

ASP.NET Core Identity — это мощная система управления пользователями и ролями, встроенная в ASP.NET Core, которая предоставляет готовые механизмы для аутентификации, авторизации и управления данными пользователей. Она обеспечивает работу с регистрацией, входом, управлением ролями, а также поддерживает такие функции, как двухфакторная аутентификация и восстановление пароля.

ASP.NET Core Identity — это система, построенная на принципах модульности и расширяемости. Она включает:

* UserManager: Управление пользователями (создание, удаление, обновление данных).
* SignInManager: Управление процессом входа и выхода пользователей.
* RoleManager: Управление ролями и их привязка к пользователям.
* DbContext: Работа с базой данных через Entity Framework для хранения данных пользователей, ролей, токенов и пр.

UserManager — это один из ключевых компонентов в ASP.NET Core Identity, который предоставляет набор методов для управления пользователями, включая создание, удаление, обновление и другие операции, связанные с учетными записями.

Создать проект ASP.NET Core Empty

В файле appsettings.json добавь строку подключения для базы данных.

"ConnectionStrings": {

  "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=UserManagerDemoDb;Trusted\_Connection=True;MultipleActiveResultSets=true"

}

Настройте Entity Framework Core:

Microsoft.AspNetCore.Identity.EntityFrameworkCore

Microsoft.EntityFrameworkCore.SqlServer

Microsoft.EntityFrameworkCore.Tools

Применение миграций ()

Add-Migration -Context UserContext UserIdentityMigrationName

Update-Database –Context UserContext

Program.cs

using Microsoft.AspNetCore.Identity; // Import ASP.NET Core Identity for authentication and authorization

using Microsoft.EntityFrameworkCore; // Import Entity Framework Core for database access

using mvc\_app.Services; // Import custom services

public class Program

{

    public static void Main(string[] args)

    {

        var builder = WebApplication.CreateBuilder(args); // Create a WebApplicationBuilder for configuring the app

        // Add service products to the DI (Dependency Injection) container

        builder.Services.AddScoped<IServiceProducts, ServiceProducts>();

        // Configure the ProductContext to use SQL Server with a connection string from app settings

        builder.Services.AddDbContext<ProductContext>(options =>

        {

            options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")); // Use SQL Server with the default connection string

        });

        // Configure the UserContext for Identity, using the same connection string

        builder.Services.AddDbContext<UserContext>(options =>

        {

            options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")); // Use SQL Server with the default connection string

        });

        // Configure Identity with custom options for password and sign-in behavior

        builder.Services.AddDefaultIdentity<IdentityUser>(options =>

        {

            // Require email confirmation for login

            options.SignIn.RequireConfirmedEmail = true;

            // Set custom password requirements

            options.Password.RequireDigit = false; // No digit required

            options.Password.RequireNonAlphanumeric = false; // No special characters required

            options.Password.RequiredLength = 4; // Minimum length of 4 characters

            options.Password.RequireUppercase = false; // No uppercase letter required

            options.Password.RequireLowercase = false; // No lowercase letter required

            options.Password.RequiredUniqueChars = 0; // No unique characters required

        })

            .AddRoles<IdentityRole>() // Add role management support to Identity

            .AddEntityFrameworkStores<UserContext>(); // Store Identity data in UserContext with Entity Framework

        builder.Services.AddControllersWithViews(); // Add support for controllers and views (MVC)

        var app = builder.Build(); // Build the app using the configured builder

        app.UseRouting(); // Enable routing for the app

        // Configure the app to use Identity's authentication and authorization

        app.UseAuthentication(); // Enable authentication

        app.UseAuthorization(); // Enable authorization

        app.UseStaticFiles(); // Serve static files (e.g., CSS, JavaScript, images)

        // Configure default routing: home page points to HomeController, Index action

        app.MapControllerRoute(

                name: "default",

                pattern: "{controller=Home}/{action=Index}/{id?}" // Default route pattern

            );

        app.Run(); // Run the application

    }

}

Models/Product.cs

using System.ComponentModel.DataAnnotations.Schema;

using System.ComponentModel.DataAnnotations;

using Microsoft.EntityFrameworkCore;

namespace mvc\_app.Models

{

    public class Product

    {

        [Key]

        [DatabaseGenerated(DatabaseGeneratedOption.Identity)]

        public int Id { get; set; }

        [Required]

        [StringLength(50)]

        public string? Name { get; set; }

        [Required]

        [Precision(10, 2)]

        public decimal Price { get; set; }

        [Required]

        [StringLength(1024)]

        public string? Description { get; set; }

    }

}

UserContext.cs

using Microsoft.AspNetCore.Identity.EntityFrameworkCore; // Импортирует пространство имен для работы с Identity и Entity Framework Core

using Microsoft.EntityFrameworkCore; // Импортирует пространство имен для работы с Entity Framework Core

// Контекст базы данных для пользователей, наследующий от IdentityDbContext

public class UserContext : IdentityDbContext

{

    // Конструктор принимает параметры конфигурации контекста базы данных

    public UserContext(DbContextOptions<UserContext> options)

        : base(options) // Передает параметры в базовый класс

    {

        // Базовый класс IdentityDbContext настроит таблицы для идентификации пользователей

    }

}

ProductContext.cs

using Microsoft.EntityFrameworkCore; // Импортирует пространство имен для работы с Entity Framework Core

using mvc\_app.Models; // Импортирует пространство имен, где находятся модели приложения

// Контекст базы данных для продуктов, наследующий от DbContext

public class ProductContext : DbContext

{

    // Конструктор принимает параметры конфигурации контекста базы данных

    public ProductContext(DbContextOptions<ProductContext> options)

        : base(options) // Передает параметры в базовый класс DbContext

    {

        // Базовый класс DbContext настроит соединение с базой данных и определит таблицы

    }

    // Свойство для доступа к таблице продуктов в базе данных

    public DbSet<Product> Products { get; set; }

}

Services/ServiceProducts.cs

/\*

Add test profucts

 use spr\_211

go

INSERT INTO Products (Name, Price, Description) VALUES ('Apple iPhone 14', 799.99, 'The latest iPhone model with 5G capability, A15 Bionic chip, and improved camera system.');

INSERT INTO Products (Name, Price, Description) VALUES ('Samsung Galaxy S23', 999.99, 'Flagship smartphone from Samsung featuring a dynamic AMOLED display and top-notch performance.');

INSERT INTO Products (Name, Price, Description) VALUES ('Sony WH-1000XM5', 399.99, 'Industry-leading noise-canceling wireless headphones with premium sound quality.');

INSERT INTO Products (Name, Price, Description) VALUES ('Dell XPS 13 Laptop', 1199.99, '13-inch ultra-thin laptop with InfinityEdge display and powerful Intel i7 processor.');

INSERT INTO Products (Name, Price, Description) VALUES ('Apple MacBook Pro 16"', 2499.99, 'High-performance laptop with M1 Pro chip, Retina display, and long battery life.');

INSERT INTO Products (Name, Price, Description) VALUES ('Google Pixel 7', 599.99, 'The latest Google phone with an advanced AI-powered camera and seamless integration with Google services.');

INSERT INTO Products (Name, Price, Description) VALUES ('Sony PlayStation 5', 499.99, 'Next-generation gaming console with 4K gaming and ultra-fast SSD for load times.');

INSERT INTO Products (Name, Price, Description) VALUES ('Microsoft Xbox Series X', 499.99, 'Powerful gaming console with 12 teraflops of processing power and 4K gameplay.');

INSERT INTO Products (Name, Price, Description) VALUES ('Bose QuietComfort Earbuds II', 299.99, 'Premium true wireless earbuds with advanced noise-canceling technology and crisp sound.');

INSERT INTO Products (Name, Price, Description) VALUES ('Apple AirPods Pro 2', 249.99, 'Second-generation wireless earbuds with active noise cancellation and transparency mode.');

INSERT INTO Products (Name, Price, Description) VALUES ('Fitbit Charge 5', 179.99, 'Advanced health and fitness tracker with built-in GPS, heart rate monitor, and stress tracking.');

INSERT INTO Products (Name, Price, Description) VALUES ('Nikon Z7 II Camera', 2999.99, 'Full-frame mirrorless camera with 45.7 MP resolution, dual processors, and 4K video recording.');

INSERT INTO Products (Name, Price, Description) VALUES ('Canon EOS R6', 2499.99, 'Mirrorless camera with 20 MP resolution, advanced autofocus system, and 4K video support.');

INSERT INTO Products (Name, Price, Description) VALUES ('Dyson V15 Detect Vacuum', 749.99, 'Powerful cordless vacuum cleaner with laser dust detection and advanced filtration system.');

INSERT INTO Products (Name, Price, Description) VALUES ('Instant Pot Duo 7-in-1', 99.99, 'Multifunctional pressure cooker with seven cooking modes, including slow cooking and steaming.');

INSERT INTO Products (Name, Price, Description) VALUES ('KitchenAid Stand Mixer', 499.99, 'Classic stand mixer with 10 speeds and durable build, perfect for baking and cooking tasks.');

INSERT INTO Products (Name, Price, Description) VALUES ('GoPro HERO11 Black', 499.99, 'Waterproof action camera with 5.3K video recording, 27 MP photos, and HyperSmooth stabilization.');

INSERT INTO Products (Name, Price, Description) VALUES ('Nintendo Switch OLED', 349.99, 'Hybrid gaming console with a 7-inch OLED screen, detachable controllers, and extensive game library.');

INSERT INTO Products (Name, Price, Description) VALUES ('Logitech MX Master 3', 99.99, 'Wireless ergonomic mouse with precision scrolling, customizable buttons, and multi-device support.');

INSERT INTO Products (Name, Price, Description) VALUES ('Razer DeathAdder V2', 69.99, 'High-precision gaming mouse with 20K DPI sensor, optical switches, and ergonomic design.');

 \*/

using Microsoft.EntityFrameworkCore; // Импорт библиотеки Entity Framework Core для работы с базой данных

using mvc\_app.Models; // Импорт моделей приложения

using System.Collections.Generic; // Импорт для работы с коллекциями

using System.Threading.Tasks; // Импорт для работы с асинхронными методами

using Microsoft.Extensions.Logging; // Импорт для работы с логированием

namespace mvc\_app.Services

{

    // Интерфейс для сервиса продуктов с методами для выполнения операций CRUD

    public interface IServiceProducts

    {

        Task<Product?> CreateAsync(Product? product); // Асинхронный метод для создания продукта

        Task<IEnumerable<Product>> ReadAsync(); // Асинхронный метод для получения всех продуктов

        Task<Product?> GetByIdAsync(int id); // Асинхронный метод для получения продукта по его ID

        Task<Product?> UpdateAsync(int id, Product? product); // Асинхронный метод для обновления продукта

        Task<bool> DeleteAsync(int id); // Асинхронный метод для удаления продукта по ID

    }

    // Реализация интерфейса IServiceProducts

    public class ServiceProducts : IServiceProducts

    {

        private readonly ProductContext \_productContext; // Хранит контекст базы данных для работы с продуктами

        private readonly ILogger<ServiceProducts> \_logger; // Логгер для записи событий и ошибок

        // Конструктор класса, который принимает контекст и логгер через внедрение зависимостей

        public ServiceProducts(ProductContext productContext, ILogger<ServiceProducts> logger)

        {

            \_productContext = productContext; // Инициализация контекста базы данных

            \_logger = logger; // Инициализация логгера

        }

        // Метод для создания нового продукта

        public async Task<Product?> CreateAsync(Product? product)

        {

            // Проверка, является ли продукт нулевым

            if (product == null)

            {

                \_logger.LogWarning("Попытка создать продукт с нулевым значением."); // Логирование предупреждения

                return null; // Возврат нуля, если продукт нулевой

            }

            // Добавление продукта в контекст базы данных

            await \_productContext.Products.AddAsync(product);

            // Сохранение изменений в базе данных

            await \_productContext.SaveChangesAsync();

            return product; // Возврат созданного продукта

        }

        // Метод для удаления продукта по его ID

        public async Task<bool> DeleteAsync(int id)

        {

            // Поиск продукта в базе данных по его ID

            var product = await \_productContext.Products.FindAsync(id);

            // Если продукт не найден, вернуть false

            if (product == null)

            {

                return false;

            }

            // Удаление продукта из контекста базы данных

            \_productContext.Products.Remove(product);

            // Сохранение изменений в базе данных

            await \_productContext.SaveChangesAsync();

            return true; // Возврат true, если продукт успешно удален

        }

        // Метод для получения продукта по его ID

        public async Task<Product?> GetByIdAsync(int id)

        {

            // Поиск продукта в базе данных по его ID

            return await \_productContext.Products.FindAsync(id);

        }

        // Метод для получения всех продуктов из базы данных

        public async Task<IEnumerable<Product>> ReadAsync()

        {

            // Возврат списка всех продуктов

            return await \_productContext.Products.ToListAsync();

        }

        // Метод для обновления существующего продукта

        public async Task<Product?> UpdateAsync(int id, Product? product)

        {

            // Проверка, является ли продукт нулевым или идентификатор не совпадает

            if (product == null || id != product.Id)

            {

                \_logger.LogWarning($"Несоответствие идентификатора продукта. Ожидался {id}, получен {product?.Id}."); // Логирование предупреждения

                return null; // Возврат нуля, если есть несоответствие

            }

            try

            {

                // Обновление продукта в контексте базы данных

                \_productContext.Products.Update(product);

                // Сохранение изменений в базе данных

                await \_productContext.SaveChangesAsync();

                return product; // Возврат обновленного продукта

            }

            catch (DbUpdateConcurrencyException ex)

            {

                // Логирование ошибки при обновлении продукта

                \_logger.LogError(ex, "Ошибка при обновлении продукта с идентификатором {Id}.", id);

                return null; // Возврат нуля в случае ошибки

            }

        }

    }

}

HomeController.cs

using Microsoft.AspNetCore.Mvc; // Импортирует пространство имен для работы с MVC-контроллерами

namespace mvc\_app.Controllers // Определяет пространство имен для контроллеров приложения

{

    // Контроллер для главной страницы

    public class HomeController : Controller

    {

        // Метод действия для обработки запросов на главную страницу

        public ViewResult Index()

        {

            // Возвращает представление по умолчанию (Index.cshtml) для отображения

            return View();

        }

    }

}

ProductsController.cs

using Microsoft.AspNetCore.Authorization; // Import for authorization

using Microsoft.AspNetCore.Mvc; // Import for MVC controller and actions

using mvc\_app.Models; // Import the models (e.g., Product)

using mvc\_app.Services; // Import the service interface for products

namespace mvc\_app.Controllers

{

    public class ProductsController : Controller

    {

        private readonly IServiceProducts \_serviceProducts; // Dependency on product service

        // Constructor to inject the product service dependency

        public ProductsController(IServiceProducts serviceProducts)

        {

            \_serviceProducts = serviceProducts;

        }

        // GET: http://localhost:[port]/products

        // Displays the list of products

        public async Task<ViewResult> Index()

        {

            var products = await \_serviceProducts.ReadAsync(); // Retrieve all products asynchronously

            return View(products); // Return the products to the view

        }

        // GET: http://localhost:[port]/products/details/{id}

        // Displays details of a single product by ID

        public async Task<ViewResult> Details(int id)

        {

            var product = await \_serviceProducts.GetByIdAsync(id); // Retrieve product by ID asynchronously

            return View(product); // Return the product details to the view

        }

        [Authorize(Roles = "admin")] // Restrict access to admin role

        [HttpGet]

        // GET: http://localhost:[port]/products/create

        // Display the product creation form

        public ViewResult Create() => View();

        [Authorize(Roles = "admin")] // Restrict access to admin role

        [HttpPost]

        [ValidateAntiForgeryToken] // Validate the anti-forgery token for security

        // POST: http://localhost:[port]/products/create

        // Handle product creation form submission

        public async Task<IActionResult> Create([Bind("Id,Name,Price,Description")] Product product)

        {

            if (ModelState.IsValid) // Check if the form data is valid

            {

                await \_serviceProducts.CreateAsync(product); // Create the product asynchronously

                return RedirectToAction(nameof(Index)); // Redirect to the product list

            }

            return View(product); // If validation fails, return to the form with the entered data

        }

        [Authorize(Roles = "admin")] // Restrict access to admin role

        [HttpGet]

        // GET: http://localhost:[port]/products/update

        // Display the product update form

        public ViewResult Update() => View();

        [Authorize(Roles = "admin")] // Restrict access to admin role

        [HttpPost]

        [ValidateAntiForgeryToken] // Validate the anti-forgery token for security

        // POST: http://localhost:[port]/products/update/{id}

        // Handle product update form submission

        public async Task<IActionResult> Update(int id, [Bind("Id,Name,Price,Description")] Product product)

        {

            if (ModelState.IsValid) // Check if the form data is valid

            {

                await \_serviceProducts.UpdateAsync(id, product); // Update the product asynchronously

                return RedirectToAction(nameof(Index)); // Redirect to the product list

            }

            return View(product); // If validation fails, return to the form with the entered data

