МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ

КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМ. ТАРАСА ШЕВЧЕНКА

ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА МЕРЕЖЕВИХ ТА ІНТЕРНЕТ ТЕХНОЛОГІЙ

Лабораторне заняття №4

Тема: АВТЕНТИФІКАЦІЯ ТА АВТОРИЗАЦІЯ КОРИСТУВАЧІВ У ЗАСТОСУНКУ ASP.NET CORE

Виконав студент групи: МІТ-41

Кухарчук Богдан Петрович

Хід роботи

1. Налаштування глобального обмеження доступу

Було налаштовано глобальний фільтр авторизації, який забороняє доступ анонімним користувачам до всіх контролерів за замовчуванням. Для забезпечення доступу до головної сторінки та сторінок входу/реєстрації було використано атрибут [AllowAnonymous]. Також було явно дозволено анонімний доступ до Razor Pages (Identity).

```
// ЛАБ 4 | Завдання 1: Закриття доступу до контролерів за замовчуванням
builder.Services.AddControllersWithViews(options =>
{
    var policy = new AuthorizationPolicyBuilder()
        .RequireAuthenticatedUser()
        .Build();
    options.Filters.Add(new AuthorizeFilter(policy));
});
```

Рис. 4.1. Налаштування глобального фільтра авторизації RequireAuthenticatedUser у Program.cs.

Для забезпечення доступу неавторизованих користувачів до головної сторінки було використано атрибут [AllowAnonymous].

```
[AllowAnonymous]
public async Task<IActionResult> IndexAsync()
{
    var users = await _repository.ReadAll<WebApplicationUser>().ToListAsync();


    return View(users);
}
[AllowAnonymous]
public IActionResult Privacy()
{

    return View();
}
```

Рис. 4.2 Додавання атрибута [AllowAnonymous] для відкриття доступу до HomeController.

2. Реалізація політики на основі тверджень (Claims)

Було зареєстровано політику ArchivePolicy, яка вимагає наявності Claim IsVerifiedClient. Також у цьому блоці коду зареєстровано всі інші політики та хендлери.

```
// НАЛАШТУВАННЯ ПОЛІТИК АВТОРИЗАЦІЇ

builder.Services.AddAuthorization(options =>
{
    // ЛАБ 4 | Завдання 2
    options.AddPolicy("ArchivePolicy", policy =>
        policy.RequireClaim("IsVerifiedClient"));
```

Рис. 4.3 Реєстрація політик авторизації та обробників у Program.cs.

Для тестування політики було модифіковано процес реєстрації: новому користувачу автоматично додаються Claims (IsVerifiedClient, WorkingHours, IsMentor).

```
if (result.Succeeded)
{
    await _userManager.AddClaimAsync(user, new System.Security.Claims.Claim("IsVerifiedClient", "true"));
    await _userManager.AddClaimAsync(user, new System.Security.Claims.Claim("WorkingHours", "1"));
    await _userManager.AddClaimAsync(user, new System.Security.Claims.Claim("IsMentor", "true"));
```

Рис. 4.4 Автоматичне додавання тверджень (Claims) при реєстрації користувача.

Створено контролер ArchiveController, доступ до якого обмежено політикою ArchivePolicy.

```
1  using Microsoft.AspNetCore.Authorization;
2  using Microsoft.AspNetCore.Mvc;
3
4  namespace WebApplication1.Controllers
5  {
6      [Authorize(Policy = "ArchivePolicy")]
7      public class ArchiveController : Controller
8      {
9          public IActionResult Index()
10         {
11             return Content("Ласкаво просимо до Архіву! Тільки для Verified Clients.");
12         }
13     }
14 }
15
```

Рис. 4.5 Захист контролера ArchiveController за допомогою атрибута [Authorize]

3. Реалізація ресурсної авторизації (Resource-based Authorization)

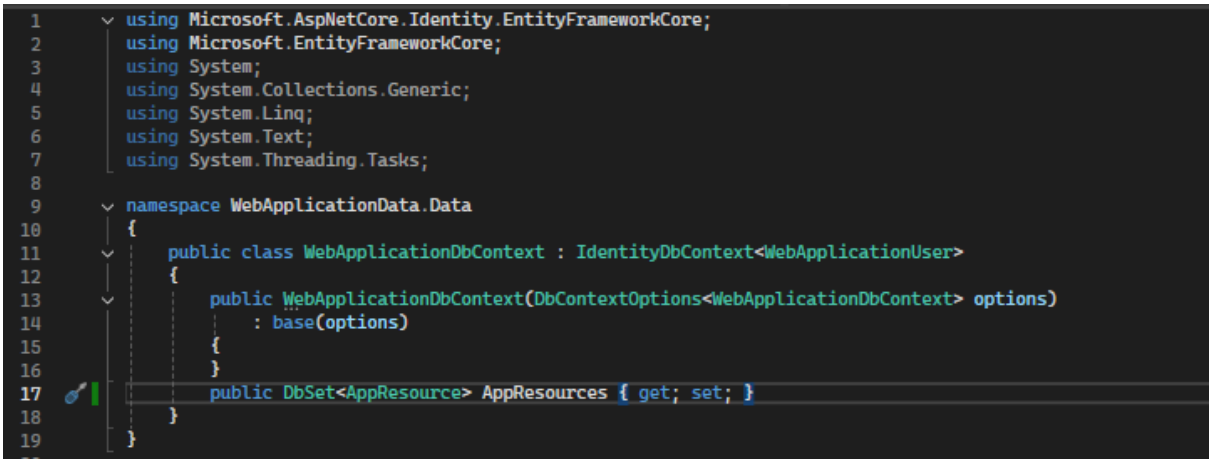
У цьому пункті було вирішено задачу розмежування доступу до конкретних об'єктів даних. Стандартний атрибут [Authorize] не підходить для ситуацій, коли права доступу залежать від властивостей самого об'єкта (наприклад, "редагувати статтю може лише її автор"). Тому було застосовано імперативну авторизацію.

