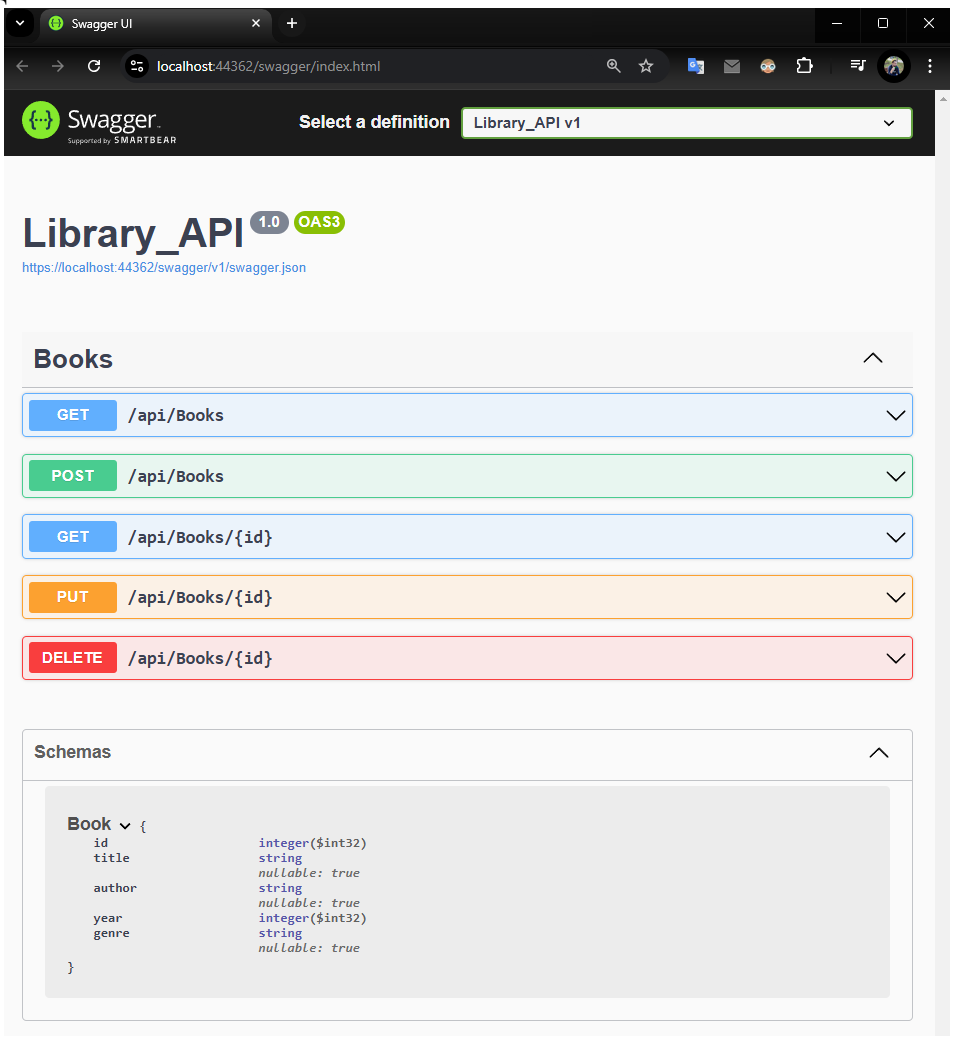

Рисунок 2 – Swagger

Також для перевірки даних можна перейти на сторінку - localhost:44362/api/books/

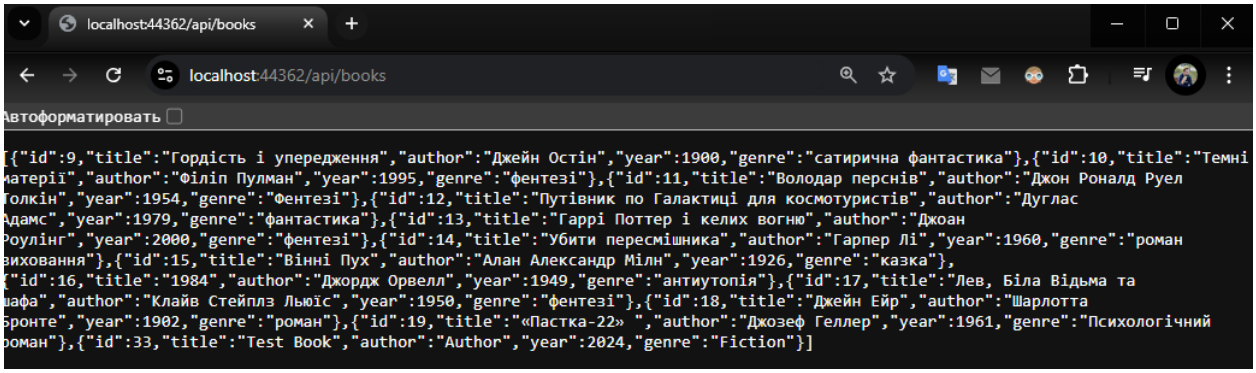

Рисунок 3 – Сторінка localhost:44362/api/books/

Розробка веб-додатка.