        }

        [Authorize(Roles = "admin")] // Restrict access to admin role

        [HttpGet]

        // GET: http://localhost:[port]/products/delete

        // Display the product delete confirmation form

        public ViewResult Delete() => View();

        [Authorize(Roles = "admin")] // Restrict access to admin role

        [HttpPost]

        [ValidateAntiForgeryToken] // Validate the anti-forgery token for security

        // POST: http://localhost:[port]/products/delete/{id}

        // Handle product deletion

        public async Task<IActionResult> Delete(int id)

        {

            await \_serviceProducts.DeleteAsync(id); // Delete the product asynchronously

            return RedirectToAction(nameof(Index)); // Redirect to the product list

        }

    }

}

UserController.cs

using Microsoft.AspNetCore.Identity; // Импортирует пространство имен для работы с пользователями и аутентификацией

using Microsoft.AspNetCore.Mvc; // Импортирует пространство имен для работы с MVC-контроллерами

namespace mvc\_app.Controllers // Определяет пространство имен для контроллеров приложения

{

    // Контроллер для управления пользователями

    public class UserController : Controller

    {

        private readonly UserManager<IdentityUser> \_userManager; // Менеджер для управления пользователями

        private readonly SignInManager<IdentityUser> \_signInManager; // Менеджер для управления аутентификацией пользователей

private readonly RoleManager<IdentityRole> \_roleManager;

        // Конструктор контроллера, принимает UserManager и SignInManager

       public UserController(

            UserManager<IdentityUser> userManager,

            SignInManager<IdentityUser> signInManager,

            RoleManager<IdentityRole> roleManager

            )

        {

            \_userManager = userManager;

            \_signInManager = signInManager;

            \_roleManager = roleManager;

        }

        // GET: http://localhost:[port]/user/register

        // Метод действия для отображения формы регистрации

        [HttpGet]

        public IActionResult Register()

        {

            return View(); // Возвращает представление для регистрации

        }

        // POST: http://localhost:[port]/user/register

        // Метод действия для обработки регистрации пользователя

        [HttpPost]

        public async Task<IActionResult> Register(string email, string password)

        {

            // Проверка на наличие email и пароля

            if(string.IsNullOrEmpty(email) || string.IsNullOrEmpty(password))

            {

                return BadRequest("Email or password are important ..."); // Возвращает ошибку, если данные не заполнены

            }

            // Создание нового пользователя

            var user = new IdentityUser

            {

                UserName = email, // Установка имени пользователя

                Email = email, // Установка email

                EmailConfirmed = true // Подтверждение email

            };

            // Создание пользователя с использованием UserManager

            var result = await \_userManager.CreateAsync(user, password);

            if (result.Succeeded) // Если регистрация прошла успешно

            {

                return Ok("User is registered ..."); // Возвращает сообщение об успешной регистрации

            }

            // Вывод ошибок, если регистрация не удалась

            foreach (var item in result.Errors)

            {

                Console.WriteLine(item); // Отладочное сообщение

            }

            return BadRequest(Json(result.Errors)); // Возвращает ошибки валидации

        }

        // GET: http://localhost:[port]/user/auth

        // Метод действия для отображения формы аутентификации

        [HttpGet]

        public IActionResult Auth()

        {

            return View(); // Возвращает представление для аутентификации

        }

        // POST: http://localhost:[port]/user/auth

        // Метод действия для обработки аутентификации пользователя

        [HttpPost]

        public async Task<IActionResult> Auth(string email, string password)

        {

            // Проверка на наличие email и пароля

            if (string.IsNullOrEmpty(email) || string.IsNullOrEmpty(password))

            {

                return BadRequest("Email or password are important ..."); // Возвращает ошибку, если данные не заполнены

            }

            // Попытка входа с использованием SignInManager

            var result = await \_signInManager.PasswordSignInAsync(

                    email,

                    password,

                    isPersistent: false, // Указывает, будет ли сессия постоянной

                    lockoutOnFailure: false // Указывает, будет ли блокировка после неудачной попытки входа

                );

            if(result.Succeeded) // Если аутентификация прошла успешно

            {

                return RedirectToAction("Index", "Home"); // Перенаправляет на домашнюю страницу

                //return Ok("Auth OK");

            }

            return BadRequest("Email or password are error ..."); // Возвращает ошибку аутентификации

        }

        // POST: http://localhost:[port]/user/logout

        // Метод действия для выхода пользователя из системы

        [HttpPost]

        public async Task<IActionResult> Logout()

        {

            await \_signInManager.SignOutAsync(); // Выход из системы

            return RedirectToAction("Index", "Home"); // Перенаправляет на домашнюю страницу

        }

        // POST-метод для удаления пользователя по его идентификатору (userId)

        [HttpPost]

        public async Task<IActionResult> Delete(string userId)

        {

            // Находим пользователя по его идентификатору (userId)

            var user = await \_userManager.FindByIdAsync(userId);

            // Если пользователь не найден, возвращаем ошибку "Не найден"

            if (user == null)

            {

                return NotFound("Пользователь не найден."); // Возвращает сообщение об ошибке

            }

            // Удаляем пользователя с помощью UserManager

            var result = await \_userManager.DeleteAsync(user);

            // Если удаление прошло успешно, возвращаем сообщение об успешном удалении

            if (result.Succeeded)

            {

                return Ok("Пользователь удален."); // Возвращает сообщение об успешном удалении

            }

            // Если произошла ошибка, возвращаем сообщение об ошибке

            return BadRequest("Не удалось удалить пользователя."); // Возвращает сообщение об ошибке

        }

        // POST-метод для обновления данных пользователя (email) по его идентификатору (userId)

        [HttpPost]

        public async Task<IActionResult> Update(string userId, string newEmail)

        {

            // Находим пользователя по его идентификатору (userId)

            var user = await \_userManager.FindByIdAsync(userId);

            // Если пользователь не найден, возвращаем ошибку "Не найден"

            if (user == null)

            {

                return NotFound("Пользователь не найден."); // Возвращает сообщение об ошибке

            }

            // Обновляем email и имя пользователя

            user.Email = newEmail; // Установка нового email

            user.UserName = newEmail; // Установка нового имени пользователя

            // Используем UserManager для обновления данных пользователя

            var result = await \_userManager.UpdateAsync(user);

            // Если обновление прошло успешно, возвращаем сообщение об успешном обновлении

            if (result.Succeeded)

            {

                return Ok("Данные пользователя обновлены."); // Возвращает сообщение об успешном обновлении

            }

            // Если произошла ошибка, возвращаем сообщение об ошибке

            return BadRequest("Не удалось обновить данные пользователя."); // Возвращает сообщение об ошибке

        }

[HttpGet]

        public IActionResult CreateRole()

        {

            return View();

        }

        [HttpPost]

        public async Task<IActionResult> CreateRole(string roleName)

        {

            if (string.IsNullOrEmpty(roleName))

            {

                return BadRequest("The role name is important ...");

            }

            var roleExists = await \_roleManager.RoleExistsAsync(roleName);

            if (roleExists)

            {

                return BadRequest("The role name is already  exists ...");

            }

            var role = new IdentityRole

            {

                Name = roleName

            };

            var result = await \_roleManager.CreateAsync(role);

            if(result.Succeeded)

            {

                return Ok("The role is created ...");

            }

            return BadRequest(Json(result.Errors));

        }

        [HttpGet]

        public IActionResult AssignRole()

        {

            return View();

        }

        [HttpPost]

        public async Task<IActionResult> AssignRole(string userId, string roleName)

        {

            if (string.IsNullOrEmpty(userId) || string.IsNullOrEmpty(roleName))

            {

                return BadRequest("User id or role name are error!");

            }

            var user = await \_userManager.FindByIdAsync(userId);

            if (user == null)

            {

                return NotFound("The user not found ...");

            }

            var roleExists = await \_roleManager.RoleExistsAsync(roleName);

            if (!roleExists)

            {

                return BadRequest("The role name is already  exists ...");

            }

            var result = await \_userManager.AddToRoleAsync(user, roleName);

            if (result.Succeeded)

            {

                return Ok("The role is signed ...");

            }

            return BadRequest(Json(result.Errors));

        }

    }

}

Shared/\_Layout.cshtml

@using Microsoft.AspNetCore.Identity

@addTagHelper \*, Microsoft.AspNetCore.Mvc.TagHelpers

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>@ViewData["Title"] - Shop academy step</title>

    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-QWTKZyjpPEjISv5WaRU9OFeRpok6YctnYmDr5pNlyT2bRjXh0JMhjY6hW+ALEwIH" crossorigin="anonymous">

    <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/js/bootstrap.bundle.min.js" integrity="sha384-YvpcrYf0tY3lHB60NNkmXc5s9fDVZLESaAA55NDzOxhy9GkcIdslK1eN7N6jIeHz" crossorigin="anonymous"></script>

</head>

<body>

    <header>

        @{

            if (User.Identity.IsAuthenticated)

            {

                <h1>Hello, @User.Identity.Name</h1>

                <form asp-action="Logout" asp-controller="User" method="post">

                    <button type="submit">exit</button>

                </form>

            }

            else

            {

                <ul>

                    <li><a href="/">main</a></li>

                    <li><a href="/user/register">register</a></li>

                    <li><a href="/user/auth">auth</a></li>

                </ul>

            }

            if(User.IsInRole("admin"))

            {

                <ul class="list-group">

                    <li class="list-group-item">

                        <a asp-action="Create" asp-controller="Products">Create Product</a>

                    </li>

                    <li class="list-group-item">

                        <a asp-action="Update" asp-controller="Products">Update Product</a>

                    </li>

                    <li class="list-group-item">

                        <a asp-action="Delete" asp-controller="Products">Delete Product</a>

                    </li>

                    <li class="list-group-item">

                        <a asp-action="Index" asp-controller="Products">Read Product</a>

                    </li>

                </ul>

            }

            else

            {

                <ul class="list-group">

                    <li class="list-group-item">

                        <a asp-action="Index" asp-controller="Products">Read Product</a>

                    </li>

                </ul>

            }

        }

    </header>

    <main>

        @RenderBody()

    </main>

    <footer>

        <p>&copy; 2024 - BookApp</p>

    </footer>

</body>

</html>

Views/Home/Index.cshtml

@addTagHelper \*, Microsoft.AspNetCore.Mvc.TagHelpers

@{

    Layout = "\_Layout";

    ViewData["Title"] = "Home";

}

<h1>Home</h1>

Products/Create.cshtml

@model mvc\_app.Models.Product

@addTagHelper \*, Microsoft.AspNetCore.Mvc.TagHelpers

@{

    Layout = "\_Layout";

    ViewData["Title"] = "Create Product";

}

<h2>Create Product</h2>

<form asp-action="Create" method="post">

    <div class="form-group">

        <label asp-for="Name">Name</label>

        <input asp-for="Name" class="form-control" />

        <span asp-validation-for="Name" class="text-danger"></span>

    </div>

    <div class="form-group">

        <label asp-for="Price">Price</label>

        <input asp-for="Price" class="form-control" />

        <span asp-validation-for="Price" class="text-danger"></span>

    </div>

    <div class="form-group">

        <label asp-for="Description">Description</label>

        <textarea asp-for="Description" class="form-control"></textarea>

        <span asp-validation-for="Description" class="text-danger"></span>

    </div>

    <button type="submit" class="btn btn-primary">Create</button>

</form>

Products/Update.cshtml

@model mvc\_app.Models.Product

@addTagHelper \*, Microsoft.AspNetCore.Mvc.TagHelpers

@{

    Layout = "\_Layout";

    ViewData["Title"] = "Update Product";

}

<h2>Create Product</h2>

<form asp-action="Update" method="post">

    <div class="form-group">

        <label asp-for="Id">Id</label>

        <input asp-for="Id" class="form-control" />

        <span asp-validation-for="Id" class="text-danger"></span>

    </div>

    <div class="form-group">

        <label asp-for="Name">Name</label>

        <input asp-for="Name" class="form-control" />

        <span asp-validation-for="Name" class="text-danger"></span>

    </div>

    <div class="form-group">

        <label asp-for="Price">Price</label>

        <input asp-for="Price" class="form-control" />

        <span asp-validation-for="Price" class="text-danger"></span>

    </div>

    <div class="form-group">

        <label asp-for="Description">Description</label>

        <textarea asp-for="Description" class="form-control"></textarea>

        <span asp-validation-for="Description" class="text-danger"></span>

    </div>

    <button type="submit" class="btn btn-primary">Update</button>

</form>

Products/Delete.cshtml

@addTagHelper \*, Microsoft.AspNetCore.Mvc.TagHelpers

@{

    Layout = "\_Layout";

    ViewData["Title"] = "Delete Product";

}

<form asp-controller="Products" asp-action="Delete" method="post">

    <div class="form-group">

        <label for="id">Id</label>

        <input for="id" class="form-control" />

        <span for="id" class="text-danger"></span>

    </div>

    <button type="submit" class="btn btn-primary">Delete</button>

</form>

Products/Details.cshtml

@model mvc\_app.Models.Product

@{

    Layout = "\_Layout";

    ViewData["Title"] = "Details Product";

}

<table class="table">

    <thead>

        <tr>

            <th scope="col">Id</th>

            <th scope="col">Name</th>

            <th scope="col">Price</th>

            <th scope="col">Description</th>

        </tr>

    </thead>

    <tbody>

        <tr>

            <td>@Model.Id</td>

            <td>@Model.Name</td>

            <td>@Model.Price</td>

            <td>@Model.Description</td>

        </tr>

    </tbody>

</table>

Products/Index.cshtml

@model IEnumerable<mvc\_app.Models.Product>

@{

    Layout = "\_Layout";

    ViewData["Title"] = "Read Products";

}

    <h2>Products</h2>

    <p>

        <a href="@Url.Action("Create")" class="btn btn-primary">Create New Product</a>

    </p>

<table class="table">

    <thead>

        <tr>

            <th scope="col">Id</th>

            <th scope="col">Name</th>

            <th scope="col">Price</th>

            <th scope="col">Description</th>

            <th scope="col">Tools</th>

        </tr>

    </thead>

    <tbody>

        @foreach(var product in Model)

        {

            <tr>

                <td>@product.Id</td>

                <td>@product.Name</td>

                <td>@product.Price</td>

                <td>@product.Description</td>

                <td>

                    <a href="@Url.Action("Update", new {id = product.Id})" class="btn btn-warning">Update</a>

                    <a href="@Url.Action("Details", new {id = product.Id})" class="btn btn-info">Details</a>

                    <a href="@Url.Action("Delete", new {id = product.Id})" class="btn btn-danger">Delete</a>

                </td>

            </tr>

        }

    </tbody>

</table>

Views/User/Auth.cs

@{

    Layout = "\_Layout";

    ViewData["Title"] = "Auth";

}

<h3>Page auth</h3>

<form action="Auth" method="post">

    <label for="email">Email:</label>

    <input type="email" id="email" name="email" required />

    <label for="password">Password:</label>

    <input type="password" id="password" name="password" required />

    <button type="submit">auth</button>

</form>

Views/User/Register.cs

@{

    Layout = "\_Layout";

    ViewData["Title"] = "Register";

}

<h3>Page register</h3>

<form action="Register" method="post">

    <label for="email">Email:</label>

    <input type="email" id="email" name="email" required />

    <label for="password">Password:</label>

    <input type="password" id="password" name="password" required />

    <button type="submit">register</button>

</form>

Views/User/CreateRole.cs

@{

    Layout = "\_Layout";

    ViewData["Title"] = "Create Role";

}

<h3>Page create role</h3>

<form action="CreateRole" method="post">

    <label for="roleName">roleName:</label>

    <input type="text" id="roleName" name="roleName" required />

    <button type="submit">Create</button>

</form>

Views/User/AssignRole.cs

@{

    Layout = "\_Layout";

    ViewData["Title"] = "Assign Role";

}

<h3>Page Assign Role</h3>

<form action="AssignRole" method="post">

    <label for="userId">UserId:</label>

    <input type="text" id="userId" name="userId" required />

    <label for="roleName">roleName:</label>

    <input type="text" id="roleName" name="roleName" required />

    <button type="submit">Assign</button>

</form>

Добавим API (Для партнеров):

using Microsoft.AspNetCore.Authorization;

using Microsoft.AspNetCore.Identity;

using Microsoft.AspNetCore.Mvc;

using mvc\_app.Models;

using mvc\_app.Services;

using System.Security.Claims;

using System.Threading.Tasks;

namespace mvc\_app.Controllers

{

    [Route("api/[controller]")]

    [ApiController]

    public class APIController : ControllerBase

    {

        private readonly UserManager<IdentityUser> \_userManager;

        private readonly SignInManager<IdentityUser> \_signInManager;

        private readonly RoleManager<IdentityRole> \_roleManager;

        private readonly IServiceProducts \_serviceProducts;

        public APIController(

            UserManager<IdentityUser> userManager,

            SignInManager<IdentityUser> signInManager,

            RoleManager<IdentityRole> roleManager,

            IServiceProducts serviceProducts

        )

