Лабораторна робота №3

Звіт

З дисципліни “Сучасні інтернет технології”
на тему: “Конфігурація в ASP.NET Core застосунку ”.

Студента 4 курсу: Групи МІТ-41
Демиденко Андрій

Київ - 2025р.

Хід виконання роботи

Тема: Конфігурація в ASP.NET Core застосунку.
Мета: Метою даної лабораторної роботи є здобуття практичних навичок з налаштування конфігурації проєкту ASP.NET Core. Це включає вивчення механізму постачальників конфігурації , реалізацію багатошарової конфігурації для різних середовищ (Development, Production) та безпечне зберігання конфіденційних даних (секретів) за допомогою User Secrets та змінних середовища .

Забезпечте проєкт файлами особливої конфігурації

Для забезпечення гнучкості та безпеки конфігурації, проєкт було доповнено файлами sharedsettings.json (для спільних налаштувань, які не є секретами), appsettings.Development.json (для середовища розробки) та appsettings.Production.json (для промислового середовища). У кожному файлі було задано різні значення для параметрів ApplicationName та MySpecificSetting:ApiUrl, що дозволяє застосунку змінювати поведінку залежно від середовища (рис. 1).

Рис. 1 – Файли конфігурації.

Ключовим кроком є налаштування порядку завантаження у файлі Program.cs. Джерела конфігурації за замовчуванням були очищені (builder.Configuration.Sources.Clear()), після чого файли були додані у порядку зростання пріоритету: sharedsettings.json -> appsettings.json -> appsettings.{EnvironmentName}.json . Це гарантує, що специфічні для середовища налаштування (наприклад, з appsettings.Development.json) коректно перезапишуть базові (рис. 2).