- Створіть проєкт Blazor (У прикладі Blazor Server).
- Додайте сервіс для звернення до API.

Створіть клас сервісу. У вашому Blazor проєкті створіть новий клас, наприклад, `BooksService`. Сервіс буде інкапсулювати всю логіку для взаємодії з API, щоб ви могли використовувати його в компонентах Blazor.

Код класу сервісу `BooksService`

```
public class BooksService
{
    private readonly HttpClient _httpClient;

    public BooksService(HttpClient httpClient)
    {
        _httpClient = httpClient;
    }

    // Отримання всіх книг
    public async Task<List<Book>> GetBooksAsync()
    {
        var response = await
_httpClient.GetFromJsonAsync<List<Book>>("api/books");
        return response ?? new List<Book>();
    }

    // Отримання книги за ID
    public async Task<Book> GetBookAsync(int id)
    {
        var response = await
_httpClient.GetFromJsonAsync<Book>($"api/books/{id}");
        return response;
    }

    // Створення нової книги
    public async Task AddBookAsync(Book book)
    {
        await _httpClient.PostAsJsonAsync("api/books", book);
    }

    // Оновлення існуючої книги
```

```
    public async Task UpdateBookAsync(Book book)
    {
        await _httpClient.PutAsJsonAsync($"api/books/{book.Id}", book);
    }

    // Видалення книги
    public async Task DeleteBookAsync(int id)
    {
        await _httpClient.DeleteAsync($"api/books/{id}");
    }
}
```

Зареєструйте сервіс у DI контейнері. Blazor використовує контейнер Dependency Injection (DI) для ін'єкції залежностей. Для використання сервісу BooksService його потрібно зареєструвати в Program.cs.

Код для реєстрації сервісу. Відкрийте файл Program.cs і додайте наступний рядок:

builder.Services.AddScoped<BooksService>();

Тут AddScoped означає, що один екземпляр сервісу буде створений для кожного користувача під час одного сеансу.

Налаштування базового URL для API. У файлі Program.cs вашого Blazor-додатку налаштуйте HttpClient для роботи з API:

```
builder.Services.AddScoped(sp => new HttpClient
{
    BaseAddress = new Uri("https://localhost:44362/")
      });
```

Використання сервісу в компонентах Blazor. Ін'єкція сервісу в компонент. У компоненті Blazor (наприклад, Books.razor) потрібно додати залежність від BooksService за допомогою директиви @inject.

Приклад:

```
@page "/"
@inject BooksService BooksService
@rendermode InteractiveServer
@inject NavigationManager NavigationManager

<h3>Список книг</h3>

@if (books == null)
{
    <p>Завантаження...</p>
}
else
{
    <ul>
        @foreach (var book in books)
        {
            <li class="list-item">
                <button @onclick="async() => EditBook(book.Id)" class="btn
btn-edit">Редагувати</button>
                <button @onclick="async() => DeleteBook(book.Id)" class="btn
btn-delete">Видалити</button>
                <span class="book-list"> @book.Title - @book.Author у жанрі
@book.Genre (@book.Year) </span>
            </li>
```

```
        }
    </ul>
}

<button @onclick="async() => AddNewBook()" class="btn btn-add">Додати нову
книгу</button>

@code {
    private List<Book> books;

    protected override async Task OnInitializedAsync()
    {
        books = await BooksService.GetBooksAsync();
    }

    private async Task DeleteBook(int id)
    {
        await BooksService.DeleteBookAsync(id);
        books = await BooksService.GetBooksAsync(); // Перезавантажуємо
список
    }

    private void EditBook(int id)
    {
        NavigationManager.NavigateTo($"/edit/{id}");
    }

    private async Task AddNewBook()
    {
        NavigationManager.NavigateTo("/add");
    }
}
```

Налагодження сервісу. Для перевірки роботи сервісу виконайте такі кроки:

1. Запустіть бекенд-сервер ASP.NET Core Web API.