        {

            \_userManager = userManager;

            \_signInManager = signInManager;

            \_roleManager = roleManager;

            \_serviceProducts = serviceProducts;

        }

        // ------------------ Управление пользователями ------------------

        // POST: api/API/register

        [HttpPost("register")]

        public async Task<IActionResult> Register(string email, string password)

        {

            if (string.IsNullOrEmpty(email) || string.IsNullOrEmpty(password))

                return BadRequest("Email and password are required.");

            var user = new IdentityUser { UserName = email, Email = email, EmailConfirmed = true };

            var result = await \_userManager.CreateAsync(user, password);

            if (result.Succeeded)

            {

                return Ok("User registered successfully.");

            }

            return BadRequest(result.Errors);

        }

        // POST: api/API/auth

        [HttpPost("auth")]

        public async Task<IActionResult> Auth(string email, string password)

        {

            if (string.IsNullOrEmpty(email) || string.IsNullOrEmpty(password))

                return BadRequest("Email and password are required.");

            var result = await \_signInManager.PasswordSignInAsync(email, password, isPersistent: false, lockoutOnFailure: false);

            if (result.Succeeded)

            {

                return Ok("User authenticated successfully.");

            }

            return BadRequest("Invalid email or password.");

        }

        // POST: api/API/logout

        [HttpPost("logout")]

        public async Task<IActionResult> Logout()

        {

            await \_signInManager.SignOutAsync();

            return Ok("User logged out.");

        }

        // ------------------ Управление ролями ------------------

        // POST: api/API/createRole

        [Authorize(Roles = "admin")]

        [HttpPost("createRole")]

        public async Task<IActionResult> CreateRole(string roleName)

        {

            if (string.IsNullOrEmpty(roleName))

                return BadRequest("Role name is required.");

            var roleExists = await \_roleManager.RoleExistsAsync(roleName);

            if (roleExists)

                return BadRequest($"The role {roleName} already exists.");

            var result = await \_roleManager.CreateAsync(new IdentityRole(roleName));

            if (result.Succeeded)

            {

                return Ok($"Role {roleName} created successfully.");

            }

            return BadRequest(result.Errors);

        }

        // POST: api/API/assignRole

        [Authorize(Roles = "admin")]

        [HttpPost("assignRole")]

        public async Task<IActionResult> AssignRole(string userId, string roleName)

        {

            if (string.IsNullOrEmpty(userId) || string.IsNullOrEmpty(roleName))

                return BadRequest("User ID and role name are required.");

            var user = await \_userManager.FindByIdAsync(userId);

            if (user == null)

                return NotFound("User not found.");

            var roleExists = await \_roleManager.RoleExistsAsync(roleName);

            if (!roleExists)

                return BadRequest($"Role {roleName} does not exist.");

            var result = await \_userManager.AddToRoleAsync(user, roleName);

            if (result.Succeeded)

            {

                return Ok($"Role {roleName} assigned to user {user.UserName}.");

            }

            return BadRequest(result.Errors);

        }

        // ------------------ Управление продуктами ------------------

        // GET: api/API/products

        [Authorize(Roles = "admin,moderator,user")]

        [HttpGet("products")]

        public async Task<IActionResult> GetAllProducts()

        {

            var products = await \_serviceProducts.ReadAsync();

            return Ok(products);

        }

        // GET: api/API/products/{id}

        [Authorize(Roles = "admin,moderator,user")]

        [HttpGet("products/{id}")]

        public async Task<IActionResult> GetProductById(int id)

        {

            var product = await \_serviceProducts.GetByIdAsync(id);

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

            return Ok(product);

        }

        // POST: api/API/products

        [Authorize(Roles = "admin,moderator")]

        [HttpPost("products")]

        public async Task<IActionResult> CreateProduct([FromBody] Product product)

        {

            if (!ModelState.IsValid)

                return BadRequest("Invalid product data.");

            var createdProduct = await \_serviceProducts.CreateAsync(product);

            return Ok(createdProduct);

        }

        // PUT: api/API/products/{id}

        [Authorize(Roles = "admin,moderator")]

        [HttpPut("products/{id}")]

        public async Task<IActionResult> UpdateProduct(int id, [FromBody] Product product)

        {

            if (!ModelState.IsValid)

                return BadRequest("Invalid product data.");

            var updatedProduct = await \_serviceProducts.UpdateAsync(id, product);

            return Ok(updatedProduct);

        }

        // DELETE: api/API/products/{id}

        [Authorize(Roles = "admin")]

        [HttpDelete("products/{id}")]

        public async Task<IActionResult> DeleteProduct(int id)

        {

            var deleted = await \_serviceProducts.DeleteAsync(id);

            return Ok(new { message = deleted });

        }

    }

}

Изменение Program.cs:

using Microsoft.AspNetCore.Authentication.Cookies;

using Microsoft.AspNetCore.Identity;

using Microsoft.EntityFrameworkCore;

using Shop\_app.Services;

namespace Shop\_app

{

    public class Program

    {

        public static void Main(string[] args)

        {

            var builder = WebApplication.CreateBuilder(args);

            builder.Services.AddScoped<IServiceProduct, ServiceProduct>();

            builder.Services.AddDbContext<ProductContext>(options =>

            {

                options.UseSqlServer(

                    builder.Configuration.GetConnectionString("DefaultConnection")

                    );

            });

            builder.Services.AddDbContext<UserContext>(options =>

            {

                options.UseSqlServer(

                    builder.Configuration.GetConnectionString("DefaultConnection")

                    );

            });

            builder.Services.AddSession();

            builder.Services.AddIdentity<IdentityUser, IdentityRole>(options =>

            {

                options.SignIn.RequireConfirmedEmail = true;

                options.Password.RequireNonAlphanumeric = false;

                options.Password.RequireDigit = false;

                options.Password.RequireLowercase = false;

                options.Password.RequireUppercase = false;

                options.Password.RequiredLength = 4;

                options.Password.RequiredUniqueChars = 0;

            })

                .AddEntityFrameworkStores<UserContext>();

            builder.Services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)

            .AddCookie(options =>

            {

                options.ExpireTimeSpan = TimeSpan.FromMinutes(20);

                options.SlidingExpiration = true;

                options.AccessDeniedPath = "/api/APIUser/AccessDenied";

            });

            builder.Services.AddControllersWithViews();

            var app = builder.Build();

            // Configure the HTTP request pipeline.

            if (app.Environment.IsDevelopment())

            {

                app.UseMigrationsEndPoint();

            }

            else

            {

                app.UseExceptionHandler("/Home/Error");

                // 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.UseRouting();

            app.UseHttpsRedirection();

            app.UseAuthentication();

            app.UseAuthorization();

            app.UseStaticFiles();

            app.MapStaticAssets();

            app.MapControllerRoute(

                name: "default",

                pattern: "{controller=Home}/{action=Index}/{id?}")

                .WithStaticAssets();

            app.MapControllers(); //API routing

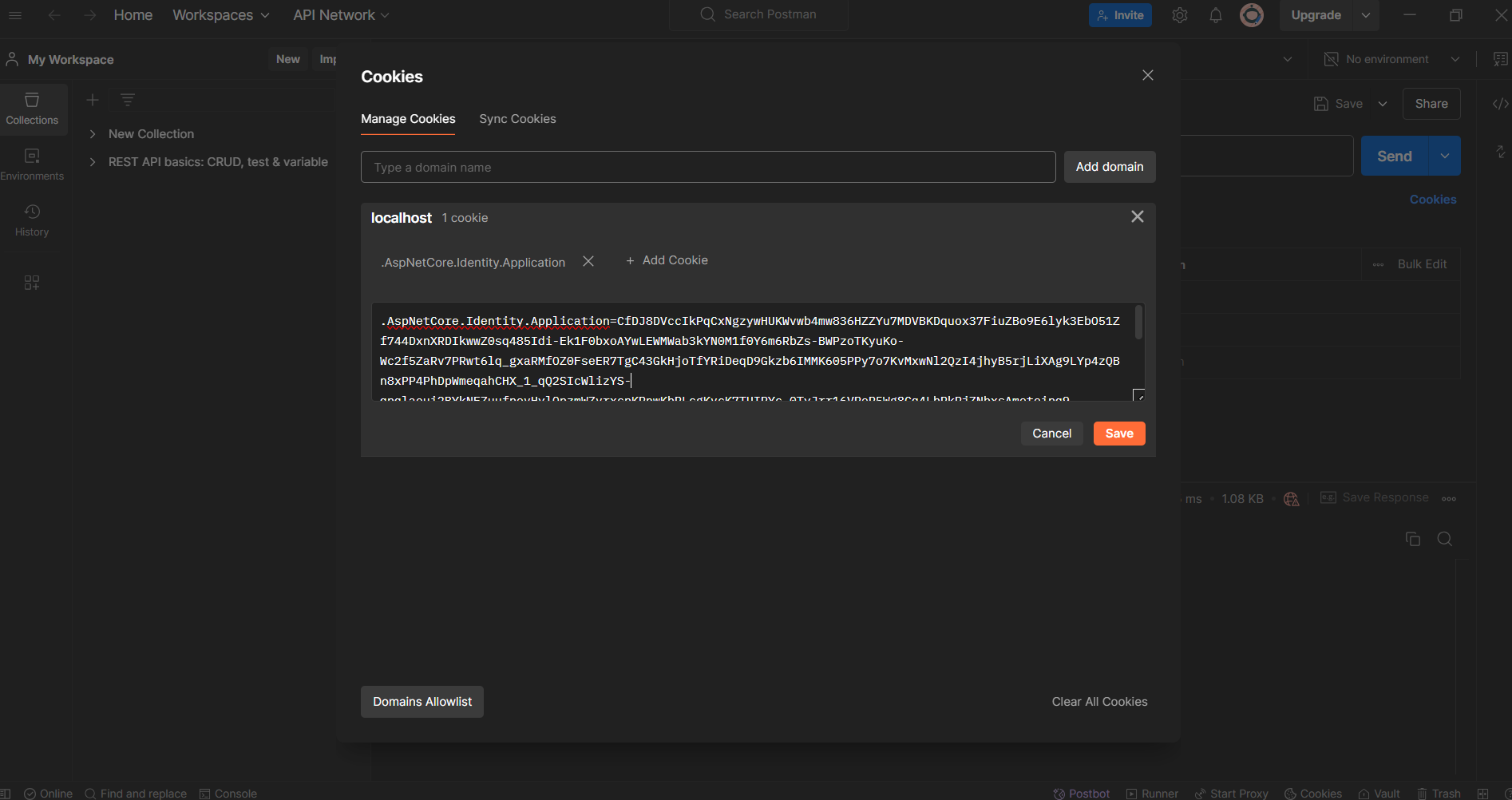
            app.Run();

        }

    }

}

Cookie будут отправлені в Postman:



### 1. ****Регистрация пользователя****

**Метод**: POST  
**URL**: http://localhost:5000/api/API/register  
**Тело запроса (Body)**: x-www-form-urlencoded

{

  "email": "user@example.com",

  "password": "YourPassword123!"

}

**Шаги**:

* В Postman создайте новый запрос.
* Выберите POST.
* Введите URL: http://localhost:5000/api/API/register.
* Перейдите на вкладку Body, выберите x-www-form-urlencoded.
* Добавьте два ключа: email и password, задайте их значения.
* Нажмите Send.

**Ожидаемый результат**: Успешный ответ с текстом "User registered successfully.".

### 2. ****Авторизация пользователя****

**Метод**: POST  
**URL**: http://localhost:5000/api/API/auth  
**Тело запроса (Body)**: x-www-form-urlencoded

{

  "email": "user@example.com",

  "password": "YourPassword123!"

}

**Шаги**:

* Создайте новый запрос.
* Выберите POST.
* Введите URL: http://localhost:5000/api/API/auth.
* Перейдите на вкладку Body, выберите x-www-form-urlencoded.
* Добавьте два ключа: email и password, задайте их значения.
* Нажмите Send.

**Ожидаемый результат**: Успешный ответ с текстом "User authenticated successfully.".

**Важно**: После успешной авторизации в ответе вы получите токен сессии, который Postman автоматически сохраняет для последующих запросов, если включен автоматический Cookie-менеджер.

**Когда вы выполняете авторизацию с помощью метода POST (например, в вашем случае при входе в систему через http://localhost:5000/api/API/auth), ASP.NET Core использует куки** для сохранения информации о сессии пользователя. В Postman автоматически сохраняются эти куки, если включен Cookie-менеджер, что позволяет не вводить токен или другие параметры авторизации вручную в каждом последующем запросе.

Вот как это работает подробнее:

**1. Cookie-менеджер в Postman:**

По умолчанию Postman имеет встроенный Cookie-менеджер, который сохраняет все куки, полученные при запросах. Когда вы логинитесь (авторизуетесь) через POST запрос, сервер возвращает сессионные куки, которые и позволяют серверу понимать, что вы авторизованы в последующих запросах.

**2. Проверка полученных кук:**

После успешной авторизации:

* Перейдите в Postman на вкладку **Cookies** (расположена справа от URL).
* Выберите домен вашего сервера, например, localhost.
* В списке кук вы должны увидеть одну или несколько, таких как .AspNetCore.Identity.Application, содержащих информацию о вашей сессии.

**3. Использование кук в последующих запросах:**

Когда вы отправляете запросы после успешной авторизации (например, для создания продукта или получения списка продуктов), Postman автоматически добавляет эти куки к каждому запросу, что позволяет серверу узнать, что запрос исходит от авторизованного пользователя.

**4. Как проверить и управлять куками в Postman:**

* После отправки запроса авторизации (например, на /auth), вы можете проверить, получил ли Postman куки.
* Нажмите на кнопку **Cookies** рядом с URL.
* Здесь вы увидите список всех кук, связанных с вашим доменом (например, localhost).

**5. Если куки не сохраняются:**

Если по какой-то причине куки не сохраняются, вот несколько вещей, которые стоит проверить:

* Убедитесь, что в настройках **Postman** включена опция Automatically follow redirects (автоматическая обработка перенаправлений).
* Убедитесь, что в ответе сервера приходят куки (их можно увидеть в разделе **Headers** в ответе).

**6. Ручное управление куками:**

Если вы хотите вручную работать с куками:

* Нажмите на **Cookies**.
* Вы можете добавить, изменить или удалить куки для конкретного домена.

**7. Использование токенов вместо кук (опционально):**

Если вы хотите использовать токен вместо куков для аутентификации, вы можете настроить API для работы с **JWT (JSON Web Token)**. Это позволяет отправлять токен авторизации в заголовке Authorization в каждом запросе, но в вашем случае используется куки-сессия.

Таким образом, при последующих запросах, таких как GET на получение списка продуктов или POST для создания продукта, вам не нужно будет добавлять токены вручную — Postman автоматически включит куки сессии в запросы.

### 3. ****Создание роли (например,**** admin****)****

**Метод**: POST  
**URL**: http://localhost:5000/api/API/createRole  
**Тело запроса (Body)**: x-www-form-urlencoded

{

  "roleName": "admin"

}

**Шаги**:

* Авторизуйтесь как администратор (вы должны быть авторизованы через предыдущий шаг).
* Создайте новый запрос.
* Выберите POST.
* Введите URL: http://localhost:5000/api/API/createRole.
* В Body выберите x-www-form-urlencoded, добавьте ключ roleName со значением admin.
* Нажмите Send.

**Ожидаемый результат**: Успешный ответ с текстом "Role admin created successfully.".

### 4. ****Присваивание роли пользователю****

**Метод**: POST  
**URL**: http://localhost:5000/api/API/assignRole  
**Тело запроса (Body)**: x-www-form-urlencoded

{

  "userId": "UserIdHere",

  "roleName": "admin"

}

**Шаги**:

* Найдите ID пользователя, который был зарегистрирован (например, через БД или API управления пользователями).
* Авторизуйтесь как администратор.
* Создайте новый запрос.
* Выберите POST.
* Введите URL: http://localhost:5000/api/API/assignRole.
* В Body выберите x-www-form-urlencoded и добавьте ключи:
  + userId: ID пользователя, которому хотите присвоить роль.
  + roleName: роль, например, admin.
* Нажмите Send.

**Ожидаемый результат**: Успешный ответ с текстом "Role admin assigned to user {user.UserName}.".

**5. Получение всех продуктов**

**Метод**: GET  
**URL**: http://localhost:5000/api/API/products

**Шаги**:

* Авторизуйтесь с ролью user, moderator или admin.
* Создайте новый запрос.
* Выберите GET.
* Введите URL: http://localhost:5000/api/API/products.
* Нажмите Send.

**Ожидаемый результат**: Ответ с JSON-массивом всех продуктов.

**6. Создание продукта**

**Метод**: POST  
**URL**: http://localhost:5000/api/API/products  
**Тело запроса (Body)**: raw (JSON)

{

  "name": "Product Name",

  "description": "Product Description",

  "price": 99.99

}

**Шаги**:

* Авторизуйтесь как admin или moderator.
* Создайте новый запрос.
* Выберите POST.
* Введите URL: http://localhost:5000/api/API/products.
* В Body выберите raw, а затем формат JSON.
* Вставьте JSON-данные продукта.
* Нажмите Send.