3.1. Створення моделі ресурсу Для демонстрації механізму було створено сутність AppResource. Вона містить поле AuthorId, яке є зовнішнім ключем до таблиці користувачів (AspNetUsers). Це дозволяє чітко визначити власника кожного запису.

```
using System.ComponentModel.DataAnnotations.Schema; // Додайте цей using

namespace WebApplicationData.Data
{
    public class AppResource
    {
        public int Id { get; set; }
        public string Title { get; set; }
        public string AuthorId { get; set; }


        [ForeignKey("AuthorId")]
        public WebApplicationUser Author { get; set; }
    }
}
```

Рис. 4.6 Модель даних AppResource із навігаційною властивістю Author для зв'язку з користувачем.

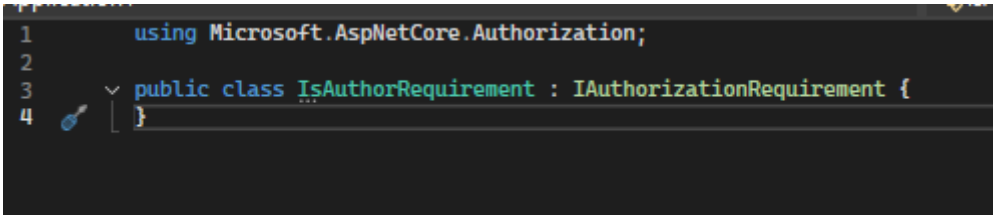
3.2. Оновлення контексту даних Нову модель було зареєстровано в контексті бази даних WebApplicationDbContext через властивість DbSet<AppResource>. Після цього було створено та застосовано міграцію для оновлення структури бази даних.



```
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace WebApplicationData.Data
{
    public class WebApplicationDbContext : IdentityDbContext<WebApplicationUser>
    {
        public WebApplicationDbContext(DbContextOptions<WebApplicationDbContext> options)
            : base(options)
        {
        }
        public DbSet<AppResource> AppResources { get; set; }
    }
}
```

Рис. 4.7 Реєстрація таблиці AppResources у контексті бази даних.

3.3. Створення вимоги (Requirement) У рамках архітектури ASP.NET Core Authorization було створено клас вимоги IsAuthorRequirement. Цей клас реалізує маркерний інтерфейс IAuthorizationRequirement і слугує ідентифікатором для нашої політики перевірки авторства.



```
using Microsoft.AspNetCore.Authorization;

public class IsAuthorRequirement : IAuthorizationRequirement {
}
```

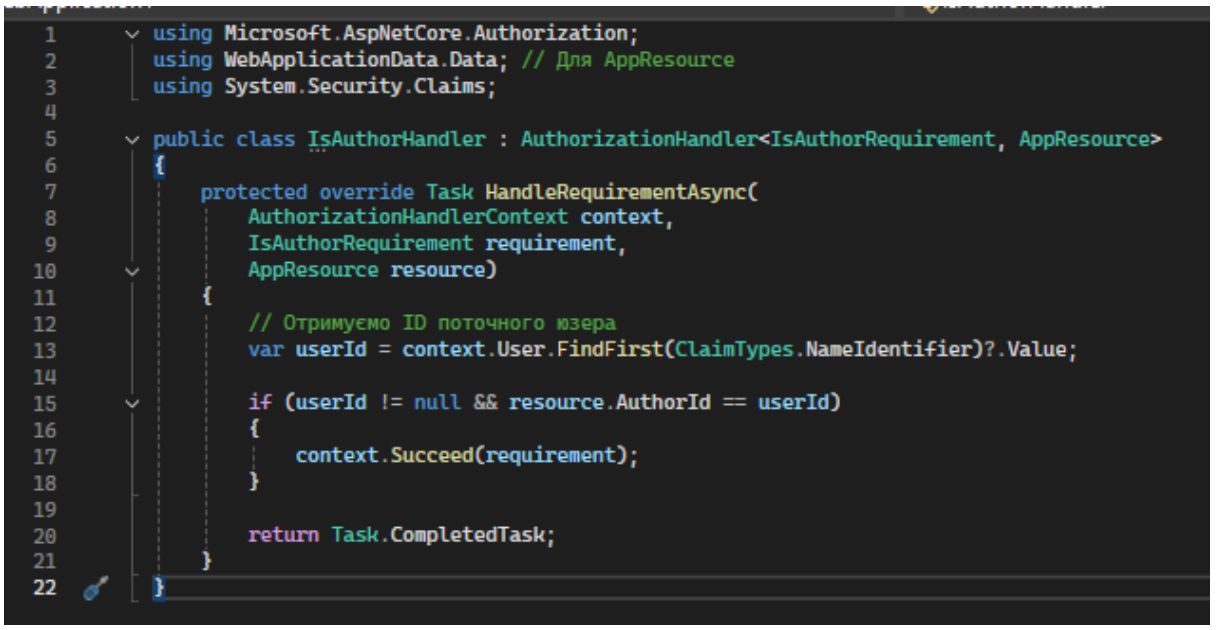
Рис. 4.8 Клас вимоги IsAuthorRequirement.