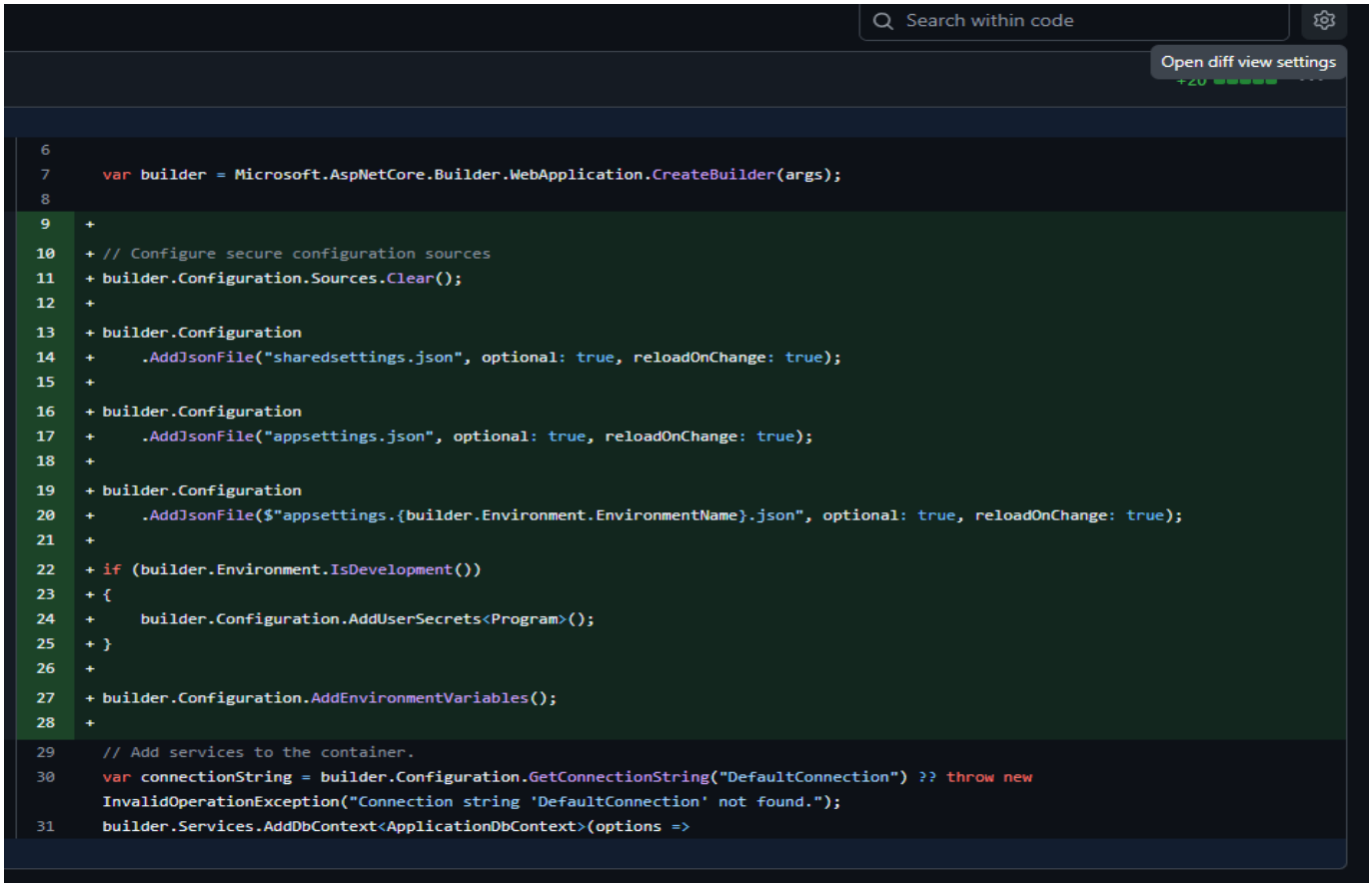

Рис. 2 – налаштування порядку завантаження у файлі Program.cs.

Безпечне зберігання ConnectionString

З метою запобігання потраплянню конфіденційних даних (рядків підключення) у систему контролю версій, параметр ConnectionStrings:DefaultConnection було повністю видалено з файлу appsettings.json (рис. 3).

Рис. 3 – Видалення ConnectionStrings:DefaultConnection.

Для середовища розробки (Development) рядок підключення було перенесено у Менеджер Секретів (User Secrets). Це безпечне сховище, що не є частиною проєкту і не потрапляє на GitHub. Завантаження секретів відбувається у Program.cs лише в режимі розробки через виклик builder.Configuration.AddUserSecrets<Program>() (рис. 4).

Для промислового середовища (Production) передбачено використання Змінних Середовища (Environment Variables). Вони завантажуються останніми (builder.Configuration.AddEnvironmentVariables()) і мають найвищий пріоритет. Таким чином, код builder.Configuration.GetConnectionString("DefaultConnection") коректно обробляє різні значення для різних середовищ, не зберігаючи секрети у коді.

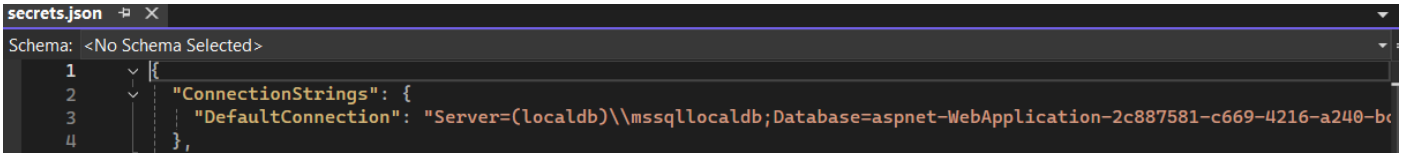

Рис. 4 – Менеджер Секретів (User Secrets).