**Ожидаемый результат**: Успешный ответ с данными нового продукта.

**7. Обновление продукта**

**Метод**: PUT  
**URL**: http://localhost:5000/api/API/products/{id}  
**Тело запроса (Body)**: raw (JSON)

{

  "name": "Updated Product Name",

  "description": "Updated Product Description",

  "price": 79.99

}

**Шаги**:

* Авторизуйтесь как admin или moderator.
* Создайте новый запрос.
* Выберите PUT.
* Введите URL: http://localhost:5000/api/API/products/{id}, заменив {id} на ID продукта.
* В Body выберите raw, а затем формат JSON.
* Вставьте обновленные данные продукта.
* Нажмите Send.

**Ожидаемый результат**: Успешный ответ с данными обновленного продукта.

**8. Удаление продукта**

**Метод**: DELETE  
**URL**: http://localhost:5000/api/API/products/{id}

**Шаги**:

* Авторизуйтесь как admin.
* Создайте новый запрос.
* Выберите DELETE.
* Введите URL: http://localhost:5000/api/API/products/{id}, заменив {id} на ID продукта.
* Нажмите Send.

**Ожидаемый результат**: Успешный ответ с сообщением об удалении продукта.

**Примечания:**

* **Авторизация**: Чтобы протестировать маршруты с авторизацией, убедитесь, что вы добавили в запрос Authorization с токеном, полученным при авторизации (Postman обычно сохраняет Cookie после успешного входа).
* **Настройка ролей**: Не забудьте создать роли и присвоить их пользователям перед тестированием защищенных маршрутов.

Frontend (Добавим html в wwwroot)

### Шаг 1: Создание HTML-страницы

Создайте новый HTML-файл (например, index.html) и добавьте следующий код:

<!DOCTYPE html>

<html lang="ru">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>API Products</title>

    <!-- Подключаем Bootstrap CSS -->

    <link href="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css" rel="stylesheet">

</head>

<body>

<div class="container mt-5">

    <h1>Управление продуктами</h1>

    <div id="product-form" class="mb-4">

        <h3>Добавить продукт</h3>

        <form id="addProductForm">

            <div class="form-group">

                <label for="productName">Название продукта</label>

                <input type="text" class="form-control" id="productName" required>

            </div>

            <div class="form-group">

                <label for="productPrice">Цена продукта</label>

                <input type="number" class="form-control" id="productPrice" required>

            </div>

            <button type="submit" class="btn btn-primary">Добавить продукт</button>

        </form>

    </div>

    <h3>Список продуктов</h3>

    <ul id="productList" class="list-group">

        <!-- Список продуктов будет добавлен здесь -->

    </ul>

</div>

<!-- Подключаем Bootstrap JS и jQuery -->

<script src="https://code.jquery.com/jquery-3.5.1.slim.min.js"></script>

<script src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.11.6/dist/umd/popper.min.js"></script>

<script src="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/js/bootstrap.min.js"></script>

<script>

    // Функция для получения всех продуктов

    function getProducts() {

        fetch('/api/APIProducts')

            .then(response => response.json())

            .then(data => {

                const productList = document.getElementById('productList');

                productList.innerHTML = ''; // Очищаем список перед обновлением

                data.forEach(product => {

                    const li = document.createElement('li');

                    li.className = 'list-group-item';

                    li.textContent = `${product.name} - $${product.price}`;

                    productList.appendChild(li);

                });

            })

            .catch(error => console.error('Ошибка:', error));

    }

    // Функция для добавления нового продукта

    document.getElementById('addProductForm').addEventListener('submit', function(event) {

        event.preventDefault();

        const productName = document.getElementById('productName').value;

        const productPrice = document.getElementById('productPrice').value;

        fetch('/api/APIProducts', {

            method: 'POST',

            headers: {

                'Content-Type': 'application/json',

                'Authorization': 'Bearer ' + sessionStorage.getItem('token') // Используем токен авторизации

            },

            body: JSON.stringify({ name: productName, price: productPrice })

        })

        .then(response => {

            if (response.ok) {

                getProducts(); // Обновляем список продуктов после добавления

                document.getElementById('addProductForm').reset(); // Очищаем форму

            } else {

                console.error('Ошибка при добавлении продукта:', response.statusText);

            }

        })

        .catch(error => console.error('Ошибка:', error));

    });

    // Инициализация страницы

    getProducts();

</script>

</body>

</html>

**Безапасность проекта**

JWT (JSON Web Tokens) и Bearer-токены — это технологии, которые широко используются для аутентификации и авторизации в веб-приложениях и API.

**JSON Web Token (JWT)** — это компактный, URL-безопасный способ представления данных между двумя сторонами. JWT состоит из трех частей, разделенных точками (.):

1. **Header (Заголовок)**: Указывает, как токен подписан. Обычно это алгоритм, используемый для подписи (например, HMAC SHA256) и тип токена (обычно "JWT").
2. **Payload (Полезная нагрузка)**: Содержит утверждения (claims) о сущности (например, пользователе) и дополнительную информацию. Утверждения могут быть зарегистрированными (например, sub, iat), публичными или частными.
3. **Signature (Подпись)**: Используется для проверки целостности токена и аутентификации отправителя. Она создается путем кодирования заголовка и полезной нагрузки, а затем подписи с использованием секретного ключа или закрытого ключа.

Пример JWT выглядит следующим образом:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV\_adQssw5c

1. **Header**: {"alg":"HS256","typ":"JWT"}
2. **Payload**: {"sub":"1234567890","name":"John Doe","iat":1516239022}
3. **Signature**: Создается с использованием алгоритма HS256.

### Как работает JWT?

1. **Аутентификация**: Пользователь вводит свои учетные данные (например, имя пользователя и пароль) и отправляет их на сервер.
2. **Генерация токена**: Сервер проверяет учетные данные. Если они верные, сервер создает JWT, который включает информацию о пользователе, и отправляет его обратно клиенту.
3. **Отправка токена**: Клиент хранит токен (например, в localStorage или sessionStorage) и добавляет его в заголовок Authorization при последующих запросах к защищенным ресурсам.
4. **Валидация токена**: Сервер проверяет токен на действительность и извлекает информацию о пользователе, чтобы предоставить доступ к запрашиваемым ресурсам.

### Что такое Bearer Token?

**Bearer-токен** — это тип токена, который используется в HTTP-заголовках для авторизации. Он часто используется в сочетании с JWT, но не обязательно. Токены типа Bearer могут представлять собой любые токены, которые выдает сервер.

### Как использовать Bearer Token с JWT?

1. После успешной аутентификации пользователь получает JWT, который будет использоваться как Bearer-токен.
2. При отправке запросов к защищенным ресурсам клиент добавляет токен в заголовок Authorization:

makefile

Copy code

Authorization: Bearer <your\_jwt\_token>

1. Сервер проверяет токен, извлекает данные пользователя и принимает решение о предоставлении или отказе в доступе к ресурсу.

### Преимущества использования JWT и Bearer Tokens

1. **Статус-патент**: Токены не требуют хранения состояния на сервере (stateless), что упрощает масштабирование.
2. **Безопасность**: JWT можно подписывать и шифровать, что обеспечивает безопасность данных.
3. **Гибкость**: Токены могут содержать любые данные, что позволяет легко передавать информацию между клиентом и сервером.

### Недостатки

1. **Размер**: JWT могут быть большими из-за своей структуры, что может негативно повлиять на производительность.
2. **Отзыв токенов**: Поскольку токены являются статeless, их нельзя отозвать до истечения срока действия, если они были скомпрометированы.

Session in asp.net core:

<https://learn.microsoft.com/ru-ru/aspnet/core/fundamentals/app-state?view=aspnetcore-8.0>

Cookie in asp.net core:

<https://learn.microsoft.com/ru-ru/aspnet/core/security/authentication/cookie?view=aspnetcore-8.0>

JWT Tokens in Identity

В современных системах аутентификации и авторизации **JWT (JSON Web Token)** стал одним из наиболее популярных и эффективных способов передачи информации между клиентом и сервером. JWT — это компактный, безопасный для передачи по URL токен, который кодирует данные в виде JSON-объекта. В этом разделе мы рассмотрим теоретические аспекты JWT, а также их практическое применение в C# с использованием фреймворка **ASP.NET Core Identity**.

### 1. ****Структура JWT****

JWT состоит из трёх частей, разделённых точками (.):

1. **Header (заголовок)** — описывает тип токена и используемый алгоритм подписи.
2. **Payload (полезная нагрузка)** — содержит утверждения (claims), представляющие информацию о пользователе и правах доступа.
3. **Signature (подпись)** — обеспечивает целостность токена и защищает его от подделки.

#### Пример JWT:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvbiBEb2UiLCJpYXQiOjE1MTYyMzkwMjJ9.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV\_adQssw5c

1. **Header**: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
2. **Payload**: eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvbiBEb2UiLCJpYXQiOjE1MTYyMzkwMjJ9
3. **Signature**: SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV\_adQssw5c

**2. Header (Заголовок)**

Заголовок JWT включает два ключевых параметра:

* alg — алгоритм подписи. Например, HS256 для HMAC SHA-256.
* typ — тип токена, обычно JWT.

Пример заголовка в формате JSON:

{

  "alg": "HS256",

  "typ": "JWT"

}

Этот JSON кодируется в base64url, что делает его безопасным для передачи по URL.

### 3. ****Payload (Полезная нагрузка)****

Полезная нагрузка содержит **утверждения** (claims), которые передают информацию о пользователе или контексте аутентификации. Существует три типа утверждений:

* **Зарезервированные** (reserved claims) — стандартные утверждения, такие как iss (издатель), exp (время истечения).
* **Публичные** — общие утверждения, например, имя пользователя или идентификатор.
* **Приватные** — утверждения, определяемые пользователем для специфических нужд приложения.

Пример полезной нагрузки:

{

  "sub": "1234567890",

  "name": "John Doe",

  "admin": true,

  "exp": 1516239022

}

Эти данные также кодируются в base64url.

**4. Signature (Подпись)**

Подпись создаётся с использованием заголовка и полезной нагрузки, к которым добавляется секретный ключ. Например, для алгоритма HMAC SHA-256 подпись выглядит так:

HMACSHA256(

  base64UrlEncode(header) + "." + base64UrlEncode(payload),

  secret)

Подпись гарантирует, что токен не был изменён в процессе передачи.

### 5. ****JWT в ASP.NET Core****

JWT интегрируется в ASP.NET Core для реализации механизма аутентификации. Рассмотрим пример настройки JWT-аутентификации в ASP.NET Core.

JWT-токены обеспечивают надёжный и безопасный механизм аутентификации и авторизации в современных веб-приложениях. Они легковесны, просты в использовании и поддерживаются многими фреймворками, включая **ASP.NET Core**. Зная их структуру и способы интеграции в C#, разработчики могут создать масштабируемую и надёжную систему идентификации для своих приложений.

Добавление JWT токенов в наш проект:

Models/LoginModel.cs

namespace mvc\_app.Models

{

    public class LoginModel

    {

        public string Email { get; set; } = string.Empty;

        public string Password { get; set; } = string.Empty;

    }

}

Models/RegisterModel.cs

namespace mvc\_app.Models

{

    public class RegisterModel

    {

        public string Email { get; set; } = string.Empty;

        public string Password { get; set; } = string.Empty;

    }

}

APIAuthController.cs

using Microsoft.AspNetCore.Http;  // Подключение для работы с HTTP запросами

using Microsoft.AspNetCore.Identity;  // Подключение для работы с системой идентификации (UserManager, SignInManager)

using Microsoft.AspNetCore.Mvc;  // Подключение для использования MVC-контроллеров

using Microsoft.IdentityModel.Tokens;  // Подключение для работы с JWT-токенами и их параметрами безопасности

using mvc\_app.Models;  // Подключение моделей, используемых в приложении (например, RegisterModel, LoginModel)

using System.IdentityModel.Tokens.Jwt;  // Подключение для работы с JWT-токенами (создание, генерация, парсинг)

using System.Security.Claims;  // Подключение для работы с утверждениями (claims) в токене

using System.Text;  // Подключение для кодирования строк (например, ключа безопасности)

namespace mvc\_app.Controllers  // Определение пространства имён, в котором находится контроллер

{

    [Route("api/[controller]")]  // Указание маршрута для данного контроллера, API будет доступен по /api/auth

    [ApiController]  // Атрибут, указывающий, что этот класс является API-контроллером

    public class APIAuthController : ControllerBase  // Наследование от базового контроллера API

    {

        private readonly UserManager<IdentityUser> \_userManager;  // Менеджер для работы с пользователями

        private readonly SignInManager<IdentityUser> \_signInManager;  // Менеджер для аутентификации пользователей

        private readonly IConfiguration \_configuration;  // Интерфейс для работы с конфигурацией (например, для получения ключей)

        public APIAuthController(  // Конструктор контроллера, инициализирует менеджеры и конфигурацию

            UserManager<IdentityUser> userManager,

            SignInManager<IdentityUser> signInManager,

            IConfiguration configuration

            )

        {

            \_userManager = userManager;  // Инициализация менеджера пользователей

            \_signInManager = signInManager;  // Инициализация менеджера аутентификации

            \_configuration = configuration;  // Инициализация конфигурации

        }

        // Регистрация нового пользователя

        [HttpPost("register")]  // Определение HTTP POST метода для регистрации по маршруту /api/auth/register

        public async Task<IActionResult> Register([FromBody] RegisterModel model)  // Метод регистрации, принимает модель RegisterModel из тела запроса

        {

            var user = new IdentityUser  // Создание нового объекта пользователя

            {

                UserName = model.Email,  // Установка имени пользователя на основе email

                Email = model.Email,  // Установка email

                EmailConfirmed = true  // Подтверждение email (без фактической верификации в этом примере)

            };

            var result = await \_userManager.CreateAsync(user, model.Password);  // Создание пользователя с указанным паролем

            if (result.Succeeded)  // Проверка, успешно ли создан пользователь

            {

                return Ok(new { message = "User registered successfully" });  // Возврат успешного ответа

            }

            else

            {

                return BadRequest(result.Errors);  // Возврат ошибки, если регистрация не удалась

            }

        }

        // Вход пользователя

        [HttpPost("login")]  // Определение HTTP POST метода для входа по маршруту /api/auth/login

        public async Task<IActionResult> Login([FromBody] LoginModel model)  // Метод для входа пользователя, принимает LoginModel из тела запроса

        {

            var user = await \_userManager.FindByEmailAsync(model.Email);  // Поиск пользователя по email

            if (user == null)  // Если пользователь не найден

            {

                return Unauthorized("Invalid login attempt");  // Возврат ошибки аутентификации

            }

            var result = await \_signInManager.CheckPasswordSignInAsync(user, model.Password, false);  // Проверка пароля пользователя

            if (result.Succeeded)  // Если пароль верен

            {

                var token = GenerateJwtToken(user);  // Генерация JWT-токена для пользователя

                return Ok(new { Token = token });  // Возврат токена в ответе

            }

            return Unauthorized("Invalid login attempt");  // Возврат ошибки, если пароль неверен

        }

        // Генерация JWT-токена для пользователя

        private string GenerateJwtToken(IdentityUser user)

        {

            var tokenHandler = new JwtSecurityTokenHandler();  // Создание обработчика для токенов

            var key = Encoding.ASCII.GetBytes(\_configuration["Jwt:Key"]);  // Получение ключа из конфигурации и его кодирование в байты

            var tokenDescriptor = new SecurityTokenDescriptor  // Определение параметров токена (описание токена)

            {

                Subject = new ClaimsIdentity(new[]  // Добавление утверждений (claims) о пользователе в токен

                {

                    new Claim(ClaimTypes.NameIdentifier, user.Id),  // Утверждение: идентификатор пользователя

                    new Claim(ClaimTypes.Name, user.UserName),  // Утверждение: имя пользователя

                    new Claim(ClaimTypes.Email, user.Email)  // Утверждение: email пользователя

                }),

                Expires = DateTime.UtcNow.AddMinutes(Convert.ToDouble(\_configuration["Jwt:DurationInMinutes"])),  // Установка срока действия токена

                SigningCredentials = new SigningCredentials(new SymmetricSecurityKey(key), SecurityAlgorithms.HmacSha256),  // Подпись токена с использованием симметричного ключа и алгоритма HMAC SHA256 (потрібен пароль у appsettings.json мінімум 32 символи)

                Issuer = \_configuration["Jwt:Issuer"],  // Установка издателя токена (опционально)

                Audience = \_configuration["Jwt:Audience"]  // Установка аудитории токена (опционально)

            };

            var token = tokenHandler.CreateToken(tokenDescriptor);  // Создание токена на основе описания

            return tokenHandler.WriteToken(token);  // Возврат сгенерированного токена в строковом формате