3.4. Реалізація обробника (Handler) Основну логіку перевірки реалізовано в класі IsAuthorHandler. Цей клас успадковується від AuthorizationHandler<IsAuthorRequirement, AppResource>. У методі HandleRequirementAsync виконується порівняння:

1. Отримується Id поточного користувача з Claims (User.Identity.Name або NameIdentifier).

2. Отримується AuthorId з переданого ресурсу (AppResource).

3. Якщо ідентифікатори збігаються, викликається context.Succeed(), що надає дозвіл.



```
using Microsoft.AspNetCore.Authorization;
using WebApplicationData.Data; // Для AppResource
using System.Security.Claims;

public class IsAuthorHandler : AuthorizationHandler<IsAuthorRequirement, AppResource>
{
    protected override Task HandleRequirementAsync(
        AuthorizationHandlerContext context,
        IsAuthorRequirement requirement,
        AppResource resource)
    {
        // Отримуємо ID поточного юзера
        var userId = context.User.FindFirst(ClaimTypes.NameIdentifier)?.Value;

        if (userId != null && resource.AuthorId == userId)
        {
            context.Succeed(requirement);
        }

        return Task.CompletedTask;
    }
}
```

Рис. 4.9 Логіка порівняння ID користувача та автора ресурсу в IsAuthorHandler.

3.5. Реєстрація сервісів та політики Створений обробник (IsAuthorHandler) було зареєстровано в DI-контейнері як Singleton. Також було налаштовано політику авторизації ResourceOwner, яка використовує вимогу IsAuthorRequirement.

```
builder.Services.AddSingleton<IAuthorizationHandler, IsAuthorHandler>();
```

Рис. 4.10 Реєстрація обробника IsAuthorHandler та політики ResourceOwner.

3.6. Застосування імперативної перевірки в контролері У контролері ResourceController (метод Edit) неможливо використати атрибут [Authorize], оскільки рішення про доступ приймається вже після завантаження даних з бази. Тому було використано сервіс IAuthorizationService. Логіка методу:

1. Завантажується ресурс з бази даних за id.

2. Викликається метод _authorizationService.AuthorizeAsync(User, resource, "ResourceOwner").

3. Якщо результат перевірки негативний (!result.Succeeded), повертається статус 403 Forbid.

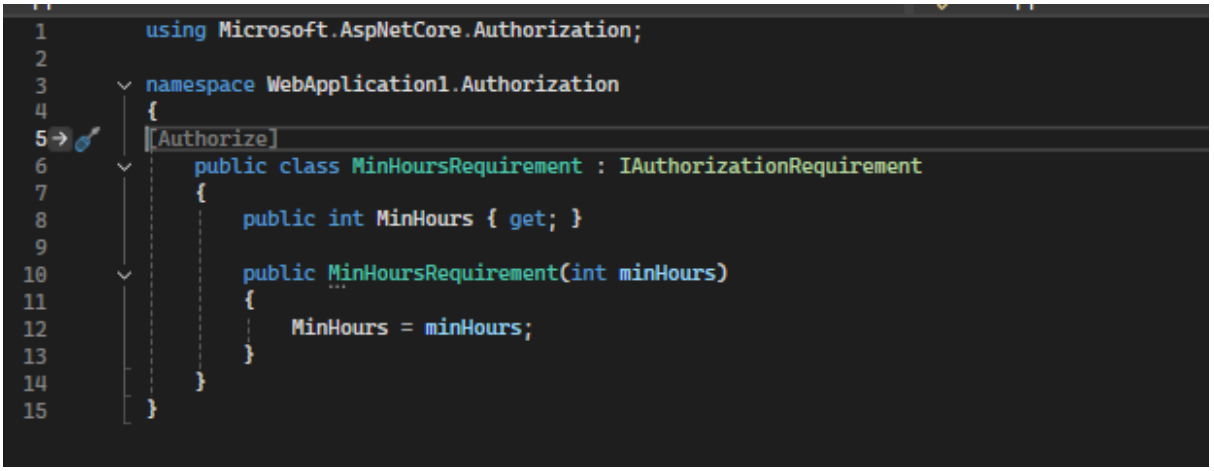
```
        // 5. Логіка редагування (POST)
        [HttpPost]
        public async Task<IActionResult> Edit(AppResource model)
        {
            // Зчитуємо оригінальний ресурс з БД, щоб не втратити AuthorId
            var resource = await _repository.SingleAsync<AppResource>(r => r.Id == model.Id);

            if (resource == null) return NotFound();

            // Повторна перевірка авторизації (безпека понад усе)
            var authorizationResult = await _authorizationService
                .AuthorizeAsync(User, resource, "ResourceOwner");

            if (!authorizationResult.Succeeded)
            {
                return Forbid();
            }

            // Оновлюємо дані
            resource.Title = model.Title;

            await _repository.UpdateAsync(resource);

            return RedirectToAction("Index");
        }
    }
}
```

Рис. 4.11 Імперативний виклик сервісу авторизації для перевірки прав доступу до конкретного ресурсу.