Реалізація шаблону "Options Pattern"

Для уникнення помилок при роботі з рядковими ключами та для дотримання принципів SOLID було реалізовано шаблон "Options Pattern". Для цього було створено ієрархію C#-класів (WebAppConfiguration, MySpecificSettingConfig, ConnectionStringsConfig тощо), що точно відображають структуру JSON-файлів конфігурації (рис. 5).

```
public class WebAppConfiguration
{
    public LoggingConfig? Logging { get; set; }
    public string? AllowedHosts { get; set; }

    public ConnectionStringsConfig? ConnectionStrings { get; set; }
    public string? ApplicationName { get; set; }
    public MySpecificSettingConfig? MySpecificSetting { get; set; }
}

public class LoggingConfig
{
    public LogLevelConfig? LogLevel { get; set; }
}

public class LogLevelConfig
{
    public string? Default { get; set; }
    public string? MicrosoftAspNetCore { get; set; }
}

public class ConnectionStringsConfig
{
    public string? DefaultConnection { get; set; }
}

public class MySpecificSettingConfig
{
    public string? ApiUrl { get; set; }
    public string? ApiKey { get; set; }
}
```

Рис. 5 – WebAppConfiguration, MySpecificSettingConfig, ConnectionStringsConfig.

У Program.cs вся конфігурація була зчитана та прив'язана до екземпляра класу WebAppConfiguration методом builder.Configuration.Get<WebAppConfiguration>(). Цей об'єкт був зареєстрований у контейнері впровадження залежностей (DI) з життєвим циклом Singleton (рис. 6).

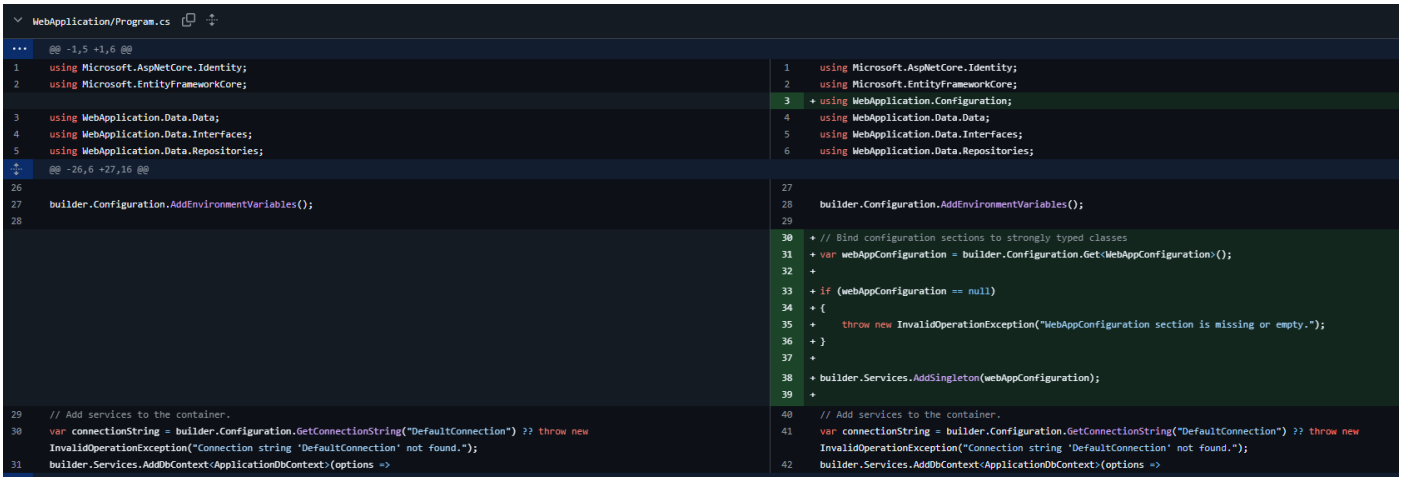

Рис. 6 – Singleton.

Зареєстрований сервіс конфігурації WebAppConfiguration було "інжектовано" через конструктор у HomeController . Параметри ApplicationName та MySpecificSetting.ApiUrl були зчитані з об'єкта конфігурації та передані у ViewData (рис. 7).

Рис. 7 – конструктор у HomeController.

На останок, у файлі Views/Shared/_Layout.cshtml ці значення з ViewData були використані для динамічного відображення у Footer веб-інтерфейсу, що демонструє роботу всього ланцюга (рис. 8).

