Лабораторна робота №4

Звіт

З дисципліни “Сучасні інтернет технології”

на тему: “Автентифікація та авторизація користувачів у застосунку ASP.NET core ”.

Студента 4 курсу: Групи МІТ-41
Демиденко Андрій

Київ - 2025р.

Хід виконання роботи

Тема: Автентифікація та авторизація користувачів у застосунку ASP.NET core.
Мета: Мета роботи
Метою роботи було впровадження системи безпеки в ASP.NET Core, охопивши автентифікацію (Identity) та авторизацію (Policies, Claims, Resource-based). Ми мали реалізувати сторінки входу/реєстрації, налаштувати глобальну авторизацію, а також створити складні політики на основі тверджень, включаючи перевірку значень та авторизацію на рівні ресурсів (перевірку авторства).

Забезпечення базової автентифікації користувачів

На першому етапі було реалізовано фундаментальну систему безпеки **ASP.NET Core Identity**. Оскільки в проєкті вже було налаштовано AddDefaultIdentity, основним завданням було додати сторінки інтерфейсу користувача.

Для цього було використано вбудований генератор коду (scaffolder) Visual Studio. Через меню **Add -> New Scaffolded Item -> Identity** було згенеровано сторінки Login, Register, Logout та _LoginPartial (рис. 1). При генерації було явно вказано ApplicationDbContext проєкту WebApplication.Data як контекст даних.

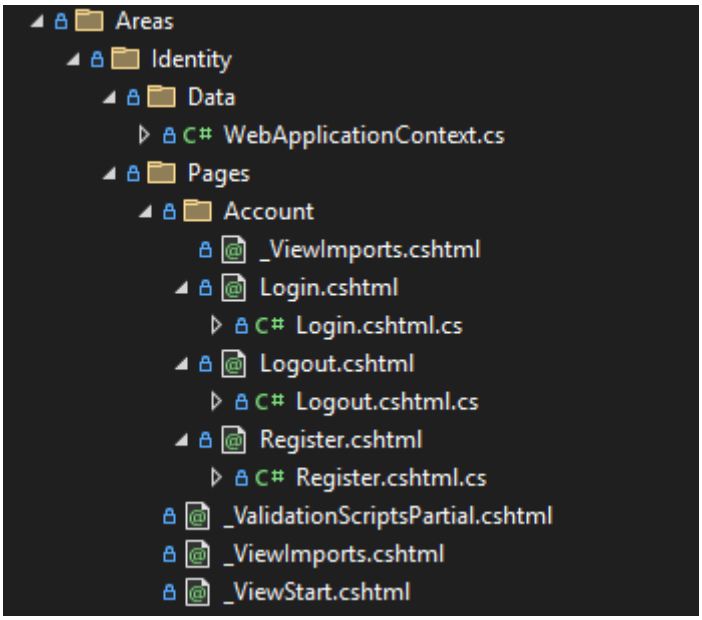

Рис. 1 – Додавання Areas + Identity

Критичним моментом стала адаптація згенерованих файлів. Оскільки наш проєкт (з Л.Р. 1) використовує кастомний клас ApplicationUser (з полями FirstName, LastName), а генератор за замовчуванням використовує базовий IdentityUser, виникла помилка InvalidOperationException: Cannot create a DbSet for 'IdentityUser'.