4. Створення кастомної вимоги (MinimumWorkingHours Requirement)

У цьому пункті було реалізовано політику доступу до преміум-контенту, яка базується на бізнес-логіці: доступ дозволено лише працівникам, які відпрацювали певну кількість годин (наприклад, 100 або більше). Для цього було використано механізм Requirements (Вимог) та Handlers (Обробників) в ASP.NET Core.

4.1. Створення класу вимоги (Requirement) Спочатку було створено клас MinHoursRequirement, який реалізує маркерний інтерфейс IAuthorizationRequirement. Цей клас слугує контейнером для даних, необхідних для перевірки політики. У ньому визначено властивість MinHours, яка зберігає порогове значення (у нашому випадку — 100 годин).



```
using Microsoft.AspNetCore.Authorization;

namespace WebApplication1.Authorization
{
[Authorize]
    public class MinHoursRequirement : IAuthorizationRequirement
    {
        public int MinHours { get; }

        public MinHoursRequirement(int minHours)
        {
            MinHours = minHours;
        }
    }
}
```

Рис. 4.12 Клас вимоги MinHoursRequirement, що містить порогове значення годин.

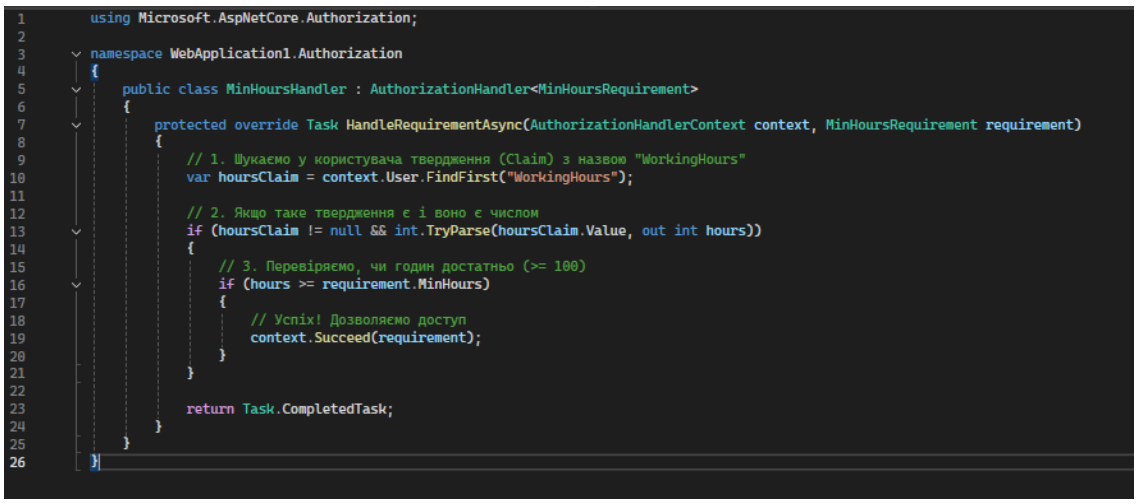
4.2. Реалізація обробника (Handler) Логіку перевірки винесено у клас MinHoursHandler, який успадковується від AuthorizationHandler<MinHoursRequirement>. Алгоритм роботи методу HandleRequirementAsync:

1. З колекції User.Claims вилучається твердження з типом WorkingHours.

2. Значення твердження (яке зберігається як рядок) парситься у ціле число (int).

3. Отримане число порівнюється з вимогою requirement.MinHours.

4. Якщо умова виконується (hours >= 100), викликається метод context.Succeed(), що надає доступ.



```
using Microsoft.AspNetCore.Authorization;

namespace WebApplication1.Authorization
{
    public class MinHoursHandler : AuthorizationHandler<MinHoursRequirement>
    {
        protected override Task HandleRequirementAsync(AuthorizationHandlerContext context, MinHoursRequirement requirement)
        {
            // 1. Шукаємо у користувача твердження (Claim) з назвою "WorkingHours"
            var hoursClaim = context.User.FindFirst("WorkingHours");

            // 2. Якщо таке твердження є і воно є числом
            if (hoursClaim != null && int.TryParse(hoursClaim.Value, out int hours))
            {
                // 3. Перевіряємо, чи годин достатньо (>= 100)
                if (hours >= requirement.MinHours)
                {
                    // Успіх! Дозволяємо доступ
                    context.Succeed(requirement);
                }
            }

            return Task.CompletedTask;
        }
    }
}
```

Рис. 4.13 Реалізація логіки перевірки числового значення Claim у MinHoursHandler.

4.3. Реєстрація у Program.cs Для того щоб механізм запрацював, було виконано налаштування DI-контейнера та системи авторизації:

1. Обробник MinHoursHandler зареєстровано як сервіс із життєвим циклом Singleton.