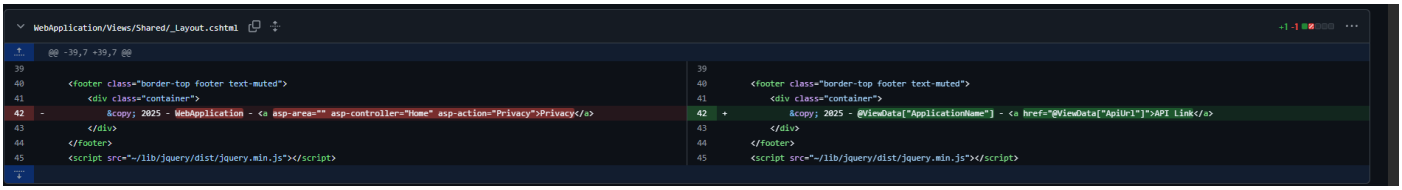

Рис. 8 – Зміни у Layout.cshtml.

Безпечне налаштування ApiKey

Аналогічно до ConnectionString, забезпечено безпечне зберігання ApiKey. Властивість ApiKey було додано до строго типізованого класу MySpecificSettingConfig (рис. 9).

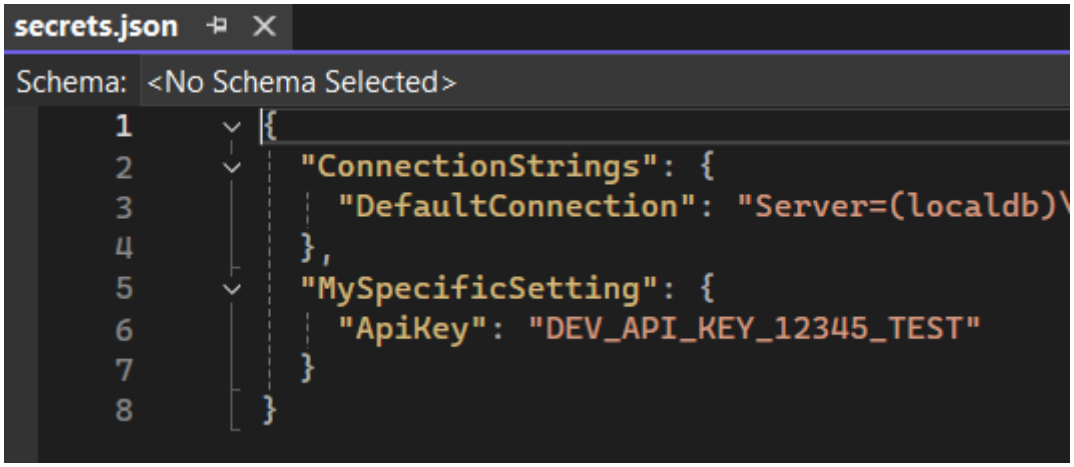

Рис. 9 – безпечне зберігання ApiKey.

Для середовища розробки значення ApiKey було додано до файлу secrets.json. Для Production середовища передбачено використання змінної середовища MySpecificSetting:ApiKey. Це гарантує, що для різних середовищ використовуються різні, безпечно збережені ключі.

Теоретичні основи Middleware

Конвеєр обробки запитів (Request Pipeline) — це центральна концепція ASP.NET Core. Коли надходить HTTP-запит, він не одразу потрапляє до контролера, а проходить через "конвеєр", що складається з компонентів **Middleware** (проміжне програмне забезпечення).

Кожен компонент Middleware отримує запит, може виконати над ним певні дії (наприклад, перевірити автентифікацію, залогувати дані) і потім, за допомогою **делегата next**, вирішує, чи передати запит наступному компоненту в ланцюжку. Існують два основні типи Middleware:

1. **Системне (вбудоване):** Надається самим фреймворком.
    - *Приклади:* app.UseRouting() (визначає, який контролер викликати), app.UseAuthentication() (ідентифікує користувача), app.UseStaticFiles() (роздає CSS, JS, зображення).
2. **Користувацьке (Custom):** Створюється розробником для специфічних завдань (наприклад, логування, обробка специфічних помилок).