Цю проблему було вирішено шляхом заміни *всіх* згадок IdentityUser на ApplicationUser у файлах Program.cs (AddDefaultIdentity<ApplicationUser>), Register.cshtml.cs, Login.cshtml.cs, Logout.cshtml.cs та _LoginPartial.cshtml (рис. 2).

```
namespace WebApplication.Areas.Identity.Pages.Account
{
    [AllowAnonymous]
    11 references
    public class RegisterModel : PageModel
    {
        private readonly SignInManager<ApplicationUser> _signInManager;
        private readonly UserManager<ApplicationUser> _userManager;
        private readonly IUserStore<ApplicationUser> _userStore;
        private readonly IUserEmailStore<ApplicationUser> _emailStore;
        private readonly ILogger<RegisterModel> _logger;
        private readonly IEmailSender _emailSender;

        0 references
        public RegisterModel(
            UserManager<ApplicationUser> userManager,
            IUserStore<ApplicationUser> userStore,
            SignInManager<ApplicationUser> signInManager,
            ILogger<RegisterModel> logger,
            IEmailSender emailSender)
        {
            _userManager = userManager;
            _userStore = userStore;
            _emailStore = GetEmailStore();
            _signInManager = signInManager;
            _logger = logger;
            _emailSender = emailSender;
        }
```

Рис. 2 – ApplicationUser.

Також виникла помилка SqlException: Cannot insert the value NULL into column 'FirstName', оскільки наша база даних вимагала ці поля. Щоб реєстрація залишалася простою (лише Email/Пароль), ми оновили модель ApplicationUser, зробивши поля FirstName та LastName необов'язковими (nullable, string?), та застосували нову міграцію (MakeUserNamesNullable) (рис. 3).

```
WebApplication.Data/Models/ApplicationUser.cs                    +2 -2
@@ -4,7 +4,7 @@ namespace WebApplication.Data.Models
 4   {                                                            4   {
 5       public class ApplicationUser : IdentityUser              5       public class ApplicationUser : IdentityUser
 6       {                                                        6       {
 7   -       public string FirstName { get; set; }                7   +       public string? FirstName { get; set; }
 8   -       public string LastName { get; set; }                 8   +       public string? LastName { get; set; }
 9       }                                                        9       }
10   }                                                           10   }
```

Рис. 3 – Оновлення моделі ApplicationUser.

Наостанок, щоб **заблокувати весь сайт** для неавторизованих користувачів, у Program.cs було додано глобальну політику FallbackPolicy, що вимагає автентифікації

для всіх сторінок. Для сторінок, які мають бути публічними (Home/Index, Home/Error, Account/Login, Account/Register), було додано атрибут [AllowAnonymous], що створює виняток із глобального правила (рис. 4)

WebApplication/Controllers/HomeController.cs +4 -1

```
@@ -3,6 +3,7 @@
using WebApplication.Data.Models;
using WebApplication.Data.Interfaces;
using WebApplication.Configuration;
+ using Microsoft.AspNetCore.Authorization;

namespace WebApplication.Controllers
{
@@ -17,7 +18,7 @@ public HomeController(IWebAppRepository repository, WebAppConfiguration config)
            _repository = repository;
            _config = config;
        }
-
+       [AllowAnonymous]
        public Task<IActionResult> Index()
        {
            var users = _repository.All<ApplicationUser>();
@@ -28,12 +29,14 @@ public Task<IActionResult> Index()
            return Task.FromResult<IActionResult>(View(users));
        }

+
        public IActionResult Privacy()
        {
            return View();
        }

        [ResponseCache(Duration = 0, Location = ResponseCacheLocation.None, NoStore = true)]
+       [AllowAnonymous]
        public IActionResult Error()
        {
            return View(new ErrorViewModel { RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier });
```

```
+ // Configure Authorization
+ builder.Services.AddAuthorization(options =>
+ {
+     options.FallbackPolicy = new AuthorizationPolicyBuilder()
+         .RequireAuthenticatedUser()
+         .Build();
+ });
```

Рис. 4 – Додавання FallbackPolicy та надання відповідних атрибутів в контролері.

Створення політики авторизації на основі тверджень (IsVerifiedClient)

Друге завдання полягало у створенні політики, що обмежує доступ до сторінки «Архів» лише для користувачів із певним **твердженням (Claim)**. Це реалізує авторизацію на основі тверджень, де перевіряються не лише дані користувача, а й його повноваження. Спочатку в Program.cs у секції builder.Services.AddAuthorization було додано нову політику з назвою ArchiveAccessPolicy. Ця політика використовує метод RequireClaim, вимагаючи наявності твердження IsVerifiedClient зі значенням "true" (рис. 5).