        }

    }

}

APIProductsController.cs

using Microsoft.AspNetCore.Authentication.JwtBearer;  // Подключение для работы с аутентификацией на основе JWT

using Microsoft.AspNetCore.Authorization;  // Подключение для работы с авторизацией в контроллере

using Microsoft.AspNetCore.Http;  // Подключение для работы с HTTP запросами

using Microsoft.AspNetCore.Mvc;  // Подключение для работы с MVC контроллерами

using Microsoft.Extensions.Logging;  // Подключение для работы с логированием

using mvc\_app.Models;  // Подключение модели данных (Product)

using mvc\_app.Services;  // Подключение сервисов, которые обрабатывают логику работы с продуктами

using System.Threading.Tasks;  // Подключение для асинхронных методов

namespace mvc\_app.Controllers  // Пространство имён для контроллера

{

    [Route("api/[controller]")]  // Маршрут контроллера, API будет доступен по /api/products

    [ApiController]  // Атрибут, указывающий, что этот класс является API-контроллером

    [Authorize(AuthenticationSchemes = JwtBearerDefaults.AuthenticationScheme)]  // Авторизация с использованием схемы JWT Bearer

    public class APIProductsController : ControllerBase  // Контроллер, работающий с продуктами

    {

        private readonly IServiceProducts \_serviceProducts;  // Сервис для работы с продуктами

        private readonly ILogger<APIProductsController> \_logger;  // Логгер для записи информации о действиях

        public APIProductsController(IServiceProducts serviceProducts, ILogger<APIProductsController> logger)  // Конструктор, инициализирующий сервис и логгер

        {

            \_serviceProducts = serviceProducts;  // Инициализация сервиса

            \_logger = logger;  // Инициализация логгера

        }

        // Получить все продукты

        [HttpGet]  // Метод будет обрабатывать HTTP GET запросы

        public async Task<IActionResult> GetAllProducts()  // Асинхронный метод для получения всех продуктов

        {

            \_logger.LogInformation("Fetching all products.");  // Логирование запроса на получение всех продуктов

            var products = await \_serviceProducts.ReadAsync();  // Асинхронное получение списка продуктов

            return Ok(products);  // Возвращаем список продуктов в ответе с кодом 200 OK

        }

        // Получить продукт по ID

        [HttpGet("{id}")]  // Метод будет обрабатывать HTTP GET запросы по маршруту с указанием ID продукта

        public async Task<IActionResult> GetProductById(int id)  // Асинхронный метод для получения продукта по ID

        {

            \_logger.LogInformation($"Fetching product with ID: {id}");  // Логирование запроса на получение продукта по ID

            var product = await \_serviceProducts.GetByIdAsync(id);  // Асинхронное получение продукта по ID

            if (product == null)  // Проверка, существует ли продукт с таким ID

            {

                \_logger.LogWarning($"Product with ID {id} not found.");  // Логирование, что продукт не найден

                return NotFound();  // Возвращаем ответ 404 Not Found, если продукт не найден

            }

            return Ok(product);  // Возвращаем найденный продукт в ответе с кодом 200 OK

        }

        // Создать новый продукт

        [HttpPost]  // Метод будет обрабатывать HTTP POST запросы

        public async Task<IActionResult> CreateProduct([FromBody] Product product)  // Асинхронный метод для создания нового продукта

        {

            if (product == null)  // Проверка, что объект продукта не равен null

            {

                \_logger.LogError("CreateProduct: Product object is null.");  // Логирование ошибки при отсутствии продукта

                return BadRequest("Product object is null.");  // Возвращаем ответ 400 Bad Request, если объект продукта отсутствует

            }

            \_logger.LogInformation("Creating a new product.");  // Логирование запроса на создание нового продукта

            var productCreated = await \_serviceProducts.CreateAsync(product);  // Асинхронное создание продукта

            return CreatedAtAction(nameof(GetProductById), new { id = productCreated.Id }, productCreated);  // Возвращаем ответ 201 Created с данными о созданном продукте

        }

        // Обновить продукт

        [HttpPut("{id}")]  // Метод будет обрабатывать HTTP PUT запросы для обновления продукта по ID

        public async Task<IActionResult> UpdateProduct(int id, [FromBody] Product product)  // Асинхронный метод для обновления продукта

        {

            if (product == null)  // Проверка, что объект продукта не равен null

            {

                \_logger.LogError("UpdateProduct: Product object is null.");  // Логирование ошибки при отсутствии продукта

                return BadRequest("Product object is null.");  // Возвращаем ответ 400 Bad Request, если объект продукта отсутствует

            }

            \_logger.LogInformation($"Updating product with ID: {id}");  // Логирование запроса на обновление продукта по ID

            var productUpdated = await \_serviceProducts.UpdateAsync(id, product);  // Асинхронное обновление продукта

            if (productUpdated == null)  // Проверка, существует ли продукт для обновления

            {

                \_logger.LogWarning($"Product with ID {id} not found for update.");  // Логирование, что продукт не найден для обновления

                return NotFound();  // Возвращаем ответ 404 Not Found, если продукт не найден

            }

            return Ok(productUpdated);  // Возвращаем обновленный продукт в ответе с кодом 200 OK

        }

        // Удалить продукт

        [HttpDelete("{id}")]  // Метод будет обрабатывать HTTP DELETE запросы для удаления продукта по ID

        public async Task<IActionResult> DeleteProduct(int id)  // Асинхронный метод для удаления продукта по ID

        {

            \_logger.LogInformation($"Deleting product with ID: {id}");  // Логирование запроса на удаление продукта по ID

            var deleted = await \_serviceProducts.DeleteAsync(id);  // Асинхронное удаление продукта

            if (!deleted)  // Проверка, был ли продукт удалён

            {

                \_logger.LogWarning($"Product with ID {id} not found for deletion.");  // Логирование, что продукт не найден для удаления

                return NotFound();  // Возвращаем ответ 404 Not Found, если продукт не найден

            }

            return Ok(new { message = "Product deleted successfully." });  // Возвращаем успешный ответ с сообщением об удалении

        }

    }

}

Appsettings.json

{

  "Logging": {

    "LogLevel": {

      "Default": "Information",

      "Microsoft": "Warning",

      "Microsoft.AspNetCore": "Warning"

    }

  },

  "ConnectionStrings": {

    "DefaultConnection": "Server=DESKTOP-EK8GAKM;Database=spr\_211;Integrated Security=True; TrustServerCertificate=True;"

  },

  "Jwt": {

    "Key": "A1B2C3D4E5F6G7H8I9J0K1L2M3N4O5P6",

    "Issuer": "https://localhost:7255",

    "Audience": "https://localhost:7255",

    "DurationInMinutes": 30

  },

  "AllowedHosts": "\*"

}

Program.cs

using Microsoft.AspNetCore.Identity; // Подключение библиотеки для работы с системой идентификации пользователей (ASP.NET Core Identity)

using Microsoft.EntityFrameworkCore; // Подключение библиотеки для работы с Entity Framework Core (для взаимодействия с базой данных)

using Microsoft.AspNetCore.Authentication.JwtBearer; // Подключение библиотеки для аутентификации с использованием JWT Bearer токенов

using Microsoft.IdentityModel.Tokens; // Подключение библиотеки для работы с JWT токенами и их проверкой

using mvc\_app.Services; // Подключение пользовательских сервисов, которые реализуются в приложении (например, IServiceProducts)

using System.Text; // Подключение библиотеки для работы с текстом, включая кодировку

using Microsoft.AspNetCore.Authentication.Cookies; // Подключение библиотеки для аутентификации с использованием cookies

using Microsoft.Extensions.Logging; // Подключение библиотеки для логирования

public class Program

{

    public static void Main(string[] args)

    {

        var builder = WebApplication.CreateBuilder(args); // Создание экземпляра WebApplicationBuilder для настройки приложения

        // Настройка логирования

        builder.Logging.ClearProviders(); // Очищает все провайдеры логирования

        builder.Logging.AddConsole(); // Добавляет логирование в консоль

        builder.Logging.AddDebug(); // Добавляет логирование отладочной информации

        // Добавление сервисов в контейнер внедрения зависимостей (DI)

        builder.Services.AddScoped<IServiceProducts, ServiceProducts>(); // Регистрируем сервис продуктов (DI-контейнер)

        // Настройка контекста базы данных для продуктов

        builder.Services.AddDbContext<ProductContext>(options =>

        {

            options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")); // Используем SQL Server для взаимодействия с базой данных

        });

        // Настройка контекста базы данных для пользователей

        builder.Services.AddDbContext<UserContext>(options =>

        {

            options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")); // Используем SQL Server для базы данных пользователей

        });

        // Конфигурация системы идентификации (Identity)

        builder.Services.AddIdentity<IdentityUser, IdentityRole>(options =>

        {

            options.SignIn.RequireConfirmedEmail = true; // Требовать подтверждения email при входе

            options.Password.RequireDigit = false; // Не требовать цифр в пароле

            options.Password.RequireNonAlphanumeric = false; // Не требовать специальных символов в пароле

            options.Password.RequiredLength = 4; // Минимальная длина пароля - 4 символа

            options.Password.RequireUppercase = false; // Не требовать заглавных букв

            options.Password.RequireLowercase = false; // Не требовать строчных букв

            options.Password.RequiredUniqueChars = 0; // Не требовать уникальных символов в пароле

        })

        .AddEntityFrameworkStores<UserContext>() // Указываем, что данные пользователей хранятся в контексте UserContext

        .AddDefaultTokenProviders(); // Добавляем провайдеры токенов по умолчанию

        // Политика CORS (разрешение запросов с определенного источника)

        builder.Services.AddCors(options =>

        {

            options.AddPolicy("AllowSpecificOrigin", // Политика, разрешающая доступ только с указанного адреса

                builder =>

                {

                    builder.WithOrigins("http://127.0.0.1:5500") // Разрешаем запросы с локального адреса

                           .AllowAnyHeader() // Разрешаем любые заголовки

                           .AllowAnyMethod(); // Разрешаем любые методы HTTP

                });

        });

        // Аутентификация с использованием Cookie и JWT

        builder.Services.AddAuthentication(options =>

        {

            options.DefaultScheme = CookieAuthenticationDefaults.AuthenticationScheme; // Схема аутентификации по умолчанию - Cookie

            options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme; // Схема вызова аутентификации - JWT

        })

        .AddCookie(options => // Настройка аутентификации через Cookie

        {

            options.LoginPath = "/Account/Login"; // Путь для входа в систему

            options.AccessDeniedPath = "/Account/AccessDenied"; // Путь при отказе в доступе

        })

        .AddJwtBearer(options => // Настройка аутентификации через JWT токены

        {

            options.TokenValidationParameters = new TokenValidationParameters

            {

                ValidateIssuer = true, // Валидация издателя токена

                ValidateAudience = true, // Валидация аудитории токена

                ValidateLifetime = true, // Валидация времени жизни токена

                ValidateIssuerSigningKey = true, // Проверка подписи токена

                ValidIssuer = builder.Configuration["Jwt:Issuer"], // Указываем допустимого издателя

                ValidAudience = builder.Configuration["Jwt:Audience"], // Указываем допустимую аудиторию

                IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(builder.Configuration["Jwt:Key"])) // Устанавливаем ключ для подписи токена

            };

            // Настройка событий JWT для логирования

            options.Events = new JwtBearerEvents

            {

                OnAuthenticationFailed = context =>

                {

                    var logger = context.HttpContext.RequestServices.GetRequiredService<ILogger<Program>>(); // Получаем логгер

                    logger.LogError("Authentication failed: {Message}", context.Exception.Message); // Логируем ошибку аутентификации

                    return Task.CompletedTask; // Завершаем задачу

                },

                OnTokenValidated = context =>

                {

                    var logger = context.HttpContext.RequestServices.GetRequiredService<ILogger<Program>>(); // Получаем логгер

                    logger.LogInformation("Token validated successfully."); // Логируем успешную проверку токена

                    return Task.CompletedTask; // Завершаем задачу

                }

            };

        });

        // Добавление поддержки авторизации

        builder.Services.AddAuthorization(); // Включение системы авторизации

        // Добавление поддержки MVC и API контроллеров

        builder.Services.AddControllersWithViews(); // Добавляем поддержку контроллеров с представлениями (MVC)

        var app = builder.Build(); // Строим и конфигурируем приложение

        // Middleware pipeline (Конвейер промежуточного ПО)

        app.UseRouting(); // Включение маршрутизации

        // Включите CORS перед вызовом UseAuthentication и UseAuthorization

        app.UseCors("AllowAll"); // Включение CORS с политикой "AllowAll" (разрешает все источники)

        app.UseAuthentication(); // Включение аутентификации

        app.UseAuthorization(); // Включение авторизации

        // Serve static files (Обслуживание статических файлов)

        app.UseStaticFiles(); // Включение поддержки статических файлов

        // Default MVC route (Маршрут по умолчанию для MVC)

        app.MapControllerRoute(

            name: "default",

            pattern: "{controller=Home}/{action=Index}/{id?}"); // Определение маршрута по умолчанию

        // API Controllers route (Маршрутизация для API контроллеров)

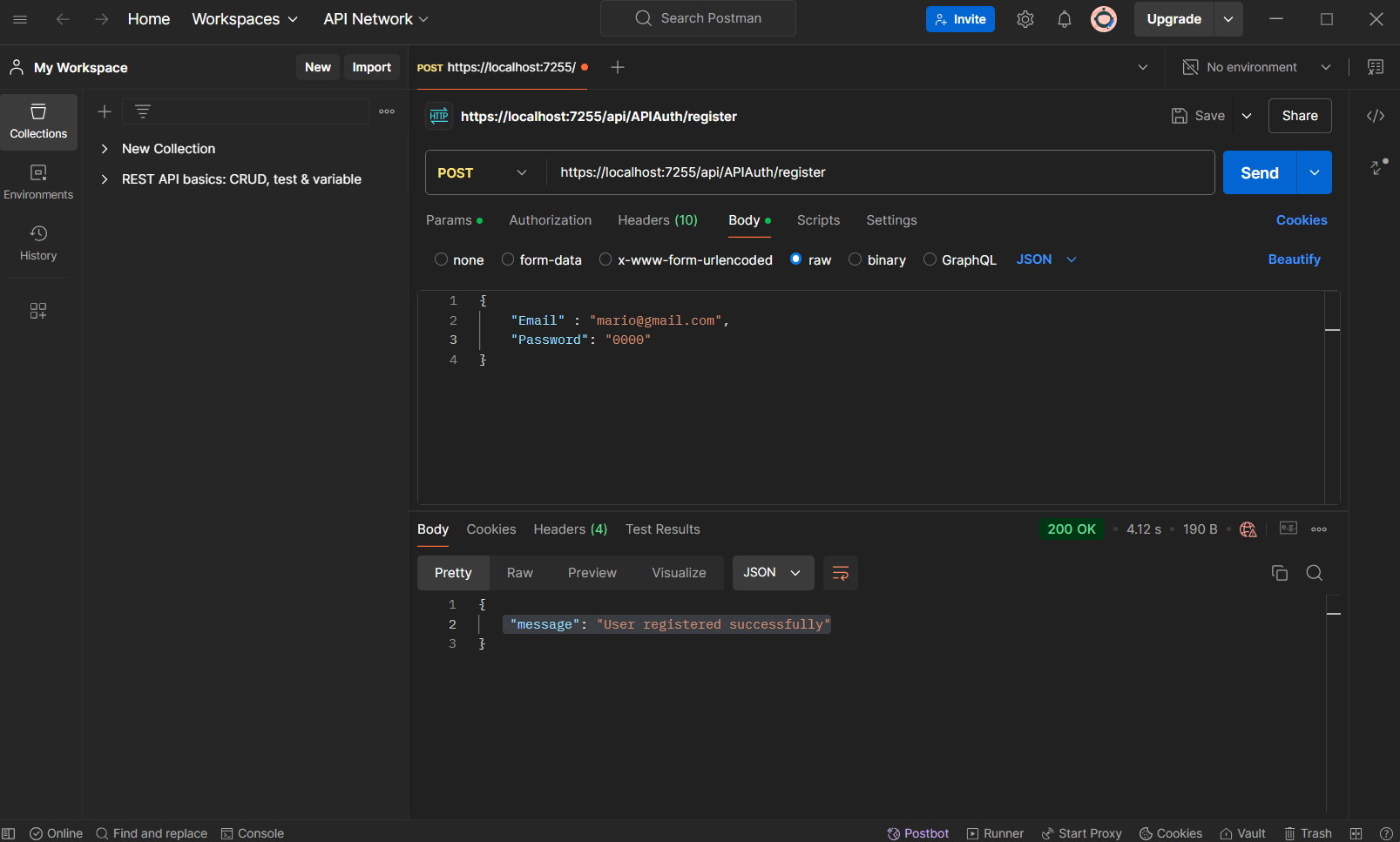
app.MapControllers(); // Подключаем маршрутизацию для API контроллеров