2. Переконайтеся, що в Blazor WebAssembly проєкті файл appsettings.json містить правильний базовий URL вашого API.

3. Запустіть веб-додаток і перевірте, чи дані коректно відображаються на сторінці.

Додаток 1 містить лістинг основних файлів REST API, реалізованого засобами ASP.NET Core. У файлі BooksController.cs представлено реалізацію контролера для роботи з книгами, що підтримує CRUD-операції (створення, читання, оновлення та видалення записів). Файл Program.cs відповідає за конфігурацію застосунку, реєстрацію сервісів та налаштування маршрутизації. Appsettings.json містить конфігураційні параметри, зокрема налаштування бази даних. У файлі Library_API.http наведено приклади HTTP-запитів для тестування API.

Додаток 2 містить лістинг основних файлів веб-сервісу, створеного на основі Blazor. BooksService.cs реалізує логіку взаємодії веб-додатку з API. Файл Program.cs містить основні налаштування Blazor-застосунку. NavMenu.razor відповідає за навігаційне меню, а файли Books.razor, Add.razor,

Edit.razor містять компоненти для перегляду списку книг, додавання нових записів і редагування наявних. Файл app.css забезпечує стилізацію інтерфейсу.

На рисунку 4 зображено загальний інтерфейс програми, що демонструє основний функціонал застосунку. Рисунок 5 показує процес додавання нової книги через відповідну форму. На рисунку 6 продемонстровано редагування наявного запису, де користувач може змінити дані книги. Рисунок 8 ілюструє процес видалення книги з бази даних.