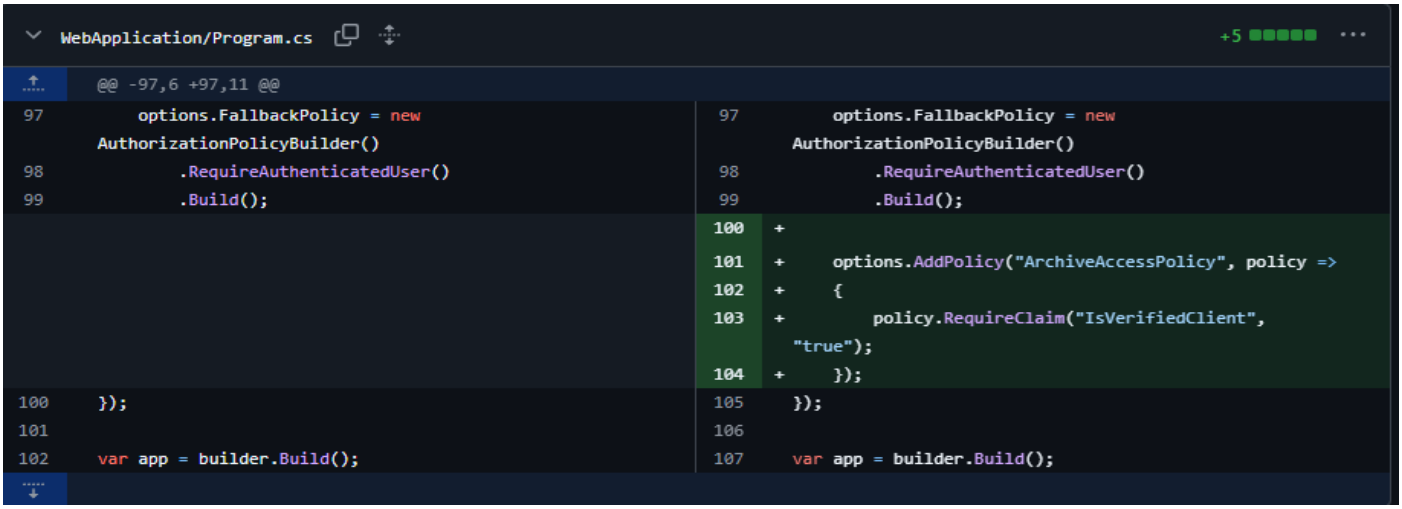

Рис. 5 – Додавання ArchiveAccessPolicy.

Далі, щоб надати користувачам це твердження, було модифіковано логіку реєстрації. У файлі Areas/Identity/Pages/Account/Register.cshtml.cs, у методі OnPostAsync, після успішного створення користувача (if (result.Succeeded)), було додано код для присвоєння нового твердження за допомогою _userManager.AddClaimAsync (рис. 6). Це гарантує, що кожен новий зареєстрований користувач автоматично отримує цей "ключ". Наостанок, було створено новий метод Archive у HomeController та захищено його атрибутом [Authorize(Policy = "ArchiveAccessPolicy")]. Також було створено відповідний View (Archive.cshtml) та додано посилання у _Layout.cshtml для перевірки доступу.

