

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).

```
WebApplication/appsettings.Development.json
...
1  {
2  -   "Logging": {
3  -     "LogLevel": {
4  -       "Default": "Information",
5  -       "Microsoft.AspNetCore": "Warning"
6  -     }
7  -   }
8  }

WebApplication/appsettings.Production.json
...
1  + {
2  +   "ApplicationName": "My WebApp (PRODUCTION)",
3  +   "MySpecificSetting": {
4  +     "ApiUrl": "https://production-api.example.com"
5  +   }
6  + }

WebApplication/appsettings.json
...
1  @@ -2,11 +2,8 @@
2  -   "ConnectionString": {
3  -     "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=aspnet-WebApplication-2c887581-c669-4216-a240-
4  bcc2e69f0381;Trusted_Connection=True;MultipleActiveResultSets=true"
5  -   },
6  -   "Logging": {
7  -     "LogLevel": {
8  -       "Default": "Information",
9  -       "Microsoft.AspNetCore": "Warning"
10 -    }
11 -   },
12 -   "AllowedHosts": "*"
13 - }
```

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).

```

6
7     var builder = Microsoft.AspNetCore.Builder.WebApplication.CreateBuilder(args);
8
9     +
10    // Configure secure configuration sources
11    builder.Configuration.Sources.Clear();
12    +
13    builder.Configuration
14      .AddJsonFile("sharedsettings.json", optional: true, reloadOnChange: true);
15    +
16    builder.Configuration
17      .AddJsonFile("appsettings.json", optional: true, reloadOnChange: true);
18    +
19    builder.Configuration
20      .AddJsonFile($"appsettings.{builder.Environment.EnvironmentName}.json", optional: true, reloadOnChange: true);
21    +
22    if (builder.Environment.IsDevelopment())
23    {
24      builder.Configuration.AddUserSecrets<Program>();
25    }
26    +
27    builder.Configuration.AddEnvironmentVariables();
28    +
29    // Add services to the container.
30    var connectionString = builder.Configuration.GetConnectionString("DefaultConnection") ?? throw new
31    InvalidOperationException("Connection string 'DefaultConnection' not found.");

```

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 файла appsettings.json (рис. 3).

```

...
@@ -1,7 +1,4 @@
1  {
2    "ConnectionStrings": {
3      "DefaultConnection": "Server=(localdb)\mssqllocaldb;Database=aspnet-WebApplication-2c887581-c669-4216-a240-
4      bcc2e69f0381;Trusted_Connection=True;MultipleActiveResultSets=true"
5    },
6    "ApplicationName": "My WebApp (Base)",
7    "MySpecificSetting": {
8      "ApiUrl": "https://api.example.com"
...

```

```

1  {
2    "ApplicationName": "My WebApp (Base)",
3    "MySpecificSetting": {
4      "ApiUrl": "https://api.example.com"

```

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.

```
secrets.json
Schema: <No Schema Selected>
1  {
2    "ConnectionStrings": {
3      "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=aspnet-WebApplication-2c887581-c669-4216-a240-bc
4    },
```

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 ієархію 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).

```

1  using Microsoft.AspNetCore.Identity;
2  using Microsoft.EntityFrameworkCore;
3
4  using WebApplication.Data.Data;
5  using WebApplication.Data.Interfaces;
6  using WebApplication.Data.Repositories;
7
8  @@ -1,5 +1,6 @@
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42

```

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).

```

1  @@ -2,23 +2,30 @@
2  using Microsoft.AspNetCore.Mvc;
3  using WebApplication.Data.Models;
4  using WebApplication.Data.Interfaces;
5
6  namespace WebApplication.Controllers
7  {
8      public class HomeController : Controller
9      {
10
11          private readonly IWebAppRepository _repository;
12
13          - public HomeController(IWebAppRepository repository)
14          {
15              _repository = repository;
16          }
17
18          public async Task<IActionResult> Index()
19          {
20              var users = _repository.All< ApplicationUser >();
21
22
23          }
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45

```

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).

```

1  @@ -39,7 +39,7 @@
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45

```

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).

```
1  {
2      "ConnectionStrings": {
3          "DefaultConnection": "Server=(localdb)\\"
4      },
5      "MySpecificSetting": {
6          "ApiKey": "DEV_API_KEY_12345_TEST"
7      }
8  }
```

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:

- Системне (вбудоване):** Надається самим фреймворком.
 - Приклади: app.UseRouting() (визначає, який контролер викликати), app.UseAuthentication() (ідентифікує користувача), app.UseStaticFiles() (роздає CSS, JS, зображення).
- 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).

```
54     + // Configure Rate Limiting
55     + builder.Services.AddRateLimiter(options =>
56     +
57     +     options.OnRejected = (context, _) =>
58     +     {
59     +         context.HttpContext.Response.StatusCode = StatusCodes.Status429TooManyRequests;
60     +         return new ValueTask();
61     +     };
62     +
63     +     options.GlobalLimiter = PartitionedRateLimiter.Create<HttpContext, string>(httpContext =>
64     +     {
65     +         if (httpContext.User.Identity?.IsAuthenticated == true)
66     +         {
67     +             var userId = httpContext.User.FindFirstValue(ClaimTypes.NameIdentifier)!;
68     +             return RateLimitPartition.GetFixedWindowLimiter(userId, _ =>
69     +                 new FixedWindowRateLimiterOptions
70     +                 {
71     +                     PermitLimit = 10,
72     +                     Window = TimeSpan.FromMinutes(1),
73     +                     AutoReplenishment = true // maybe set explicitly
74     +                 });
75     +         }
76     +         else
77     +         {
78     +             var ip = httpContext.Connection.RemoteIpAddress?.ToString() ?? "unknown_ip";
79     +             return RateLimitPartition.GetFixedWindowLimiter(ip, _ =>
80     +                 new FixedWindowRateLimiterOptions
81     +                 {
82     +                     PermitLimit = 5,
83     +                     Window = TimeSpan.FromMinutes(1),
84     +                     AutoReplenishment = true
85     +                 });
86     +         }
87     +     });
88     + });
89     + );
```

Рис. 10 – сервіс AddRateLimiter.

Наочанок, 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).

```
54     // Configure the HTTP request pipeline.
55     @@ -68,8 +107,11 @@
56
57     app.UseRouting();
58
59     app.UseAuthorization();
60
61
62     app.MapControllerRoute(
63         name: "default",
64         pattern: "{controller=Home}/{action=Index}/{id?}");
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93     // Configure the HTTP request pipeline.
```

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 ієархію 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.