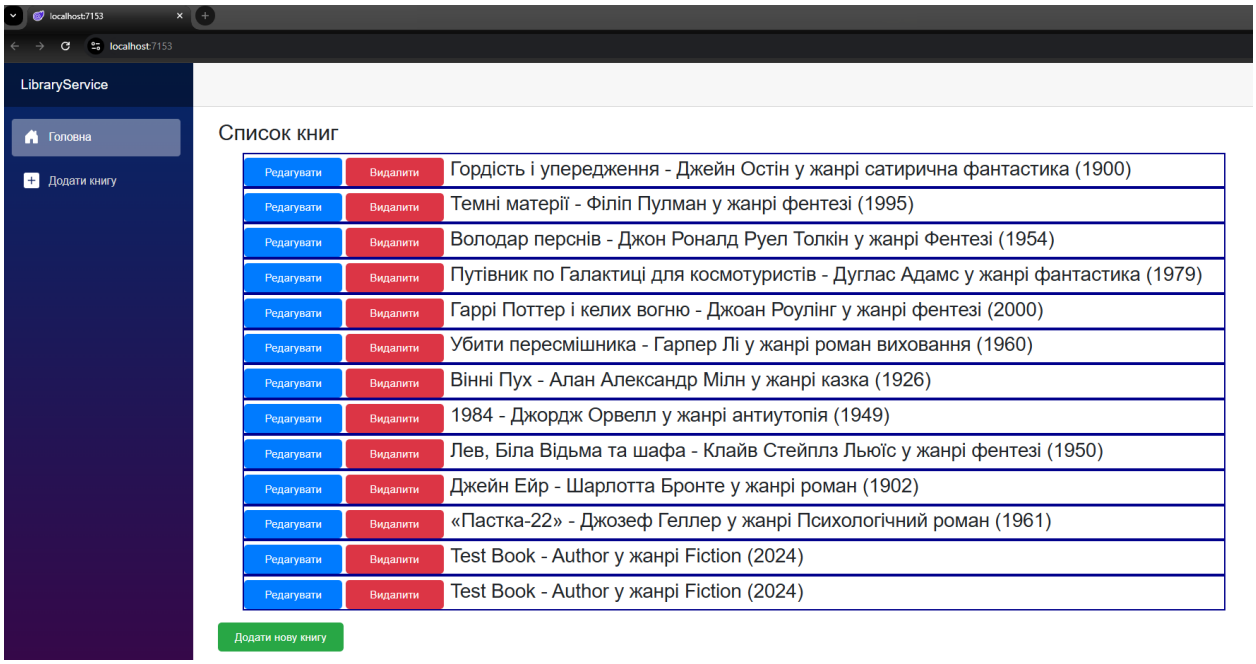

Рисунок 4 – Приклад роботи програми

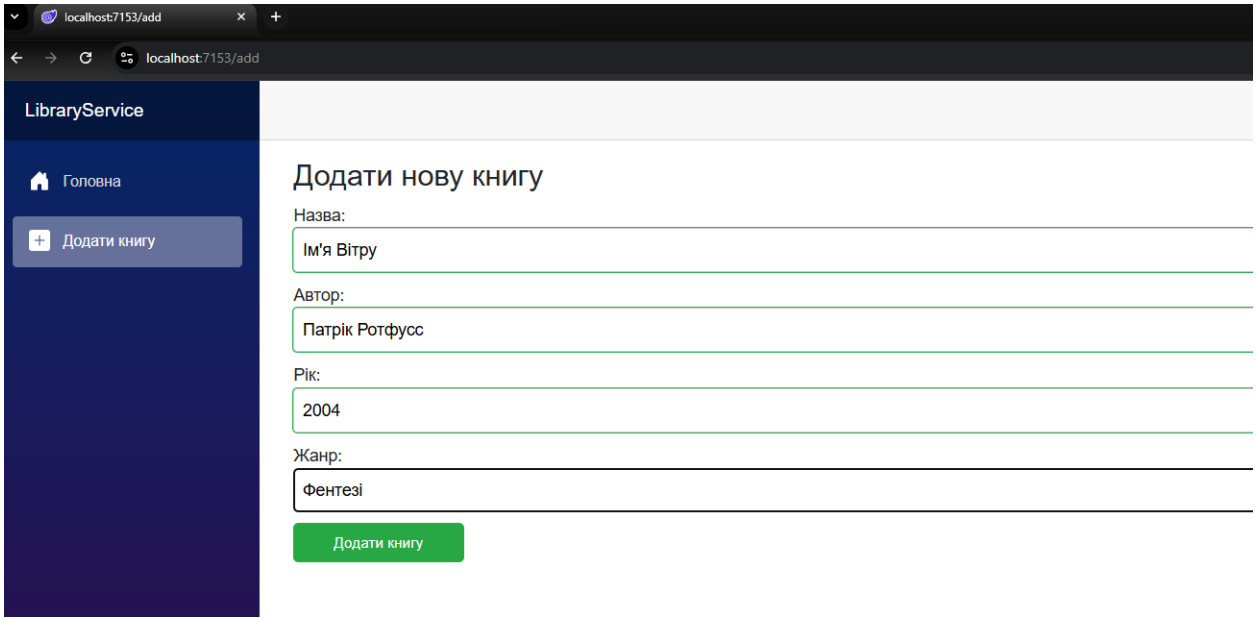

Рисунок 5 – Приклад роботи програми (Додавання)

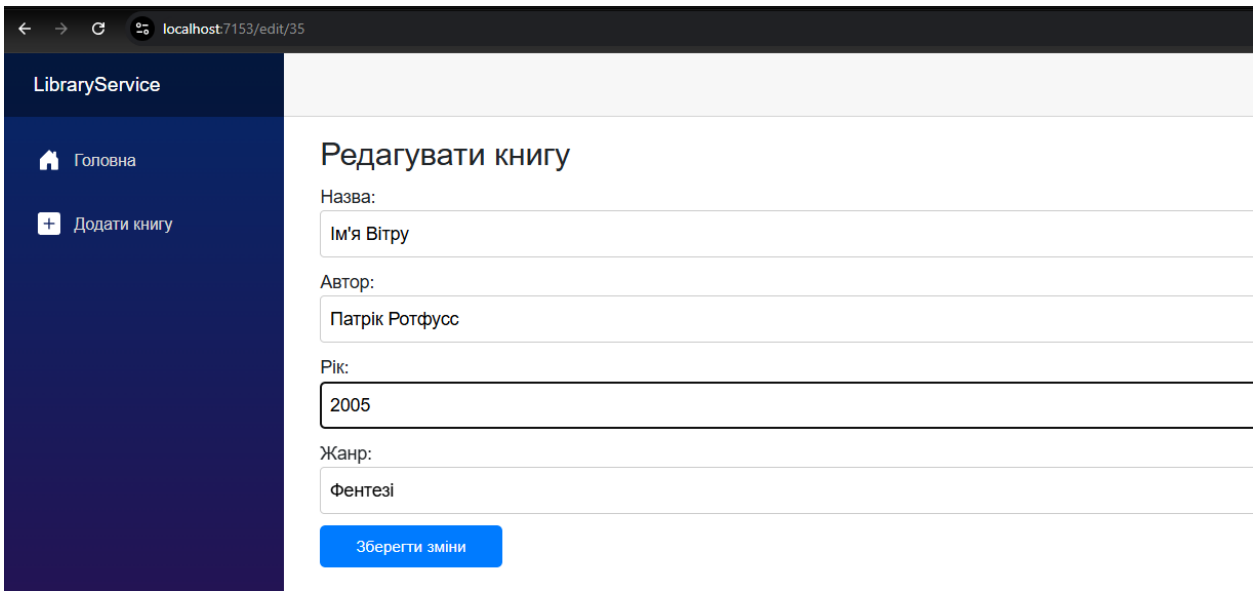

Рисунок 6 – Приклад роботи програми (Редагування)