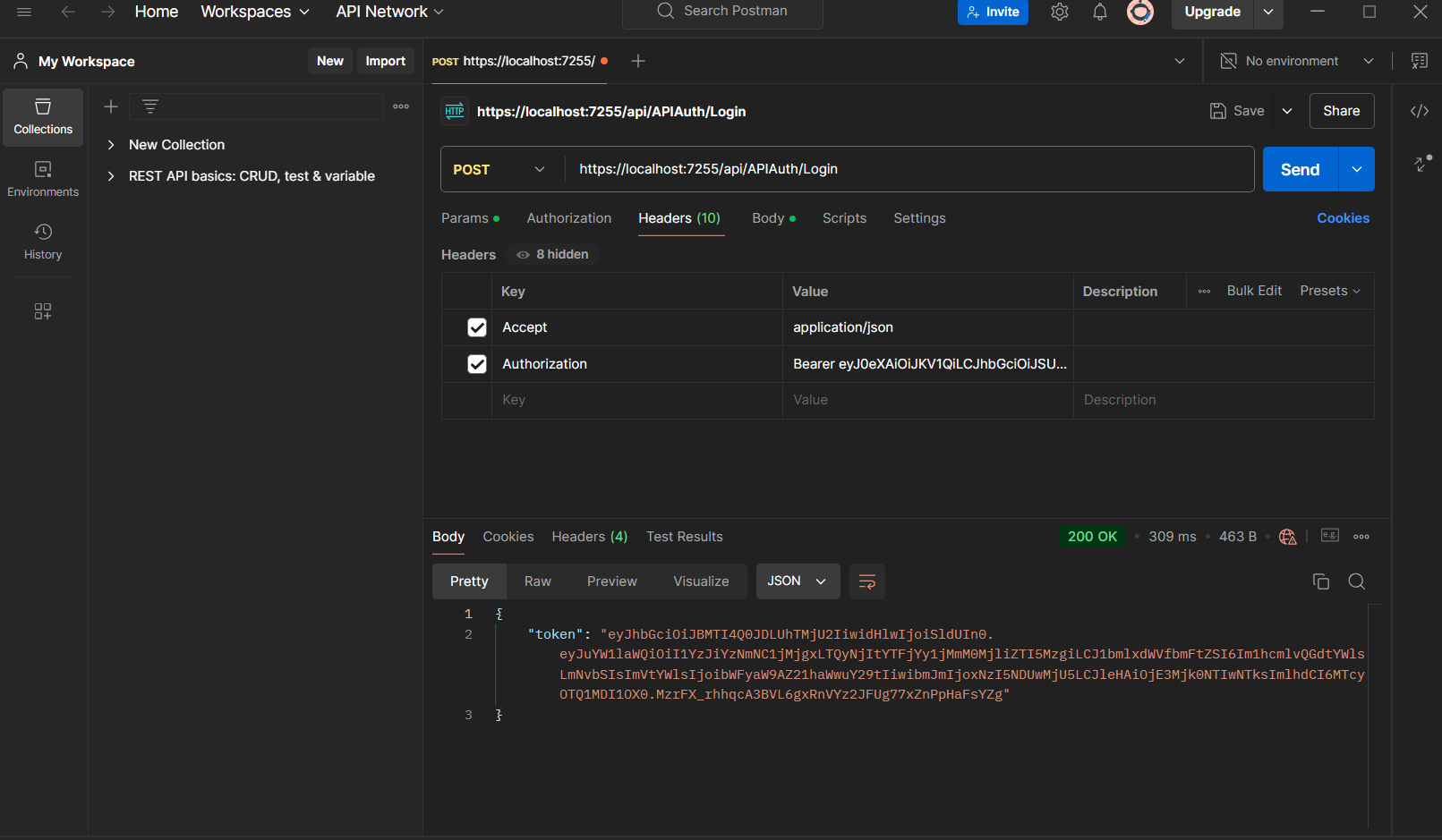
        // Run the application (Запуск приложения)

        app.Run(); // Запуск приложения

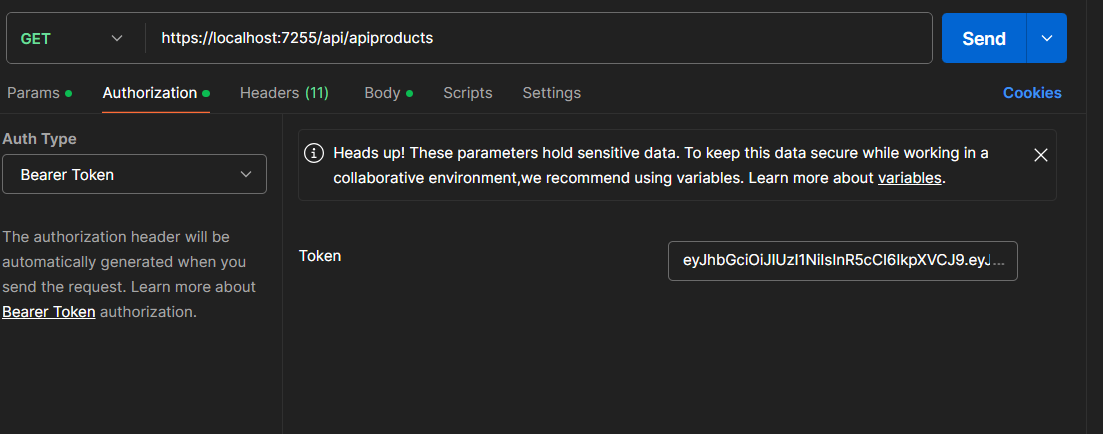
    }

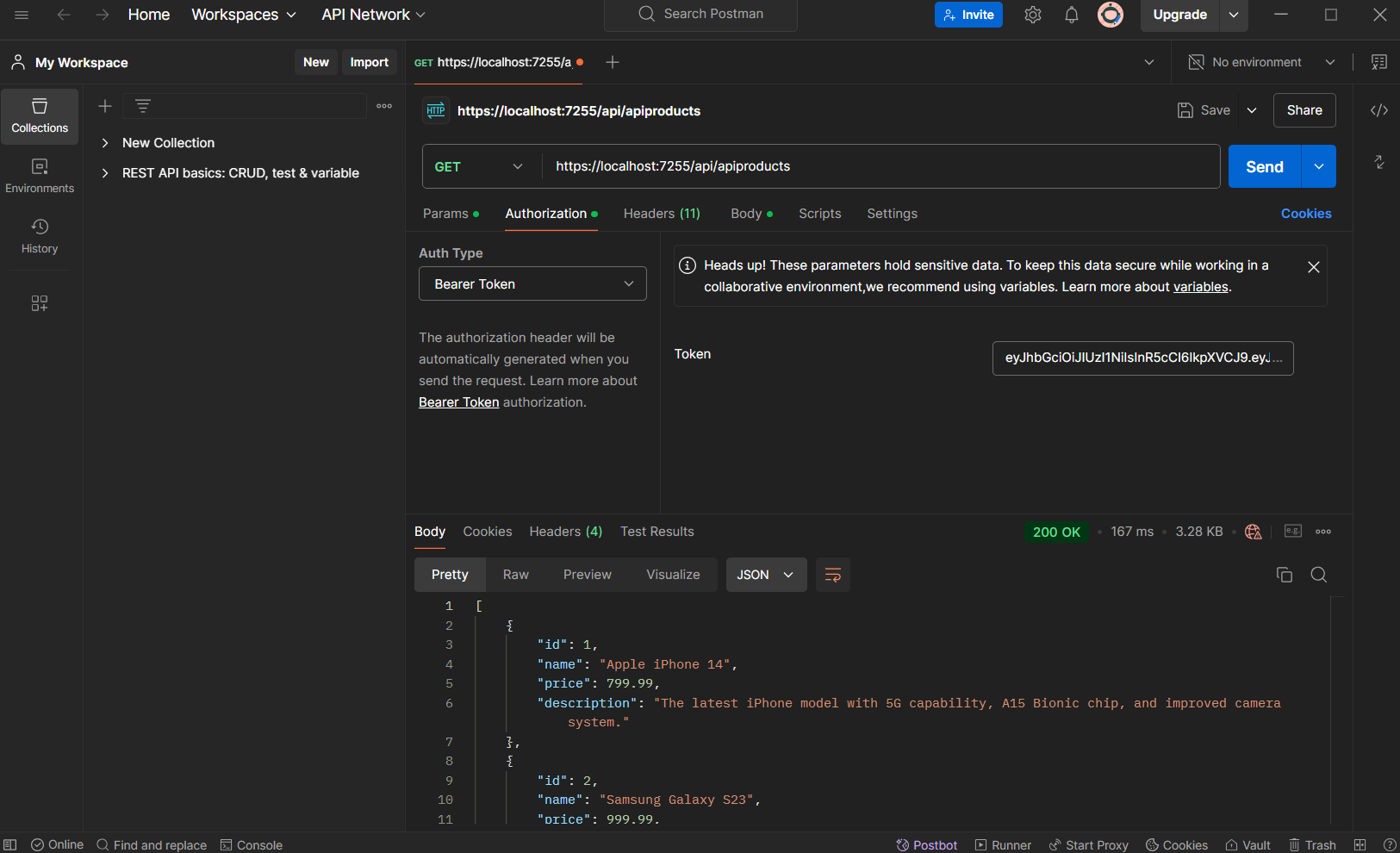
}





Получение товаров:





Фронтенд ajax js:

Index.html

<!DOCTYPE html>

<html lang="ru">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>CRUD Products</title>

    <!-- Подключение Bootstrap CSS -->

    <link href="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css" rel="stylesheet">

</head>

<body>

    <div class="container mt-5">

        <h1 class="text-center">Управление Продуктами</h1>

        <!-- Форма для создания нового продукта -->

        <form id="createProductForm">

            <div class="form-group">

                <label for="productName">Название продукта</label>

                <input type="text" class="form-control" id="productName" required>

            </div>

            <div class="form-group">

                <label for="productPrice">Цена продукта</label>

                <input type="number" class="form-control" id="productPrice" required>

            </div>

            <button type="submit" class="btn btn-primary">Создать продукт</button>

        </form>

        <hr>

        <!-- Список продуктов -->

        <h2>Список продуктов</h2>

        <ul id="productList" class="list-group"></ul>

    </div>

    <!-- Подключение jQuery и Bootstrap JS -->

    <script src="https://code.jquery.com/jquery-3.5.1.min.js"></script>

    <script src="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/js/bootstrap.bundle.min.js"></script>

    <script>

        // Получение токена JWT из localStorage

        const token = localStorage.getItem('jwtToken');

        // Функция для создания нового продукта

        function createProduct(product) {

            fetch('https://localhost:7255/api/products', {

                method: 'POST',

                headers: {

                    'Content-Type': 'application/json',

                    'Authorization': `Bearer ${token}`

                },

                body: JSON.stringify(product)

            })

            .then(response => response.json())

            .then(data => {

                console.log('Product created:', data);

                loadProducts(); // Обновляем список продуктов

            })

            .catch(error => console.error('Error creating product:', error));

        }

        // Функция для получения списка продуктов

        function loadProducts() {

            fetch('https://localhost:7255/api/products', {

                method: 'GET',

                headers: {

                    'Authorization': `Bearer ${token}`

                }

            })

            .then(response => response.json())

            .then(products => {

                const productList = document.getElementById('productList');

                productList.innerHTML = ''; // Очищаем текущий список

                // Добавляем каждый продукт в список

                products.forEach(product => {

                    const li = document.createElement('li');

                    li.className = 'list-group-item d-flex justify-content-between align-items-center';

                    li.innerHTML = `${product.name} - $${product.price}

                        <button class="btn btn-danger btn-sm" onclick="deleteProduct(${product.id})">Удалить</button>`;

                    productList.appendChild(li);

                });

            })

            .catch(error => console.error('Error loading products:', error));

        }

        // Функция для удаления продукта

        function deleteProduct(productId) {

            fetch(`https://localhost:7255/api/products/${productId}`, {

                method: 'DELETE',

                headers: {

                    'Authorization': `Bearer ${token}`

                }

            })

            .then(response => {

                if (response.ok) {

                    console.log('Product deleted');

                    loadProducts(); // Обновляем список продуктов

                } else {

                    console.error('Error deleting product');

                }

            })

            .catch(error => console.error('Error deleting product:', error));

        }

        // Обработчик отправки формы для создания продукта

        document.getElementById('createProductForm').addEventListener('submit', function(event) {

            event.preventDefault(); // Отменяем стандартное поведение формы

            const productName = document.getElementById('productName').value;

            const productPrice = document.getElementById('productPrice').value;

            const newProduct = { name: productName, price: productPrice };

            createProduct(newProduct); // Создаем новый продукт

            this.reset(); // Очищаем форму

        });

        // Загружаем продукты при загрузке страницы

        window.onload = loadProducts;

    </script>

</body>

</html>

Валидация форм, настрока прав доступа

Валидация модели:

using Microsoft.AspNetCore.Mvc;

using Microsoft.EntityFrameworkCore;

using System.ComponentModel.DataAnnotations;

using System.ComponentModel.DataAnnotations.Schema;

namespace shop\_app.Models

{

    public class Product

    {

        [Key]

        [DatabaseGenerated(DatabaseGeneratedOption.Identity)]

        [Required(ErrorMessage = "Id is required ...")]

        public int Id { get; set; } = 0;

        [Required(ErrorMessage = "Name is required ...")]

        [StringLength(20, MinimumLength = 2, ErrorMessage = "min: 2, max: 20")]

        public string? Name { get; set; } = string.Empty;

        [Required(ErrorMessage = "Price is required ...")]

        [Range(0.01, 100000.00, ErrorMessage = "min: 0.01, max: 100000.00")]

        public decimal Price { get; set; } = decimal.Zero;

        [Required(ErrorMessage = "Description is required ...")]

        [StringLength(1024, MinimumLength = 2, ErrorMessage = "min: 2, max: 1024")]

        public string? Description { get; set; } = string.Empty;

    }

}

Views/\_Layout: (Обязательно подключить Jquery механизм валидации работает через него в asp.net core)

@\* Validation important JQuery \*@

    <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>

    <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery-validate/1.19.1/jquery.validate.min.js"></script>

    <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery-validation-unobtrusive/3.2.11/jquery.validate.unobtrusive.min.js"></script>

Views/Create:

@addTagHelper \*, Microsoft.AspNetCore.Mvc.TagHelpers

@model shop\_app.Models.Product

@{

    ViewData["Title"] = "Create";

    Layout = "\_Layout";

}

<h2>Create Page</h2>

<form asp-controller="Product" asp-action="Create" method="post">

    <div class="form-group">

        <label asp-for="Name">Name:</label>

        <input type="text" asp-for="Name" class="form-control"/>

        <span asp-validation-for="Name" class="text-danger"></span>

    </div>

    <div class="form-group">

        <label asp-for="Price">Price:</label>

        <input type="text" asp-for="Price" class="form-control"/>

        <span asp-validation-for="Price" class="text-danger"></span>

    </div>

    <div class="form-group">

        <label asp-for="Description">Description:</label>

        <textarea type="text" asp-for="Description" class="form-control"></textarea>

        <span asp-validation-for="Description" class="text-danger"></span>

    </div>

    <button type="submit" class="btn btn-success">Create</button>

</form>

Views/Update:

@addTagHelper \*, Microsoft.AspNetCore.Mvc.TagHelpers

@model shop\_app.Models.Product

@{

    ViewData["Title"] = "Update";

    Layout = "\_Layout";

}

<h2>Update Page</h2>

<form asp-controller="Product" asp-action="Update" method="post">

    <div class="form-group">

        <label asp-for="Id">Id:</label>

        <input type="text" asp-for="Id" class="form-control" />

        <span asp-validation-for="Id" class="text-danger"></span>

    </div>

    <div class="form-group">

        <label asp-for="Name">Name:</label>

        <input type="text" asp-for="Name" class="form-control" />

        <span asp-validation-for="Name" class="text-danger"></span>

    </div>

    <div class="form-group">

        <label asp-for="Price">Price:</label>

        <input type="text" asp-for="Price" class="form-control" />

        <span asp-validation-for="Price" class="text-danger"></span>

    </div>

    <div class="form-group">

        <label asp-for="Description">Description:</label>

        <textarea type="text" asp-for="Description" class="form-control"></textarea>

        <span asp-validation-for="Description" class="text-danger"></span>

    </div>

    <button type="submit" class="btn btn-danger">Update</button>

</form>

Controllers/Product:

using Microsoft.AspNetCore.Authorization;

using Microsoft.AspNetCore.Mvc;

using shop\_app.Models;

using shop\_app.Services;

namespace shop\_app.Controllers

{

    public class ProductController : Controller

    {

        private readonly IServiceProduct \_serviceProduct;

        public ProductController(IServiceProduct serviceProduct)

        {

            \_serviceProduct = serviceProduct;

        }

        [HttpGet]

        public async Task<IActionResult> Read()