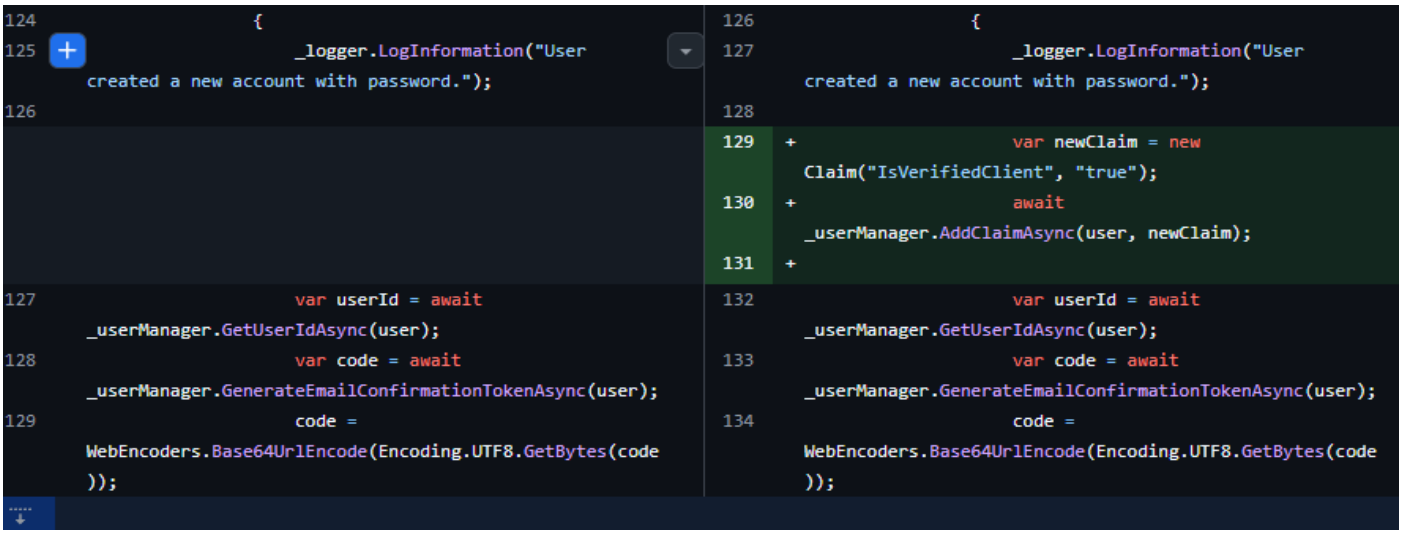

Рис. 6 – _userManager.AddClaimAsync.

Реалізація ресурсної авторизації (IsAuthor)

Це завдання вимагало реалізації найскладнішого типу авторизації — **ресурсної (Resource-based)**. Логіка полягає в тому, що користувач може редагувати ресурс (у нашому випадку Material) **тільки якщо** він є його автором.
Спочатку було створено саму модель Material.cs з ключовою властивістю AuthorId (Foreign Key до AspNetUsers) та додано DbSet<Material> в ApplicationDbContext. Було створено та застосовано міграцію для оновлення бази даних (рис. 7).

```
4   + namespace WebApplication.Data.Models
5   + {
6   +     public class Material
7   +     {
8   +         [Key]
9   +         public int Id { get; set; }
10  +
11  +         [Required]
12  +         public string? Title { get; set; }
13  +
14  +         public string? Content { get; set; }
15  +
16  +         public string? AuthorId { get; set; }
17  +
18  +         [ForeignKey("AuthorId")]
19  +         public virtual ApplicationUser? Author { get;
      set; }
```

Рис. 7 – модель Material.cs.

Далі, було створено інфраструктуру для нової політики. У новій папці WebApplication/Authorization було створено:

1. **Вимогу** IsAuthorRequirement.cs – простий клас-маркер, що реалізує IAuthorizationRequirement.
2. **Обробник** IsAuthorHandler.cs – клас, що реалізує AuthorizationHandler<IsAuthorRequirement, Material>. Саме тут міститься логіка: він порівнює AuthorId отриманого *ресурсу* з ID поточного залогіненого *користувача* (context.User) і викликає context.Succeed(requirement), лише якщо вони збігаються (рис. 8).

```
+ namespace WebApplication.Authorization
+ {
+     public class IsAuthorHandler :
  AuthorizationHandler<IsAuthorRequirement, Material>
+     {
+         protected override Task HandleRequirementAsync(
+             AuthorizationHandlerContext context,
+             IsAuthorRequirement requirement,
+             Material resource)
+         {
+             var userId =
  context.User.FindFirstValue(ClaimTypes.NameIdentifier);
+
+             if (userId != null && resource.AuthorId ==
  userId)
+             {
+                 context.Succeed(requirement);
+             }
+
+             return Task.CompletedTask;
+         }
```

Рис. 8 – AuthorizationHandler<IsAuthorRequirement, Material>.