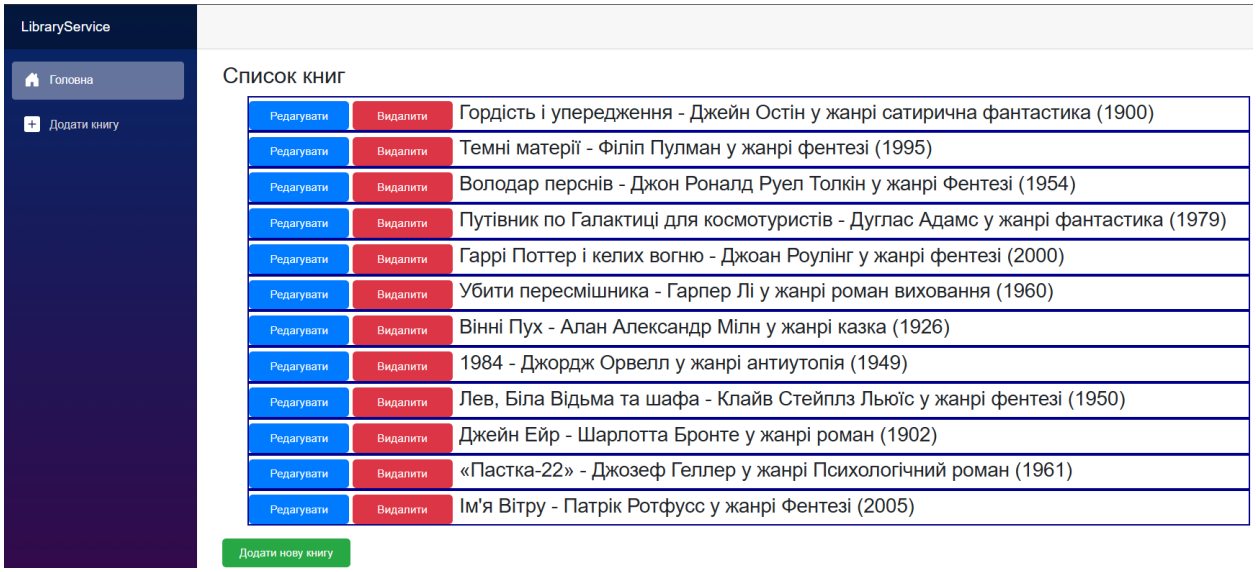

Рисунок 8 – Приклад роботи програми (Видалення)

Питання для самоконтролю

1. Що таке REST API? Які основні принципи його роботи?
2. Які HTTP-методи (GET, POST, PUT, DELETE) використовуються для роботи з ресурсами в REST API? Наведіть приклади.
3. Як налаштувати маршрут для API-контролера в ASP.NET Core?
4. Що таке модель у контексті ASP.NET Core Web API? Для чого вона використовується?
5. Як налаштувати підключення до бази даних у файлі appsettings.json?
6. Як перевірити, що ваш API правильно обробляє запити та повертає відповіді?
7. Що таке міграції у Entity Framework Core, і чому вони важливі?
8. Як створити та застосувати міграції для бази даних за допомогою команд CLI?
9. Що робити, якщо після застосування міграції виникає помилка в базі даних?
10. Які технології використовуються для побудови клієнтської частини веб-додатка?
11. Як налаштувати HTTP-клієнт у веб-додатку для звернення до REST API?
12. Як правильно обробляти відповіді API у клієнтському додатку?
13. Як додати механізм обробки помилок, якщо запит до API не виконується?

СПИСОК РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ

1. Freeman A. Essential ASP.NET Core 3. / A. Freeman. - Apress, 2020. - 1 296 p.

2. Hermes D. Xamarin Mobile Application Development: Cross-Platform C# and Xamarin.Forms Fundamentals / D. Hermes. - Apress, 2015. - 432 c.

3. Ye R. NET MAUI Cross-Platform Application Development: Leverage a first-class cross-platform UI framework to build native apps on multiple platforms / R.Ye. - Packt Publishing Ltd, 2023. - 400 p.