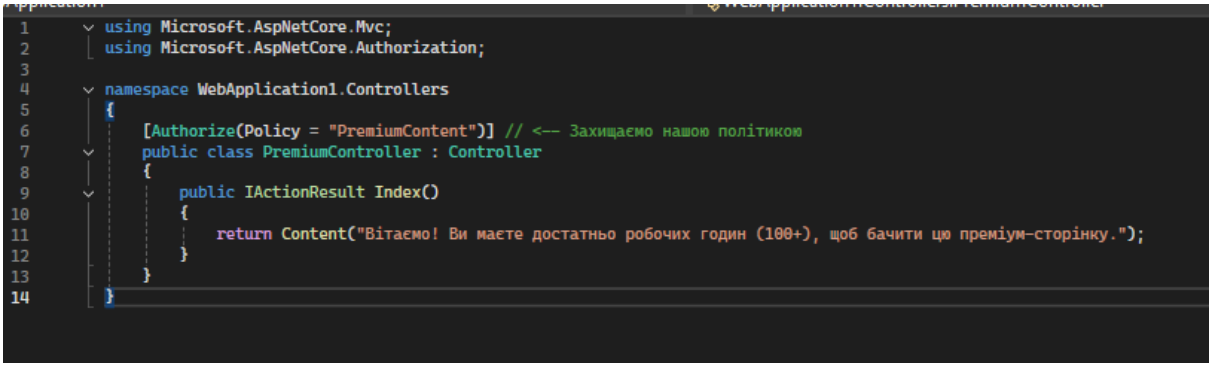
2. Створено політику авторизації з назвою PremiumContent.
    a.
3. До політики додано вимогу MinHoursRequirement зі значенням 100.

```
// ЛАБ 4 | Завдання 4 (Вимога годин)
options.AddPolicy("PremiumContent", policy =>
    policy.Requirements.Add(new MinHoursRequirement(100)));
```

Рис. 4.14 Реєстрація політики PremiumContent та її вимог у Program.cs.

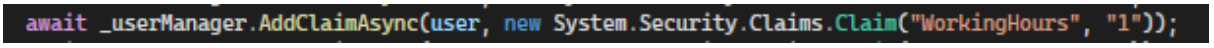
4.4. Захист контролера та надання прав Створено контролер PremiumController, доступ до якого обмежено атрибутом [Authorize(Policy = "PremiumContent")]. Це означає, що перед виконанням будь-якого методу контролера ASP.NET Core запустить наш MinHoursHandler.

Для тестування функціоналу було модифіковано логіку реєстрації (Register.cshtml.cs). Тепер кожному новому користувачу автоматично додається твердження WorkingHours зі значенням 150, що дозволяє успішно пройти перевірку.



```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Authorization;

namespace WebApplication1.Controllers
{
    [Authorize(Policy = "PremiumContent")] // <-- Захищаємо нашою політикою
    public class PremiumController : Controller
    {
        public IActionResult Index()
        {
            return Content("Вітаємо! Ви маєте достатньо робочих годин (100+), щоб бачити цю преміум-сторінку.");
        }
    }
}
```

Рис. 4.15 Контролер PremiumController, захищений кастомною політикою авторизації.



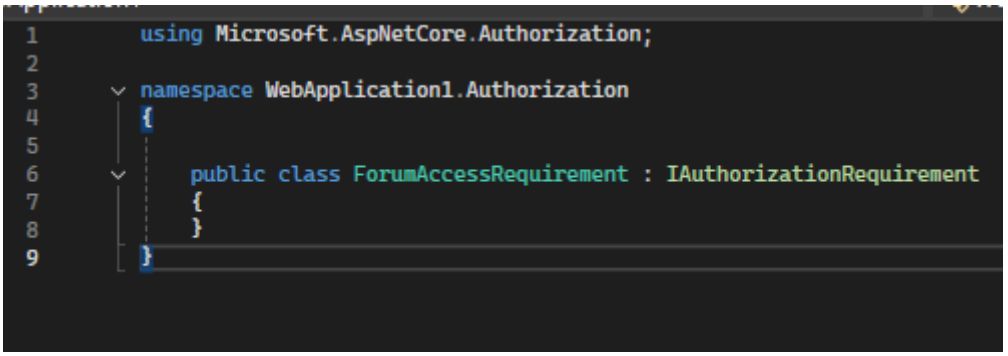
```
await _userManager.AddClaimAsync(user, new System.Security.Claims.Claim("WorkingHours", "1"));
```

Рис. 4.16 Призначення твердження WorkingHours користувачу під час реєстрації.

5. Реалізація складної політики доступу (OR логіка)

У цьому пункті було вирішено задачу надання доступу до розділу "Форум" для різних категорій користувачів. Політика безпеки вимагала, щоб доступ отримували користувачі, які мають хоча б одне з трьох тверджень: IsMentor, IsVerifiedUser або HasForumAccess. Для цього було використано кастомний обробник авторизації.