У Program.cs було зареєстровано нову політику CanManageMaterial (яка вимагає IsAuthorRequirement та сам обробник як сервіс: builder.Services.AddScoped <IAuthorizationHandler, IsAuthorHandler>() (рис. 9).

```
WebApplication/Program.cs                                              +9
@@ -9,6 +9,7 @@
 9   using WebApplication.Data.Interfaces;
 10  using WebApplication.Data.Models;
 11  using WebApplication.Data.Repositories;
 12  + using WebApplication.Authorization;
 12
 13  var builder =
     Microsoft.AspNetCore.Builder.WebApplication.CreateBuild
     er(args);
 14
@@ -102,8 +103,16 @@
 102     {
 103         policy.RequireClaim("IsVerifiedClient",
     "true");
 104     });
 106 +
 107 +   options.AddPolicy("CanManageMaterial", policy =>
 108 +   {
 109 +       policy.AddRequirements(new
     IsAuthorRequirement());
 110 +   });
 105 });
 106
 113 + // Register the authorization handler
 114 + builder.Services.AddScoped<IAuthorizationHandler,
     IsAuthorHandler>();
 115 +
```

Рис. 9 – Додавання CanManageMaterial в Program.cs

Нарешті, було реалізовано **імперативну авторизацію** у контролері MaterialController. Замість атрибута, у метод Edit було "інжектовано" сервіс IAuthorizationService. Перед відображенням сторінки редагування, код завантажує матеріал з БД, а потім явно викликає _authService.AuthorizeAsync(User, material, "CanManageMaterial"). Якщо результат !result.Succeeded, користувач отримує Forbid() (помилку 403) (рис. 10).

```
namespace WebApplication.Controllers
{
    [Authorize]
    1 reference
    public class MaterialController : Controller
    {
        private readonly IWebAppRepository _repository;
        private readonly IAuthorizationService _authService;

        private readonly UserManager<ApplicationUser> _userManager;

        0 references
        public MaterialController(IWebAppRepository repository,
                                  IAuthorizationService authService,
                                  UserManager<ApplicationUser> userManager)
        {
            _repository = repository;
            _authService = authService;
            _userManager = userManager;
        }

        0 references
        public async Task<IActionResult> Index()
        {
            var currentUserId = _userManager.GetUserId(User);

            var materials = await _repository.ReadWhere<Material>(m => m.AuthorId == currentUserId)
                                             .ToListAsync();
            return View(materials);
        }

        0 references
        public async Task<IActionResult> Edit(int id)
        {
            var material = await _repository.ReadSingleAsync<Material>(m => m.Id == id);

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

            var authorizationResult = await _authService.AuthorizeAsync(
                User, material, "CanManageMaterial");

            if (!authorizationResult.Succeeded)
            {

                return Forbid();
            }

            return View(material);
        }

        0 references
        public async Task<IActionResult> CreateTestMaterial()
        {
            var currentUser = await _userManager.GetUserAsync(User);

            var newMaterial = new Material
            {
                Title = $"Матеріал, створений {currentUser.Email}",
                Content = "Цей матеріал створив я.",
                AuthorId = currentUser.Id
            };

            await _repository.AddAsync(newMaterial);

            return Ok($"Створено матеріал з ID: {newMaterial.Id}");
        }
    }
}
```

Рис. 10 – Реалізація імперативної авторизації в контролері Material Controller.