        {

            var products = await \_serviceProduct.ReadAsync();

            return View(products);

        }

        [HttpGet]

        public async Task<IActionResult> Details(int id)

        {

            var product = await \_serviceProduct.GetByIdAsync(id);

            return View(product);

        }

        [HttpGet]

        public IActionResult Page404() => View();

        [HttpGet]

        public IActionResult Create()

            => (User.IsInRole("admin")) ? View() : RedirectToAction("Page404");

        [Authorize(Roles = "admin")]

        [HttpPost]

        [ValidateAntiForgeryToken]

        public async Task<IActionResult> Create([Bind("Id,Name,Price,Description")] Product product)

        {

            Console.WriteLine($"Id: {product.Id}, Name: {product.Name}, Price: {product.Price}, Description: {product.Description}");

            Console.WriteLine(product.Id.GetType());

            if(ModelState.IsValid)

            {

                \_ = await \_serviceProduct.CreateAsync(product);

                return RedirectToAction(nameof(Read));

            }

            return View(product);

        }

        [HttpGet]

        public IActionResult Update()

            => (User.IsInRole("admin")) ? View() : RedirectToAction("Page404");

        [Authorize(Roles = "admin")] //if admin or moderator [Authorize(Roles = "admin,moderator")]

        [HttpPost]

        [ValidateAntiForgeryToken]

        public async Task<IActionResult> Update(int id, [Bind("Id,Name,Price,Description")] Product product)

        {

            if (ModelState.IsValid)

            {

                \_ = await \_serviceProduct.UpdateAsync(id, product);

                return RedirectToAction(nameof(Read));

            }

            return View(product);

        }

        [HttpGet]

        public IActionResult Delete()

            => (User.IsInRole("admin")) ? View() : RedirectToAction("Page404");

        [Authorize(Roles = "admin")]

        [HttpPost]

        public async Task<IActionResult> Delete(int id)

        {

            \_ = await \_serviceProduct.DeleteAsync(id);

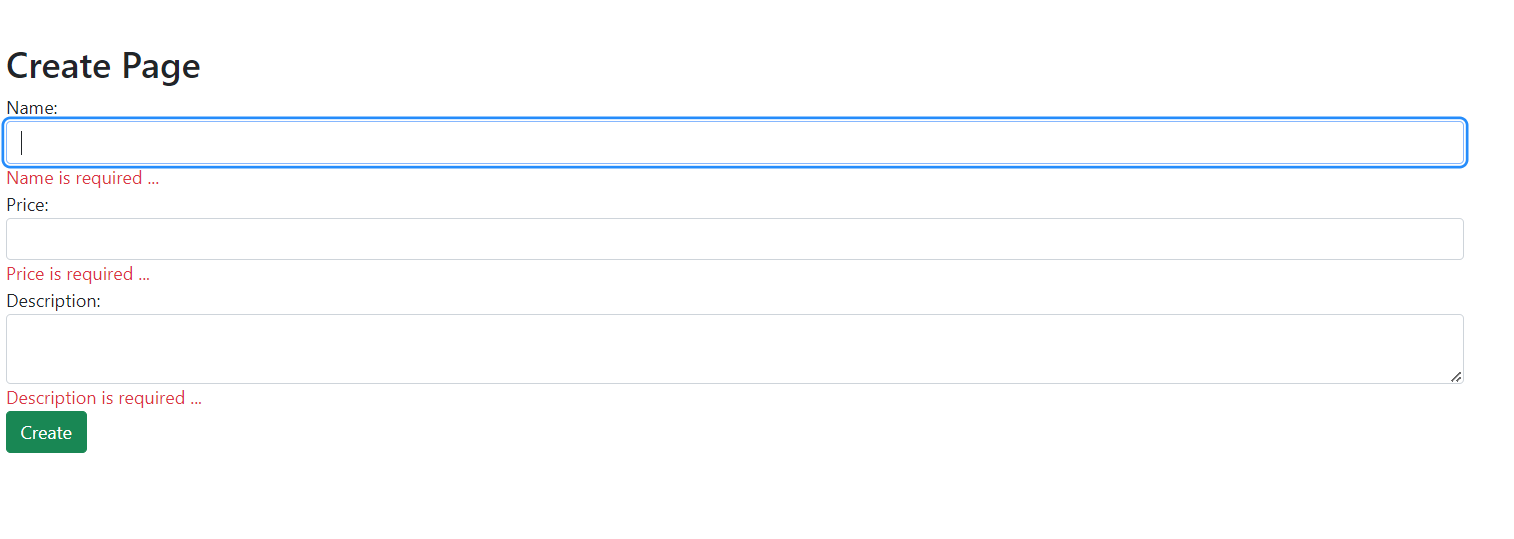
            return RedirectToAction(nameof(Read));

        }

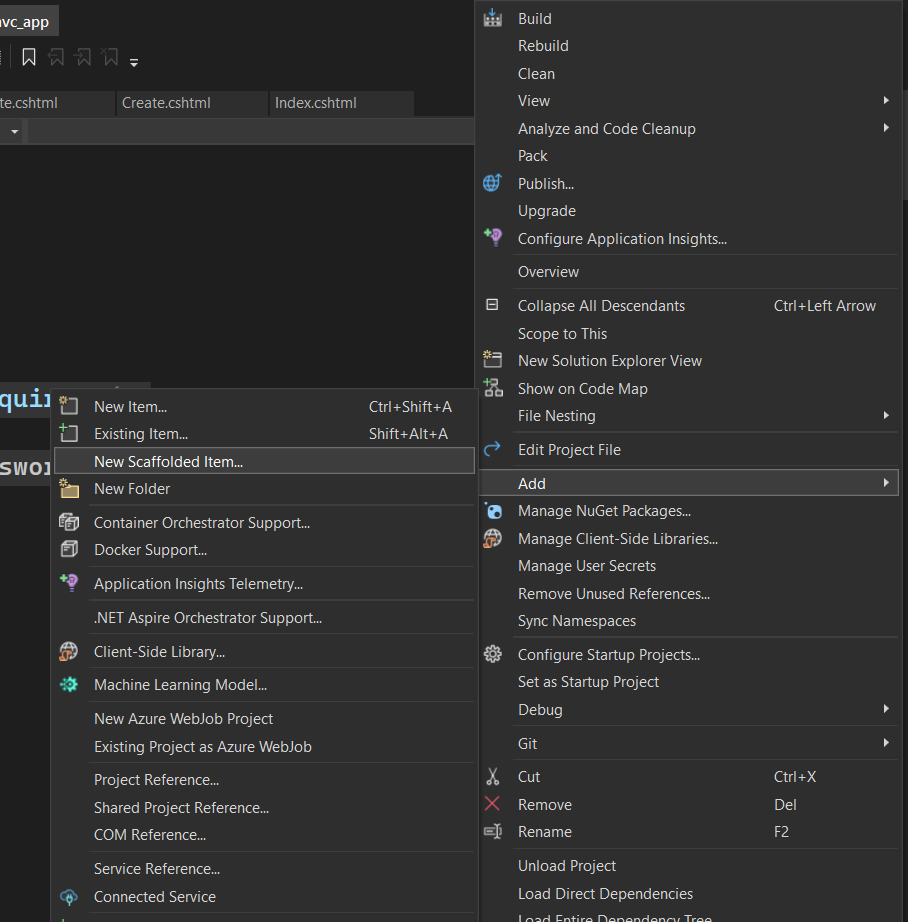
    }

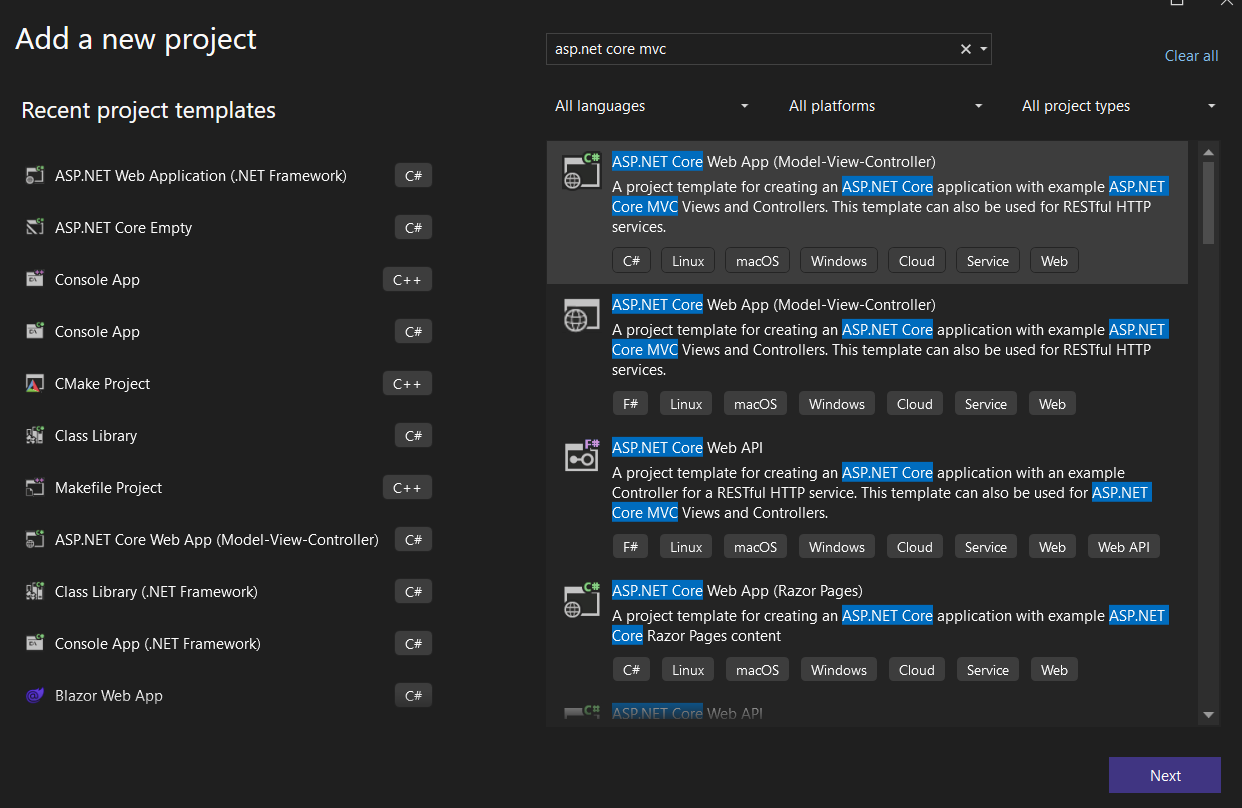
}

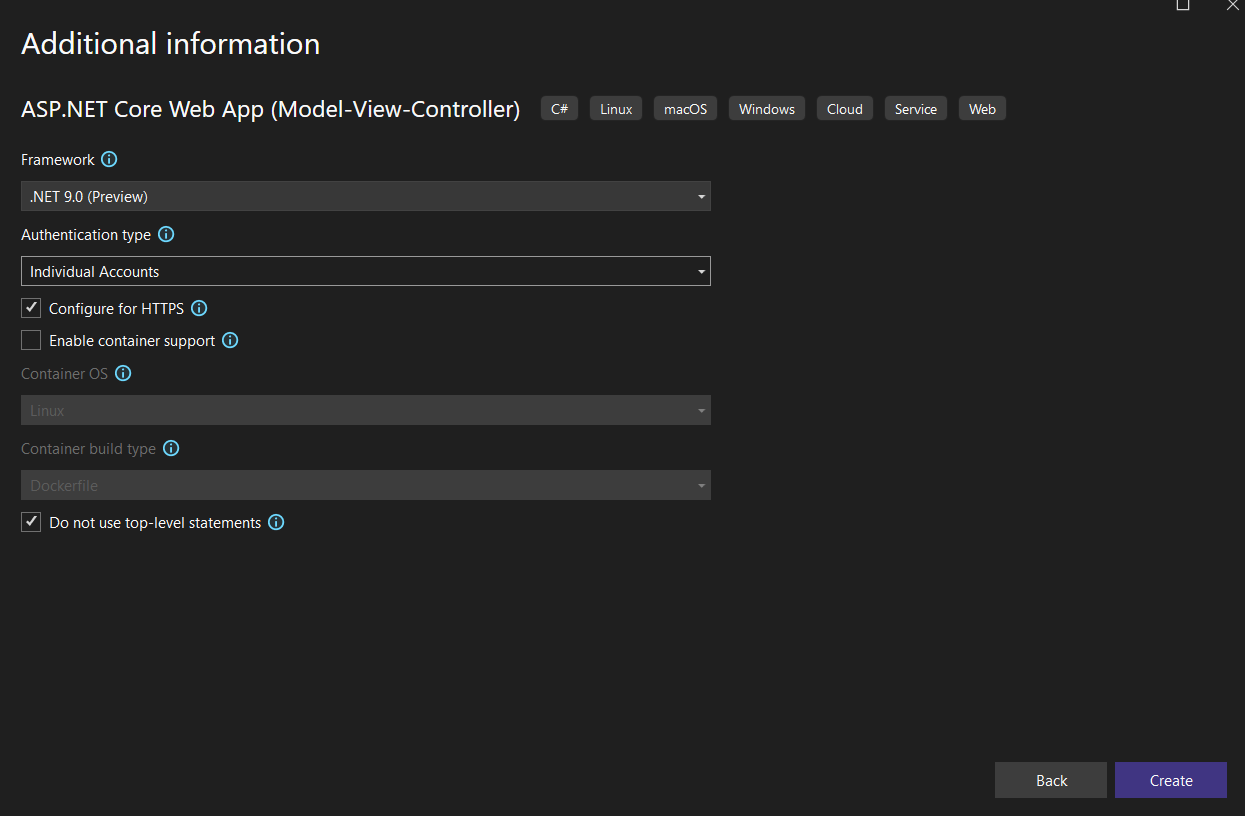
Result:



Практика (Scaffolded)







В файле appsettings.json добавь строку подключения для базы данных.

"ConnectionStrings": {

  "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=UserManagerDemoDb;Trusted\_Connection=True;MultipleActiveResultSets=true"

}

Настройте Entity Framework Core:

Microsoft.AspNetCore.Identity.EntityFrameworkCore

Microsoft.AspNetCore.Identity.UI

Microsoft.EntityFrameworkCore.SqlServer

Microsoft.EntityFrameworkCore.Tools

Применение миграций

Add-Migration -Context UserContext UserIdentityMigrationName

Update-Database –Context UserContext

Создание контроллера для работы с UserManager

using Microsoft.AspNetCore.Identity;   // Подключаем пространство имен для работы с UserManager и IdentityUser

using Microsoft.AspNetCore.Mvc;       // Подключаем пространство имен для использования контроллеров и действий

using System.Threading.Tasks;         // Подключаем пространство имен для работы с асинхронными методами

public class UserController : Controller

{

    // Поле для хранения экземпляра UserManager, который отвечает за управление пользователями

    private readonly UserManager<IdentityUser> \_userManager;

    private readonly SignInManager<IdentityUser> \_signInManager;

    // Конструктор контроллера. Через Dependency Injection (DI) внедряем UserManager для управления пользователями

    public UserController(UserManager<IdentityUser> userManager, SignInManager<IdentityUser> signInManager)

    {

        \_userManager = userManager;  // Присваиваем экземпляр UserManager локальной переменной

        \_signInManager = signInManager; // Присваиваем экземпляр SignInManager локальной переменной

    }

    // GET-метод для отображения страницы с формами создания, удаления и обновления пользователей

    [HttpGet]

    public IActionResult Index()

    {

        // Возвращаем представление, которое отображает страницу с пользовательскими операциями

        return View();

    }

    // GET-метод для отображения формы регистрации

    public IActionResult Register()

    {

        return View();

    }

    // POST-метод для создания нового пользователя

    [HttpPost]

    public async Task<IActionResult> Create(string email, string password)

    {

        // Проверяем, что email и пароль были переданы

        if (string.IsNullOrEmpty(email) || string.IsNullOrEmpty(password))

        {

            // Если email или пароль отсутствуют, возвращаем ошибку

            return BadRequest("Email и пароль обязательны.");

        }

        // Создаем нового пользователя IdentityUser с указанным email

        var user = new IdentityUser {

UserName = email,

Email = email,

EmailConfirmed = true //Imitation check email

};

        // Используем UserManager для создания пользователя с переданным паролем

        var result = await \_userManager.CreateAsync(user, password);

        // Если создание прошло успешно

        if (result.Succeeded)

        {

            // Возвращаем успешный ответ с сообщением

            return Ok("Пользователь создан.");

        }

        // Если возникли ошибки, добавляем их в ModelState для отображения пользователю

        foreach (var error in result.Errors)

        {

            ModelState.AddModelError(string.Empty, error.Description);

        }

        // Возвращаем ошибки, если не удалось создать пользователя

        return BadRequest(ModelState);

    }

    // GET-метод для отображения формы авторизации

    public IActionResult Login()

    {

        return View();