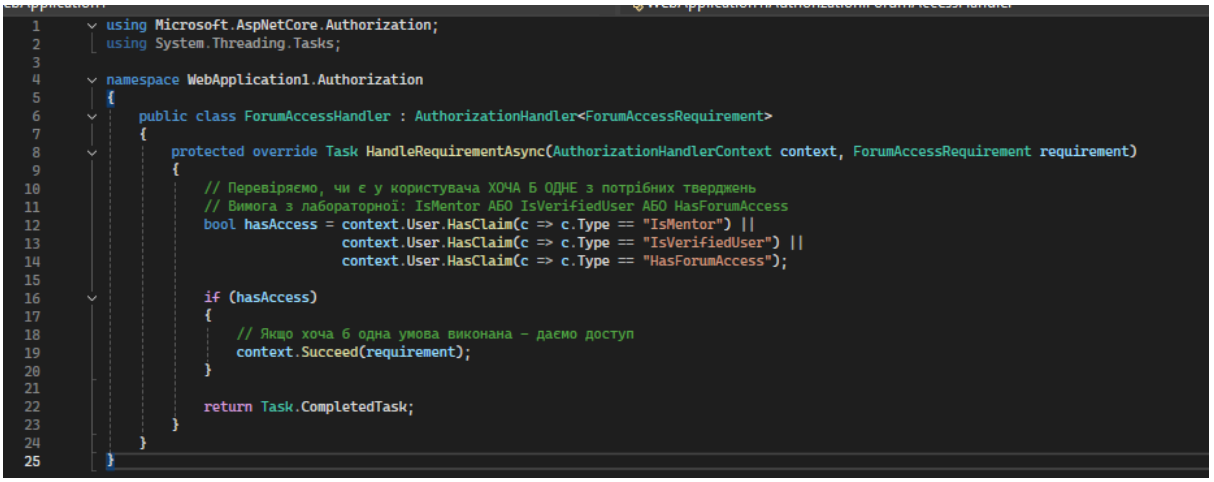
5.1. Створення класу вимоги (Requirement) Було створено клас ForumAccessRequirement, який реалізує інтерфейс IAuthorizationRequirement. У цьому випадку клас не містить властивостей, оскільки він слугує лише ідентифікатором (маркером) для прив'язки нашої специфічної логіки до політики.



```
using Microsoft.AspNetCore.Authorization;

namespace WebApplication1.Authorization
{

    public class ForumAccessRequirement : IAuthorizationRequirement
    {
    }
}
```

Рис. 4.17 Клас-маркер вимоги ForumAccessRequirement.

5.2. Реалізація обробника з логікою "АБО" Основна бізнес-логіка реалізована в класі ForumAccessHandler. В обробнику перевіряється наявність Claims у поточного користувача (context.User). Використано умовний оператор OR (||), щоб перевірити наявність хоча б одного з необхідних типів тверджень. Якщо умова виконується, викликається context.Succeed(requirement), що надає користувачу доступ до ресурсу.



```
using Microsoft.AspNetCore.Authorization;
using System.Threading.Tasks;

namespace WebApplication1.Authorization
{
    public class ForumAccessHandler : AuthorizationHandler<ForumAccessRequirement>
    {
        protected override Task HandleRequirementAsync(AuthorizationHandlerContext context, ForumAccessRequirement requirement)
        {
            // Перевіряємо, чи є у користувача ХОЧА Б ОДНЕ з потрібних тверджень
            // Вимога з лабораторної: IsMentor АБО IsVerifiedUser АБО HasForumAccess
            bool hasAccess = context.User.HasClaim(c => c.Type == "IsMentor") ||
                             context.User.HasClaim(c => c.Type == "IsVerifiedUser") ||
                             context.User.HasClaim(c => c.Type == "HasForumAccess");

            if (hasAccess)
            {
                // Якщо хоча б одна умова виконана – даємо доступ
                context.Succeed(requirement);
            }

            return Task.CompletedTask;
        }
    }
}
```

Рис. 4.18 Реалізація логіки "АБО" для перевірки множинних прав доступу в ForumAccessHandler.

5.3. Реєстрація та налаштування політики У файлі Program.cs було виконано інтеграцію нової логіки в систему:

1. ForumAccessHandler зареєстровано в DI-контейнері.

2. Створено нову політику авторизації з назвою ForumPolicy.

3. До політики додано нашу вимогу ForumAccessRequirement.

```
// ЛАБ 4 | Завдання 5 (OR логіка для форуму)
options.AddPolicy("ForumPolicy", policy =>
    policy.Requirements.Add(new ForumAccessRequirement()));
```

Рис. 4.19 Реєстрація політики ForumPolicy та відповідного обробника.

5.4. Захист контролера та перевірка Створено контролер ForumController, методи якого захищені атрибутом [Authorize(Policy = "ForumPolicy")]. Це гарантує, що жоден користувач без відповідних прав не зможе отримати доступ до методів цього контролера.

Для демонстрації роботи механізму, в логіку реєстрації (Register.cshtml.cs) було додано надання тестового твердження IsMentor. Це дозволяє новоствореному користувачу успішно пройти перевірку "АБО" і отримати доступ до форуму.

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Authorization;

namespace WebApplication1.Controllers
{
    [Authorize(Policy = "ForumPolicy")] // <-- Захищаємо нашою новою політикою
    public class ForumController : Controller
    {
        public IActionResult Index()
        {
            return Content("Ласкаво просимо на Форум! Ви маєте одне з необхідних прав доступу.");
        }
    }
}
```

Рис. 4.20 Контролер ForumController, доступ до якого регулюється політикою ForumPolicy.