4. Engström J. Web Development with Blazor: A practical guide to building interactive UIs with C# 12 and .NET 8. /J. Engström. - 3rd ed. - Packt Publishing Ltd, 2024. - 366 p.

5. Wright T. B. Blazor WebAssembly by Example: A project-based guide to building web apps with .NET, Blazor WebAssembly, and C# / T. B. Wright, S. Hanselman. - Packt Publishing Ltd, 2021. - 266 p.

6. Price M. J. C# 9 and .NET 5 – Modern Cross-Platform Development: Build intelligent apps, websites, and services with Blazor, ASP.NET Core, and Entity Framework Core using Visual Studio Code. - 5th ed. - Packt Publishing Ltd, 2020. - 822 p.

Додаток А. Лістинг основних файлів. REST API ASP.NET Core.

BooksController.cs

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;

namespace Library_API.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class BooksController : ControllerBase
    {
        private readonly LibraryContext _context;

        public BooksController(LibraryContext context)
        {
            _context = context;
        }

        [HttpGet]
        public async Task<ActionResult<IEnumerable<Book>>> GetBooks()
        {
            return await _context.Books.ToListAsync();
        }

        [HttpGet("{id}")]
        public async Task<ActionResult<Book>> GetBook(int id)
        {
            var book = await _context.Books.FindAsync(id);
            if (book == null) return NotFound();
            return book;
        }

        [HttpPost]
        public async Task<ActionResult<Book>> PostBook(Book book)
        {
            _context.Books.Add(book);
            await _context.SaveChangesAsync();
            return CreatedAtAction("GetBook", new { id = book.Id }, book);
        }

        [HttpPut("{id}")]
        public async Task<IActionResult> PutBook(int id, Book book)
        {
            if (id != book.Id) return BadRequest();
            _context.Entry(book).State = EntityState.Modified;
            await _context.SaveChangesAsync();
            return NoContent();
        }

        [HttpDelete("{id}")]
        public async Task<IActionResult> DeleteBook(int id)
        {
            var book = await _context.Books.FindAsync(id);
            if (book == null) return NotFound();
            _context.Books.Remove(book);
            await _context.SaveChangesAsync();
            return NoContent();
        }
    }
}
```

Program.cs

```
using System.Collections.Generic;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Sqlite;
using System.Reflection.Emit;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddControllers();
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();
// Додаємо сервіс для EF Core
builder.Services.AddDbContext<LibraryContext>(options =>

options.UseSqlite(builder.Configuration.GetConnectionString("DefaultConnection")));
// Якщо використовуєте SQL Server:
//
options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection"))

var app = builder.Build();

//using (var scope = app.Services.CreateScope())
//{
//var context = scope.ServiceProvider.GetRequiredService<LibraryContext>();
//context.Database.EnsureCreated(); // Перевіряє та створює базу даних, якщо її
немає
//context.Books.Add(new Book { Title = "Test Book", Author = "Author", Year = 2024,
Genre = "Fiction" });
//context.SaveChanges();
//}
// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}

app.UseHttpsRedirection();

app.MapControllers();
app.Run();
public class Book
{
    public int Id { get; set; } // Первинний ключ
    public string Title { get; set; } // Назва книги
    public string Author { get; set; } // Автор
    public int Year { get; set; } // Рік видання
    public string Genre { get; set; } // Жанр
}
public class LibraryContext : DbContext
{
    public LibraryContext(DbContextOptions<LibraryContext> options) : base(options)
{ }

    // Визначення таблиць у базі даних
    public DbSet<Book> Books { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        // Налаштування таблиці Books
        modelBuilder.Entity<Book>().ToTable("Books");
        modelBuilder.Entity<Book>().HasKey(b => b.Id); // Вказуємо первинний ключ
        modelBuilder.Entity<Book>().Property(b => b.Title).IsRequired(); // Поле
Title є обов'язковим
        modelBuilder.Entity<Book>().Property(b => b.Author).IsRequired();
```

```
    }
}
```

appsettings.json

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "DefaultConnection": "Data Source=library.db"
  }
}
```

Library_API.http

```
@Library_API_HostAddress = http://localhost:5046

GET {{Library_API_HostAddress}}/api/BooksController/
Accept: application/json
```