Впровадження Rate Limiting Middleware

Для захисту застосунку від надмірної кількості запитів було реалізовано **розділене (partitioned) обмеження швидкості**. У Program.cs було налаштовано сервіс AddRateLimiter. Використовуючи PartitionedRateLimiter.Create, було створено два різні ліміти:

- **Для автентифікованих користувачів:** Ліміт PermitLimit = 10 на хвилину, ідентифікація за ClaimTypes.NameIdentifier (ID користувача).
- **Для анонімних користувачів:** Ліміт PermitLimit = 5 на хвилину, ідентифікація за httpContext.Connection.RemoteIpAddress (IP-адреса).

Також було налаштовано обробник OnRejected, який у разі перевищення ліміту повертає клієнту статус **429 Too Many Requests** (рис. 10).

```
+ // Configure Rate Limiting
+ builder.Services.AddRateLimiter(options =>
+ {
+     options.OnRejected = (context, _) =>
+     {
+         context.HttpContext.Response.StatusCode = StatusCodes.Status429TooManyRequests;
+         return new ValueTask();
+     };
+
+     options.GlobalLimiter = PartitionedRateLimiter.Create<HttpContext, string>(httpContext =>
+     {
+         if (httpContext.User.Identity?.IsAuthenticated == true)
+         {
+             var userId = httpContext.User.FindFirstValue(ClaimTypes.NameIdentifier)!;
+             return RateLimitPartition.GetFixedWindowLimiter(userId, _ =>
+                 new FixedWindowRateLimiterOptions
+                 {
+                     PermitLimit = 10,
+                     Window = TimeSpan.FromMinutes(1),
+                     AutoReplenishment = true // maybe set explicitly
+                 });
+         }
+         else
+         {
+             var ip = httpContext.Connection.RemoteIpAddress?.ToString() ?? "unknown_ip";
+             return RateLimitPartition.GetFixedWindowLimiter(ip, _ =>
+                 new FixedWindowRateLimiterOptions
+                 {
+                     PermitLimit = 5,
+                     Window = TimeSpan.FromMinutes(1),
+                     AutoReplenishment = true
+                 });
+         }
+     });
+ });
```

Рис. 10 – сервіс AddRateLimiter.

Наостанок, middleware було додано до конвеєра обробки запитів викликом app.UseRateLimiter(). Його було розміщено *після* app.UseAuthentication() та app.UseAuthorization(), щоб RateLimiter мав доступ до даних про автентифікованого користувача (httpContext.User) (рис. 11).

```
// Configure the HTTP request pipeline.
@@ -68,8 +107,11 @@

app.UseRouting();

app.UseAuthorization();

app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

```
// Configure the HTTP request pipeline.

app.UseRouting();

+ app.UseAuthentication();
app.UseAuthorization();

+ app.UseRateLimiter();
+
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

Рис. 11 – app.UseRateLimiter().

Висновки:

Під час виконання лабораторної роботи було досягнуто всіх поставлених цілей. Ми успішно реалізували багатошарову систему конфігурації, розділивши налаштування на файли sharedsettings.json, appsettings.json та специфічні для середовищ appsettings.Development.json і appsettings.Production.json.

Було забезпечено належний рівень безпеки для конфіденційних даних: ConnectionString та ApiKey були вилучені з файлів конфігурації, що потрапляють до репозиторію. Натомість, для середовища Development вони зберігаються у User Secrets, а для Production — у Змінних Середовища.

Було успішно реалізовано шаблон "Options Pattern". Ми створили ієрархію класів конфігурації (WebAppConfiguration), зареєстрували її в контейнері DI як Singleton та впровадили у HomeController для динамічного відображення параметрів у Footer інтерфейсу.

Нарешті, було вивчено теорію middleware та реалізовано розділене (partitioned) Rate Limiting middleware. Застосунок коректно обмежує кількість запитів, надаючи різні привілеї для автентифікованих та анонімних користувачів, і повертає статус 429 Too Many Requests у разі перевищення ліміту.