```
await _userManager.AddClaimAsync(user, new System.Security.Claims.Claim("IsMentor", "true"));
```

Рис. 4.21 Додавання твердження IsMentor для тестування доступу до форуму.

6. Налаштування Identity та сторінок профілю

У цьому пункті було виконано адаптацію стандартного інтерфейсу керування акаунтом (Identity UI) для роботи з розширеною моделлю користувача WebApplicationUser. Також було додано функціонал для перегляду та редагування особистих даних (імені та прізвища), яких немає у стандартній реалізації.

6.1. Оновлення моделі сторінки профілю Було модифіковано клас IndexModel (Code-behind файл для сторінки профілю). Внесено такі зміни:

1. Замінено всі посилання на IdentityUser класом WebApplicationUser.

2. У внутрішній клас InputModel додано властивості FirstName та LastName з атрибутами валідації ([Required]).

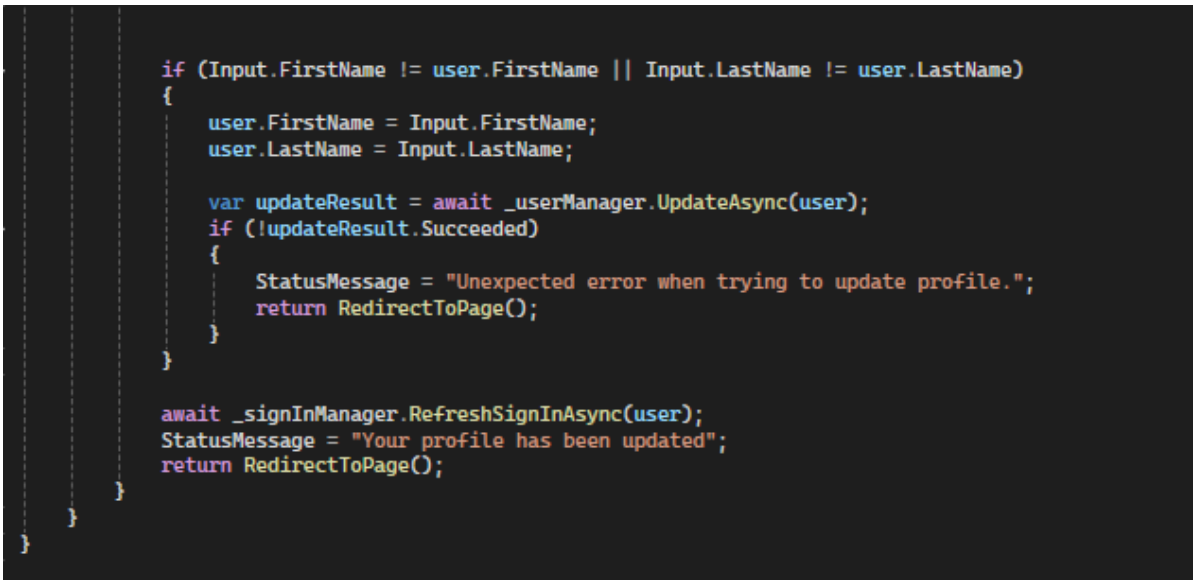
3. У методі LoadAsync додано зчитування поточних значень імені та прізвища з бази даних.

4. У методі OnPostAsync реалізовано логіку збереження змін: якщо введені дані відрізняються від збережених, виконується оновлення користувача через _userManager.UpdateAsync(user).

```
namespace WebApplication1.Areas.Identity.Pages.Account.Manage
{
    public class IndexModel : PageModel
    {
        private readonly UserManager<WebApplicationUser> _userManager;
        private readonly SignInManager<WebApplicationUser> _signInManager;

        public IndexModel(
            UserManager<WebApplicationUser> userManager,
            SignInManager<WebApplicationUser> signInManager)
        {
            _userManager = userManager;
            _signInManager = signInManager;
        }

        public string Username { get; set; }

        [TempData]
        public string StatusMessage { get; set; }

        [BindProperty]
        public InputModel Input { get; set; }

        public class InputModel
        {
            [Phone]
            [Display(Name = "Phone number")]
            public string PhoneNumber { get; set; }


            [Required]
            [Display(Name = "First Name")]
            public string FirstName { get; set; }

            [Required]
            [Display(Name = "Last Name")]
            public string LastName { get; set; }
        }

        private async Task LoadAsync(WebApplicationUser user)
        {
            var userName = await _userManager.GetUserNameAsync(user);
            var phoneNumber = await _userManager.GetPhoneNumberAsync(user);

            Username = userName;

            Input = new InputModel
```

Рис. 4.22 Оновлена модель IndexModel з логікою редагування імені та прізвища.

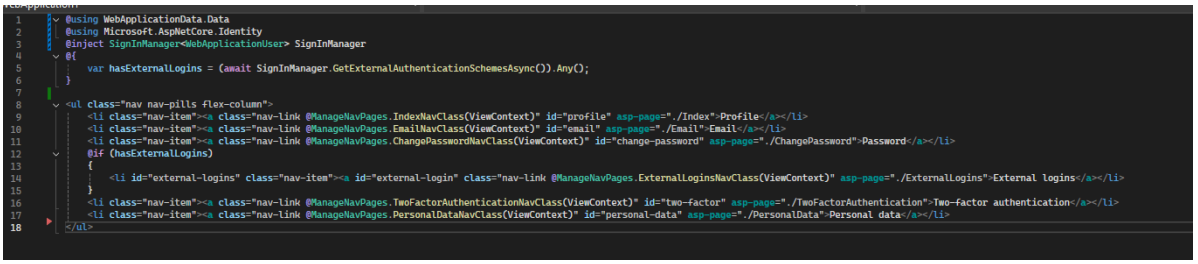
6.2. Оновлення інтерфейсу (View) сторінки профілю На сторінці Index.cshtml було додано HTML-розмітку для нових полів вводу. Використано Tag Helpers (asp-for) для прив'язки полів Input.FirstName та Input.LastName до форми. Це дозволяє користувачеві бачити свої поточні дані та змінювати їх.