Додаток Б. Лістинг основних файлів веб-сервісу в Blazor

BooksService.cs

```
using System.Net.Http;
using System.Net.Http.Json;
using System.Threading.Tasks;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
public class Book
{
    public int Id { get; set; }
    [Required(ErrorMessage = "Назва книги є обов'язковою")]
    public string Title { get; set; }

    [Required(ErrorMessage = "Автор книги є обов'язковим")]
    public string Author { get; set; }

    [Range(1900, 2100, ErrorMessage = "Введіть коректний рік")]
    public int Year { get; set; }

    public string Genre { get; set; }
}
public class BooksService
{
    private readonly HttpClient _httpClient;

    public BooksService(HttpClient httpClient)
    {
        _httpClient = httpClient;
    }

    // Отримання всіх книг
    public async Task<List<Book>> GetBooksAsync()
    {
        var response = await _httpClient.GetFromJsonAsync<List<Book>>("api/books");
        return response ?? new List<Book>();
    }

    // Отримання книги за ID
    public async Task<Book> GetBookAsync(int id)
    {
        var response = await _httpClient.GetFromJsonAsync<Book>($"api/books/{id}");
        return response;
    }

    // Створення нової книги
    public async Task AddBookAsync(Book book)
    {
        await _httpClient.PostAsJsonAsync("api/books", book);
    }

    // Оновлення існуючої книги
    public async Task UpdateBookAsync(Book book)
    {
        await _httpClient.PutAsJsonAsync($"api/books/{book.Id}", book);
    }

    // Видалення книги
    public async Task DeleteBookAsync(int id)
    {
        await _httpClient.DeleteAsync($"api/books/{id}");    }
```

```
}
```

Program.cs

```
using LibraryService.Components;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddScoped(sp => new HttpClient
{
    BaseAddress = new Uri("https://localhost:44362/")
});
// Add services to the container.
builder.Services.AddRazorComponents()
    .AddInteractiveServerComponents();

builder.Services.AddScoped<BooksService>();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error", createScopeForErrors: true);
    // The default HSTS value is 30 days. You may want to change this for production
scenarios, see https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseAntiforgery();
app.MapRazorComponents<App>()
    .AddInteractiveServerRenderMode();
app.Run();
```

NavMenu.razor

```
<div class="top-row ps-3 navbar navbar-dark">
    <div class="container-fluid">
        <a class="navbar-brand" href="">LibraryService</a>
    </div>
</div>

<input type="checkbox" title="Navigation menu" class="navbar-toggler" />

<div class="nav-scrollable" onclick="document.querySelector('.navbar-
toggler').click()">
    <nav class="flex-column">
        <div class="nav-item px-3">
            <NavLink class="nav-link" href="" Match="NavLinkMatch.All">
                <span class="bi bi-house-door-fill-nav-menu" aria-
hidden="true"></span> Головна
            </NavLink>
        </div>

        <div class="nav-item px-3">
            <NavLink class="nav-link" href="Add">
                <span class="bi bi-plus-square-fill-nav-menu" aria-
hidden="true"></span> Додати книгу
            </NavLink>
        </div>
    </nav>
```

```
</div>
```

Books.razor

```
@page "/"
@inject BooksService BooksService
@rendermode InteractiveServer
@inject NavigationManager NavigationManager

<h3>Список книг</h3>

@if (books == null)
{
    <p>Завантаження...</p>
}
else
{
    <ul>
        @foreach (var book in books)
        {
            <li class="list-item">
                <button @onclick="async() => EditBook(book.Id)" class="btn btn-
edit">Редагувати</button>
                <button @onclick="() => DeleteBook(book.Id)" class="btn btn-
delete">Видалити</button>
                <span class="book-list"> @book.Title - @book.Author у жанрі
@book.Genre (@book.Year) </span>
            </li>

        }
    </ul>
}

<button @onclick="async() => AddNewBook()" class="btn btn-add">Додати нову
книгу</button>

@code {
    private List<Book> books;

    protected override async Task OnInitializedAsync()
    {
        books = await BooksService.GetBooksAsync();
    }

    private async Task DeleteBook(int id)
    {
        await BooksService.DeleteBookAsync(id);
        books = await BooksService.GetBooksAsync(); // Перезавантажуємо список
    }

    private void EditBook(int id)
    {
        NavigationManager.NavigateTo($"/edit/{id}");
    }

    private async Task AddNewBook()
    {
        NavigationManager.NavigateTo("/add");
    }
}
```