Створення кастомної вимоги (MinimumWorkingHoursRequirement)

Це завдання вимагало створення політики, що перевіряє *значення* твердження. Було створено політику "PremiumAccess", доступну лише тим, у кого твердження WorkingHours має значення > 100. Було створено клас вимоги MinimumWorkingHoursRequirement.cs, який у конструкторі приймає мінімальне значення (100) (рис. 11).

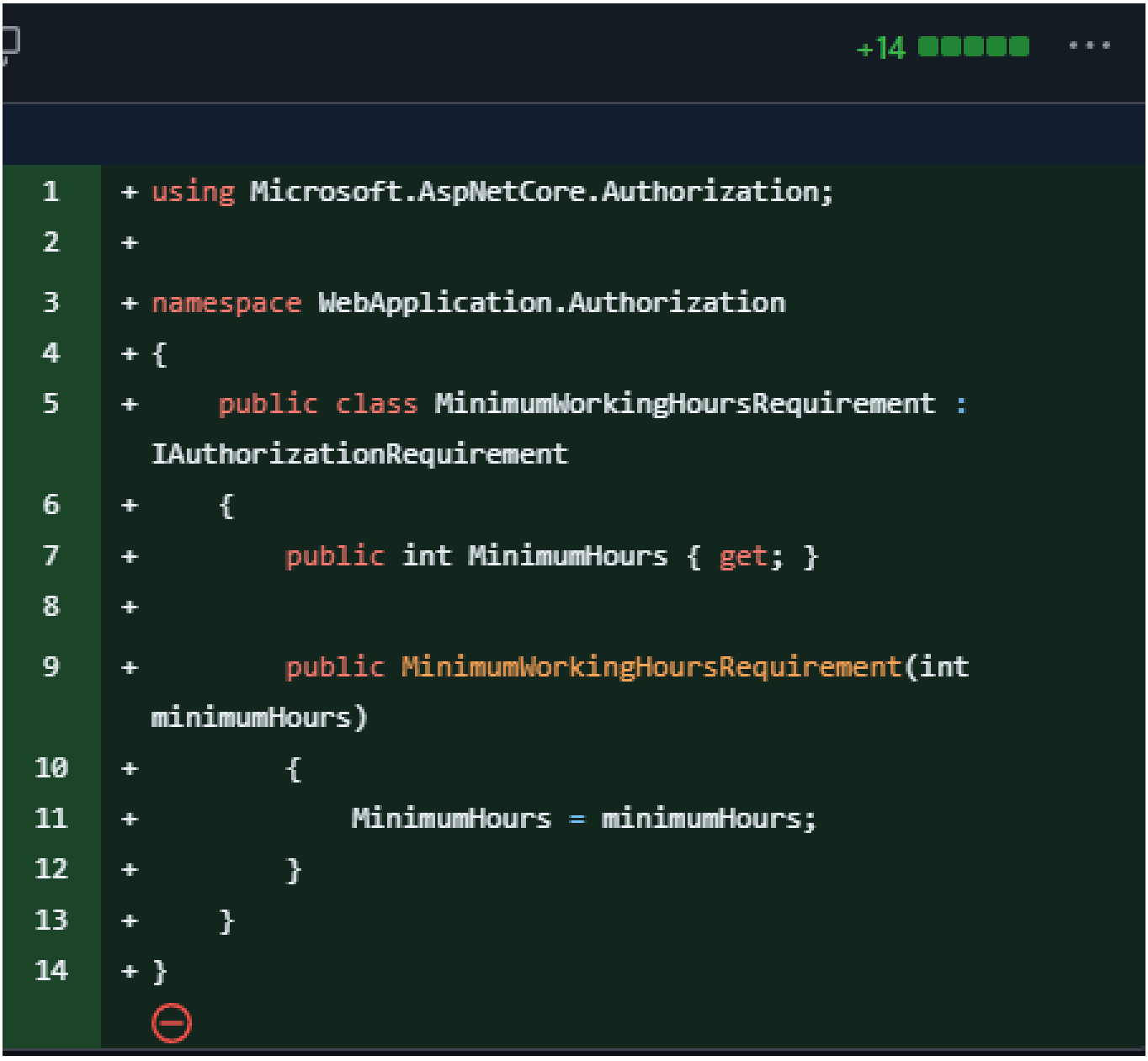

Рис. 11 – Декларативний клас MinimumWorkingHoursRequirement.cs.