```
            if (Input.FirstName != user.FirstName || Input.LastName != user.LastName)
            {
                user.FirstName = Input.FirstName;
                user.LastName = Input.LastName;

                var updateResult = await _userManager.UpdateAsync(user);
                if (!updateResult.Succeeded)
                {
                    StatusMessage = "Unexpected error when trying to update profile.";
                    return RedirectToPage();
                }
            }

            await _signInManager.RefreshSignInAsync(user);
            StatusMessage = "Your profile has been updated";
            return RedirectToPage();
        }
    }
}
```

Рис. 4.23 Додавання полів вводу FirstName та LastName у форму профілю.

6.3. Виправлення типізації у навігації При спробі відкрити будь-яку сторінку керування акаунтом виникала помилка InvalidOperationException через невідповідність типів у сервісах. Це ставалося тому, що часткове представлення навігації _ManageNav намагалося інжектувати SignInManager<IdentityUser>, тоді як у Program.cs зареєстровано SignInManager<WebApplicationUser>. Проблему було вирішено шляхом зміни типу у директиві @inject та додавання відповідного using.



```
@using WebApplicationData.Data
@using Microsoft.AspNetCore.Identity
@inject SignInManager<WebApplicationUser> SignInManager
@{
    var hasExternalLogins = (await SignInManager.GetExternalAuthenticationSchemesAsync()).Any();
}

<ul class="nav nav-pills flex-column">
    <li class="nav-item"><a class="nav-link @ManageNavPages.IndexNavClass(ViewContext)" id="profile" asp-page="./Index">Profile</a></li>
    <li class="nav-item"><a class="nav-link @ManageNavPages.EmailNavClass(ViewContext)" id="email" asp-page="./Email">Email</a></li>
    <li class="nav-item"><a class="nav-link @ManageNavPages.ChangePasswordNavClass(ViewContext)" id="change-password" asp-page="./ChangePassword">Password</a></li>
    @if (hasExternalLogins)
    {
        <li id="external-logins" class="nav-item"><a id="external-login" class="nav-link @ManageNavPages.ExternalLoginsNavClass(ViewContext)" asp-page="./ExternalLogins">External logins</a></li>
    }
    <li class="nav-item"><a class="nav-link @ManageNavPages.TwoFactorAuthenticationNavClass(ViewContext)" id="two-factor" asp-page="./TwoFactorAuthentication">Two-factor authentication</a></li>
    <li class="nav-item"><a class="nav-link @ManageNavPages.PersonalDataNavClass(ViewContext)" id="personal-data" asp-page="./PersonalData">Personal data</a></li>
</ul>
```

Рис. 4.24 Виправлення ін'єкції SignInManager у навігаційному меню профілю.

Результат роботи: Тепер авторизований користувач може безперешкодно заходити у свій профіль, бачити своє ім'я та прізвище, а також успішно їх змінювати. Дані коректно зберігаються у базі даних.

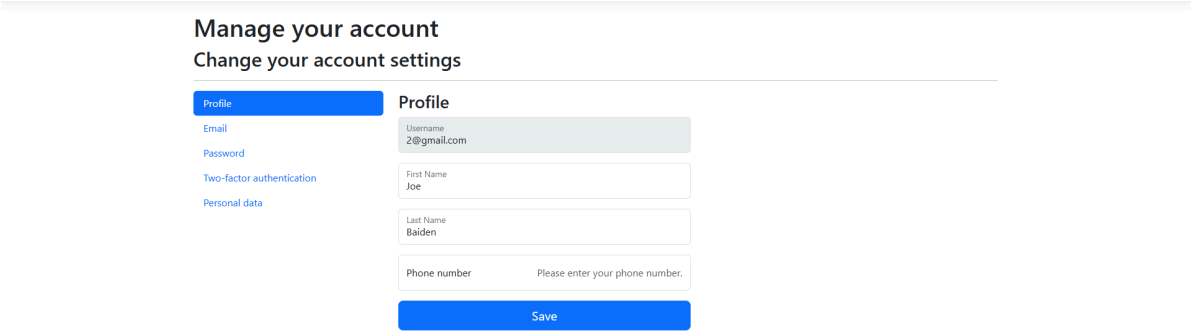


Рис. 4.25 сторінка профілю з робочими полями для редагування даних.

Висновок: У ході виконання лабораторної роботи було успішно реалізовано комплексну систему безпеки для веб-застосунку на базі ASP.NET Core. Було налаштовано механізми автентифікації (встановлення особи) та авторизації (перевірка прав доступу) з використанням ASP.NET Core Identity.

Глобальна безпека: Застосовано глобальний фільтр авторизації, що забороняє анонімний доступ до контролерів за замовчуванням, із використанням атрибута [AllowAnonymous] для публічних сторінок.

Авторизація на основі Claims: Реалізовано політики доступу (Policies), що базуються на наявності специфічних тверджень користувача (IsVerifiedClient), які автоматично призначаються при реєстрації.

Ресурсна авторизація: Імплементовано механізм захисту даних, який дозволяє редагування ресурсу (AppResource) лише його автору. Для цього використано імперативну перевірку через сервіс IAuthorizationService.

Кастомні вимоги (Custom Requirements): Створено власні класи вимог та обробників (Handlers) для реалізації нестандартної логіки: перевірка числового значення (MinimumWorkingHours) та складна логіка "АБО" для доступу до форуму.

Адаптація Identity UI: Стандартні сторінки керування акаунтом було адаптовано для роботи з розширеною моделлю користувача WebApplicationUser, що дозволило реалізувати редагування імені та прізвища.