Add.razor

```
@page "/add"
@inject BooksService BooksService
@rendermode InteractiveServer
@inject NavigationManager NavigationManager

<h3>Додати нову книгу</h3>

<EditForm Model="newBook" OnValidSubmit="HandleValidSubmit">
    <DataAnnotationsValidator />
    <ValidationSummary />

    <div>
        <label for="title">Назва:</label>
        <InputText id="title" @bind-Value="newBook.Title" />
    </div>
    <div>
        <label for="author">Автор:</label>
        <InputText id="author" @bind-Value="newBook.Author" />
    </div>
    <div>
        <label for="year">Рік:</label>
        <InputNumber id="year" @bind-Value="newBook.Year" />
    </div>
    <div>
        <label for="genre">Жанр:</label>
        <InputText id="genre" @bind-Value="newBook.Genre" />
    </div>

    <button type="submit" class="btn btn-add">Додати книгу</button>
</EditForm>

@code {
    private Book newBook = new Book();

    private async Task HandleValidSubmit()
    {
        await BooksService.AddBookAsync(newBook);
        NavigationManager.NavigateTo("/"); // Після додавання повертаємо на список
    }
}
```

Edit.razor

```
@page "/edit/{id:int}"
@inject BooksService BooksService
@rendermode InteractiveServer
@inject NavigationManager NavigationManager

<h3>Редагувати книгу</h3>
@if (book == null)
{
    <p>Завантаження...</p>
}
else
{

    <EditForm Model="@book" OnValidSubmit="HandleValidSubmit">
        <DataAnnotationsValidator />
        <ValidationSummary />

        <div>
```

```
            <label for="title">Назва:</label>
            <InputText id="title" @bind-Value="book.Title" />
        </div>
        <div>
            <label for="author">Автор:</label>
            <InputText id="author" @bind-Value="book.Author" />
        </div>
        <div>
            <label for="year">Рік:</label>
            <InputNumber id="year" @bind-Value="book.Year" />
        </div>
        <div>
            <label for="genre">Жанр:</label>
            <InputText id="genre" @bind-Value="book.Genre" />
        </div>

        <button type="submit" class="btn btn-edit">Зберегти зміни </button>
    </EditForm>
}

@code {
    [Parameter] public int Id { get; set; }
    private Book book;

    protected override async Task OnInitializedAsync()
    {
        book = await BooksService.GetBookAsync(Id);
    }

    private async Task HandleValidSubmit()
    {
        await BooksService.UpdateBookAsync(book);
        NavigationManager.NavigateTo("/"); // Після редагування повертаємо на список
    }
}
```

app.css

```
html, body {
    font-family: 'Helvetica Neue', Helvetica, Arial, sans-serif;
}

a, .btn-link {
    color: #006bb7;
}

.btn-primary {
    color: #fff;
    background-color: #1b6ec2;
    border-color: #1861ac;
}

.btn:focus, .btn:active:focus, .btn-link.nav-link:focus, .form-control:focus, .form-
check-input:focus {
  box-shadow: 0 0 0 0.1rem white, 0 0 0 0.25rem #258cfb;
}

.content {
    padding-top: 1.1rem;
}

h1:focus {
    outline: none;
}
```

```css
.valid.modified:not([type=checkbox]) {
    outline: 1px solid #26b050;
}

.invalid {
    outline: 1px solid #e50000;
}

.validation-message {
    color: #e50000;
}

.blazor-error-boundary {
    background:
url() no-repeat 1rem/1.8rem, #b32121;
    padding: 1rem 1rem 1rem 3.7rem;
    color: white;
}

    .blazor-error-boundary::after {
        content: "An error has occurred."
    }

.darker-border-checkbox.form-check-input {
    border-color: #929292;
}


.book-list {
    list-style-type: none;
    padding: 2px;
    margin: 2px;
    font-size: 25px;
}

.book-item {
    display: flex;
    justify-content: space-between;
    align-items: center;
    padding: 10px;
    border-bottom: 1px solid #ccc;
    margin-bottom: 10px;
}

.book-actions {
    display: flex;
```

```
    gap: 10px;
}

    .book-actions button {
        width: 120px;
    }

.btn {
    padding: 8px 12px;
    font-size: 14px;
    cursor: pointer;
    background-color: #007BFF;
    color: white;
    border: none;
    border-radius: 5px;
}

.btn-add {
    background-color: #28a745;
    color: white;
}

.btn-edit {
    background-color: #007bff;
    color: white;
}

.btn-delete {
    background-color: #dc3545;
    color: white;
}
.buttons {
    display: flex;
    gap: 10px;
}
.text {
    font-size: 25px;
}

.btn {
    width: 10%;
}
form div {
    margin-bottom: 10px;
}

input, select, button {
    width: 100%;
    padding: 8px;
    font-size: 16px;
    border: 1px solid #ccc;
    border-radius: 4px;
}

    input[type="number"] {
        width: 100%;
    }
li{
    width: 100%;
    list-style-type: none;
    border: 2px solid darkblue;
    align-content: space-between;
}
```