Також було створено обробник MinimumWorkingHoursHandler.cs. Його логіка знаходить твердження WorkingHours, намагається перетворити його значення на int, і викликає context.Succeed(), якщо userHours >= requirement.MinimumHours (рис. 12).

```
+ using Microsoft.AspNetCore.Authorization;
+ using System.Security.Claims;
+
+ namespace WebApplication.Authorization
+ {
+     public class MinimumWorkingHoursHandler
+         :
  AuthorizationHandler<MinimumWorkingHoursRequirement>
+     {
+         protected override Task HandleRequirementAsync(
+             AuthorizationHandlerContext context,
+             MinimumWorkingHoursRequirement requirement)
+         {
+             var workingHoursClaim =
  context.User.FindFirst("WorkingHours");
+
+             if (workingHoursClaim == null)
+             {
+                 return Task.CompletedTask;
+             }
+
+             if (int.TryParse(workingHoursClaim.Value,
  out int userHours))
+             {
+                 if (userHours >=
  requirement.MinimumHours)
+                 {
+                     context.Succeed(requirement);
+                 }
+             }
+
+             return Task.CompletedTask;
+         }
+     }
+ }
```

Рис. 12 – обробник MinimumWorkingHoursHandler.cs.

Політика (PremiumAccess) та обробник були зареєстровані у Program.cs, а тестове твердження (WorkingHours, "150") було додано при реєстрації користувача у Register.cshtml.cs.

Створення політики з логікою "АБО" (ForumAccess)

Останнє завдання полягало у створенні політики, що надає доступ за наявності *хоча б одного* з трьох тверджень (IsMentor, IsVerifiedUser або HasForumAccess). Було створено вимогу ForumAccessRequirement та обробник ForumAccessHandler. Ключова логіка в обробнику використовує оператор || (OR) для перевірки context.User.HasClaim(...) для кожного з трьох тверджень. Якщо хоча б одна перевірка істинна, викликається context.Succeed() (рис. 13).

```
+ using Microsoft.AspNetCore.Authorization;
+ using System.Security.Claims;
+
+ namespace WebApplication.Authorization
+ {
+     public class ForumAccessHandler : AuthorizationHandler<ForumAccessRequirement>
+     {
+         protected override Task HandleRequirementAsync(
+             AuthorizationHandlerContext context,
+             ForumAccessRequirement requirement)
+         {
+
+             if (context.User.HasClaim(c => c.Type == "IsMentor") ||
+                 context.User.HasClaim(c => c.Type == "IsVerifiedUser") ||
+                 context.User.HasClaim(c => c.Type == "HasForumAccess"))
+             {
+                 context.Succeed(requirement);
+             }
+
+             return Task.CompletedTask;
+         }
+     }
+ }
```

Рис. 13 – обробник ForumAccessHandler.

Політика (ForumAccess), обробник та тестове твердження (IsMentor) були додані аналогічно до попередніх завдань, а сторінка HomeController.Forum була захищена атрибутом [Authorize(Policy = "ForumAccess")].

Висновок

Усі завдання було виконано. Ми успішно впровадили ASP.NET Core Identity, адаптувавши його під кастомний ApplicationUser та вирішивши проблеми з міграцією (nullable поля). Було налаштовано глобальну FallbackPolicy з винятками [AllowAnonymous]. Ми реалізували три типи політик: просту (на основі твердження IsVerifiedClient), ресурсну (перевірка IsAuthorHandler) та кастомні (перевірка числового значення MinimumWorkingHoursRequirement та логіка "АБО" у ForumAccessHandler). Всі політики та обробники були коректно зареєстровані в Program.cs.