    }

    // POST-метод для авторизации пользователя

    [HttpPost]

    public async Task<IActionResult> Login(string email, string password)

    {

        // Проверяем, что email и пароль были переданы

        if (string.IsNullOrEmpty(email) || string.IsNullOrEmpty(password))

        {

            // Если email или пароль отсутствуют, возвращаем ошибку

            return BadRequest("Email и пароль обязательны.");

        }

        // Проверяем учетные данные пользователя

        var result = await \_signInManager.PasswordSignInAsync(email, password, isPersistent: false, lockoutOnFailure: false);

        // Если вход успешен

        if (result.Succeeded)

        {

            return Ok("Успешный вход.");

        }

        // Если вход не удался

        return BadRequest("Неверный email или пароль.");

    }

    // POST-метод для удаления пользователя по его идентификатору (userId)

    [HttpPost]

    public async Task<IActionResult> Delete(string userId)

    {

        // Находим пользователя по его идентификатору (userId)

        var user = await \_userManager.FindByIdAsync(userId);

        // Если пользователь не найден, возвращаем ошибку "Не найден"

        if (user == null)

        {

            return NotFound("Пользователь не найден.");

        }

        // Удаляем пользователя с помощью UserManager

        var result = await \_userManager.DeleteAsync(user);

        // Если удаление прошло успешно, возвращаем сообщение об успешном удалении

        if (result.Succeeded)

        {

            return Ok("Пользователь удален.");

        }

        // Если произошла ошибка, возвращаем сообщение об ошибке

        return BadRequest("Не удалось удалить пользователя.");

    }

    // POST-метод для обновления данных пользователя (email) по его идентификатору (userId)

    [HttpPost]

    public async Task<IActionResult> Update(string userId, string newEmail)

    {

        // Находим пользователя по его идентификатору (userId)

        var user = await \_userManager.FindByIdAsync(userId);

        // Если пользователь не найден, возвращаем ошибку "Не найден"

        if (user == null)

        {

            return NotFound("Пользователь не найден.");

        }

        // Обновляем email и имя пользователя

        user.Email = newEmail;

        user.UserName = newEmail;

        // Используем UserManager для обновления данных пользователя

        var result = await \_userManager.UpdateAsync(user);

        // Если обновление прошло успешно, возвращаем сообщение об успешном обновлении

        if (result.Succeeded)

        {

            return Ok("Данные пользователя обновлены.");

        }

        // Если произошла ошибка, возвращаем сообщение об ошибке

        return BadRequest("Не удалось обновить данные пользователя.");

    }

}

Представление Index.cshtml

@addTagHelper \*, Microsoft.AspNetCore.Mvc.TagHelpers

@model List<Microsoft.AspNetCore.Identity.IdentityUser> <!-- Модель - список пользователей -->

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="utf-8" />

    <meta name="viewport" content="width=device-width, initial-scale=1.0" />

    <title>Управление пользователями</title>

</head>

<body>

    <h1>Управление пользователями</h1>

<!-- Форма для создания нового пользователя -->

    <h2>Создать нового пользователя</h2>

    <form asp-action="Create" method="post">

        <label for="email">Email:</label>

        <input type="email" id="email" name="email" required />

        <br />

        <label for="password">Пароль:</label>

        <input type="password" id="password" name="password" required />

        <br />

        <button type="submit">Создать</button>

    </form>

    <h2>Список пользователей</h2>

    <table>

        <thead>

            <tr>

                <th>Email</th>

                <th>Действия</th>

            </tr>

        </thead>

        <tbody>

            @foreach (var user in Model)

            {

                <tr>

                    <td>@user.Email</td>

                    <td>

                        <!-- Кнопка для обновления данных пользователя -->

                        <form asp-action="Update" method="post" style="display:inline;">

                            <input type="hidden" name="userId" value="@user.Id" />

                            <input type="email" name="newEmail" placeholder="Новый Email" required />

                            <button type="submit">Обновить</button>

                        </form>

                        <!-- Кнопка для удаления пользователя -->

                        <form asp-action="Delete" method="post" style="display:inline;">

                            <input type="hidden" name="userId" value="@user.Id" />

                            <button type="submit">Удалить</button>

                        </form>

                    </td>

                </tr>

            }

        </tbody>

    </table>

</body>

</html>

Добавление аутентификации по внешним провайдерам: Если вы планируете добавить вход через внешние провайдеры (такие как Google или Facebook), вам нужно будет настроить их здесь, используя AddAuthentication.

Конфигурация cookie аутентификации: Если вам нужно изменить настройки cookie для аутентификации, вы можете это сделать так:

builder.Services.ConfigureApplicationCookie(options =>

{

    options.LoginPath = "/User/Login"; // Путь для страницы входа

    options.LogoutPath = "/User/Logout"; // Путь для выхода

});

Для внедрения RoleManager в ваш проект ASP.NET Core с использованием Identity, вам потребуется выполнить несколько шагов, чтобы добавить функциональность управления ролями.

1. Обновление Program.cs

В Program.cs нужно зарегистрировать RoleManager в контейнере зависимостей. Обычно это делается автоматически при использовании AddDefaultIdentity, но можно сделать это явно:

builder.Services.AddIdentity<IdentityUser, IdentityRole>(options =>

{

    options.SignIn.RequireConfirmedAccount = true; // Настройки для входа

})

.AddEntityFrameworkStores<ApplicationDbContext>(); // Подключение к контексту базы данных

2. Обновление UserController

Теперь обновим ваш UserController, чтобы использовать RoleManager. Вам нужно будет добавить поле для RoleManager и методы для создания и удаления ролей, а также для назначения ролей пользователям.

Вот как может выглядеть ваш обновлённый UserController:

using Microsoft.AspNetCore.Identity;  // Подключаем пространство имен для работы с UserManager и IdentityUser

using Microsoft.AspNetCore.Mvc;       // Подключаем пространство имен для использования контроллеров и действий

using System.Threading.Tasks;         // Подключаем пространство имен для работы с асинхронными методами

public class UserController : Controller

{

    private readonly UserManager<IdentityUser> \_userManager; // Поле для UserManager

    private readonly RoleManager<IdentityRole> \_roleManager; // Поле для RoleManager

    // Конструктор контроллера. Внедряем UserManager и RoleManager

    public UserController(UserManager<IdentityUser> userManager, RoleManager<IdentityRole> roleManager)

    {

        \_userManager = userManager;  // Присваиваем экземпляр UserManager

        \_roleManager = roleManager;  // Присваиваем экземпляр RoleManager

    }

    // POST-метод для создания роли

    [HttpPost]

    public async Task<IActionResult> CreateRole(string roleName)

    {

        if (string.IsNullOrEmpty(roleName)) // Проверка на пустое имя роли

        {

            return BadRequest("Имя роли обязательно.");

        }

        var roleExists = await \_roleManager.RoleExistsAsync(roleName); // Проверка, существует ли роль

        if (roleExists)

        {

            return BadRequest("Роль уже существует.");

        }

        var role = new IdentityRole(roleName); // Создание новой роли

        var result = await \_roleManager.CreateAsync(role); // Создание роли через RoleManager

        if (result.Succeeded)

        {

            return Ok("Роль создана."); // Успешное создание роли

        }

        // Обработка ошибок

        foreach (var error in result.Errors)

        {

            ModelState.AddModelError(string.Empty, error.Description);

        }

        return BadRequest(ModelState); // Возвращаем ошибки

    }

    // POST-метод для назначения роли пользователю

    [HttpPost]

    public async Task<IActionResult> AssignRole(string userId, string roleName)

    {

        var user = await \_userManager.FindByIdAsync(userId); // Поиск пользователя по ID

        if (user == null)

        {

            return NotFound("Пользователь не найден."); // Если пользователь не найден

        }

        var result = await \_userManager.AddToRoleAsync(user, roleName); // Назначение роли пользователю

        if (result.Succeeded)

        {

            return Ok("Роль назначена пользователю."); // Успешное назначение

        }

        // Обработка ошибок

        foreach (var error in result.Errors)

        {

            ModelState.AddModelError(string.Empty, error.Description);

        }

        return BadRequest(ModelState); // Возвращаем ошибки

    }

}

3. Создание представлений для управления ролями

Создайте соответствующие представления для создания и назначения ролей. Например, вы можете создать форму для ввода имени роли и формы для выбора пользователя и роли.

4. Пример представления для создания роли

@{

    ViewData["Title"] = "Создать роль";

}

<h2>Создать роль</h2>

<form asp-controller="User" asp-action="CreateRole" method="post">

    <div>

        <label for="roleName">Имя роли:</label>

        <input type="text" id="roleName" name="roleName" required />

    </div>

    <button type="submit">Создать роль</button>

</form>

5. Пример представления для назначения роли пользователю

@{

    ViewData["Title"] = "Назначить роль";

}

<h2>Назначить роль пользователю</h2>

<form asp-controller="User" asp-action="AssignRole" method="post">

    <div>

        <label for="userId">ID пользователя:</label>

        <input type="text" id="userId" name="userId" required />

    </div>

    <div>

        <label for="roleName">Имя роли:</label>

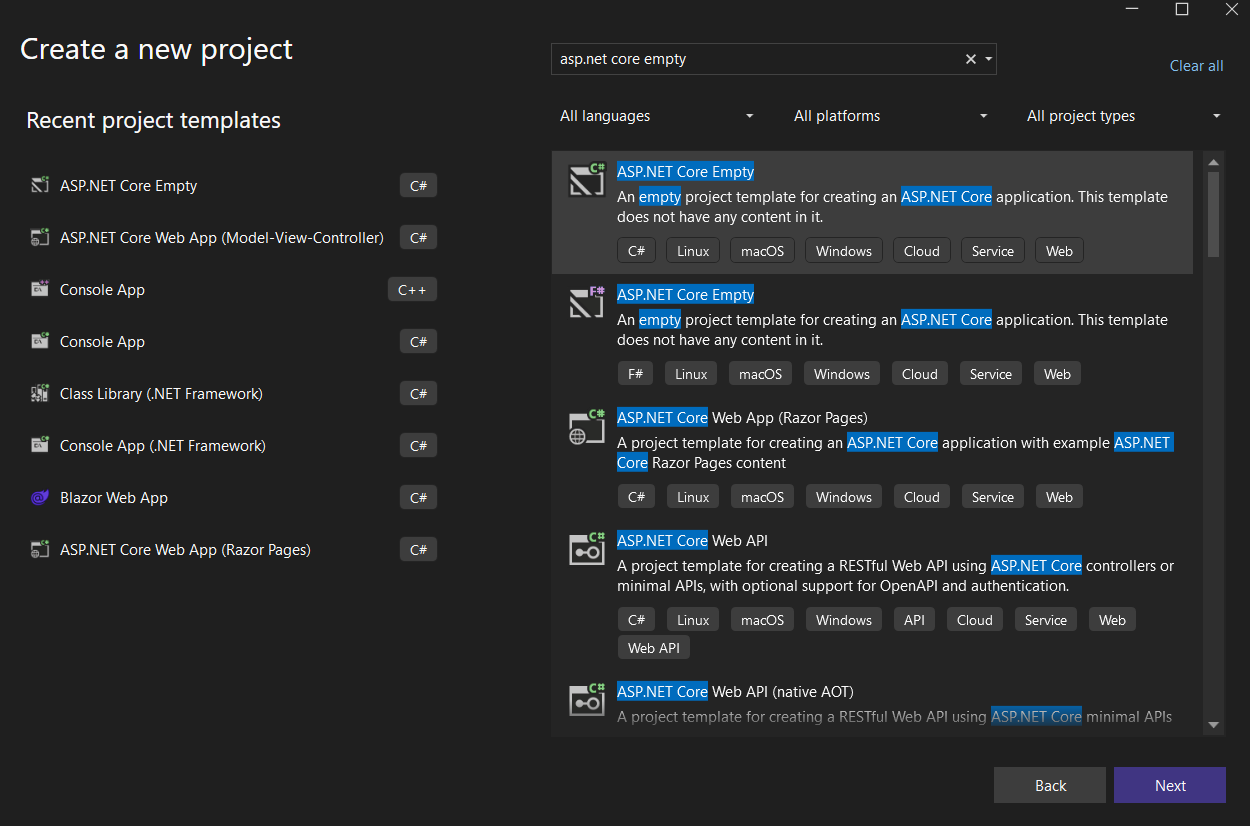
        <input type="text" id="roleName" name="roleName" required />

    </div>

    <button type="submit">Назначить роль</button>

</form>

Практика 2



namespace news\_practise\_app

{

    public class Program

    {

        public static void Main(string[] args)

        {

            var builder = WebApplication.CreateBuilder(args);

            // Добавляем сервисы MVC

            builder.Services.AddControllersWithViews();

            // Добавляем сессии

            builder.Services.AddSession();

            var app = builder.Build();

            // Включаем использование статических файлов и маршрутизацию

            app.UseStaticFiles();

            app.UseRouting();

            // Включаем сессии

            app.UseSession();

            // Включаем авторизацию и аутентификацию

            app.UseAuthentication();

            app.UseAuthorization();

            app.MapControllerRoute(

                name: "default",

                pattern: "{controller=Home}/{action=Index}/{id?}");

            app.Run();

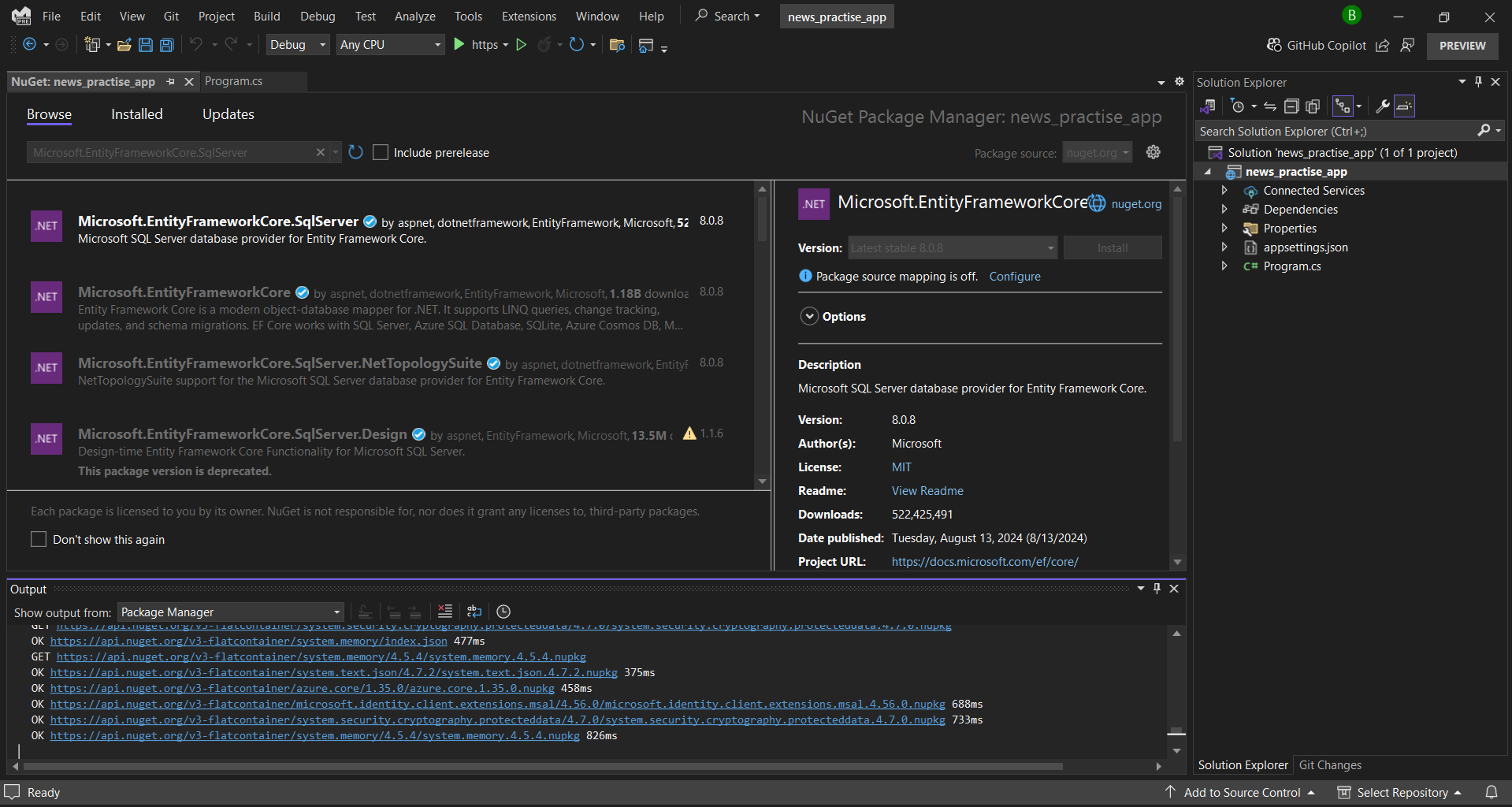
        }

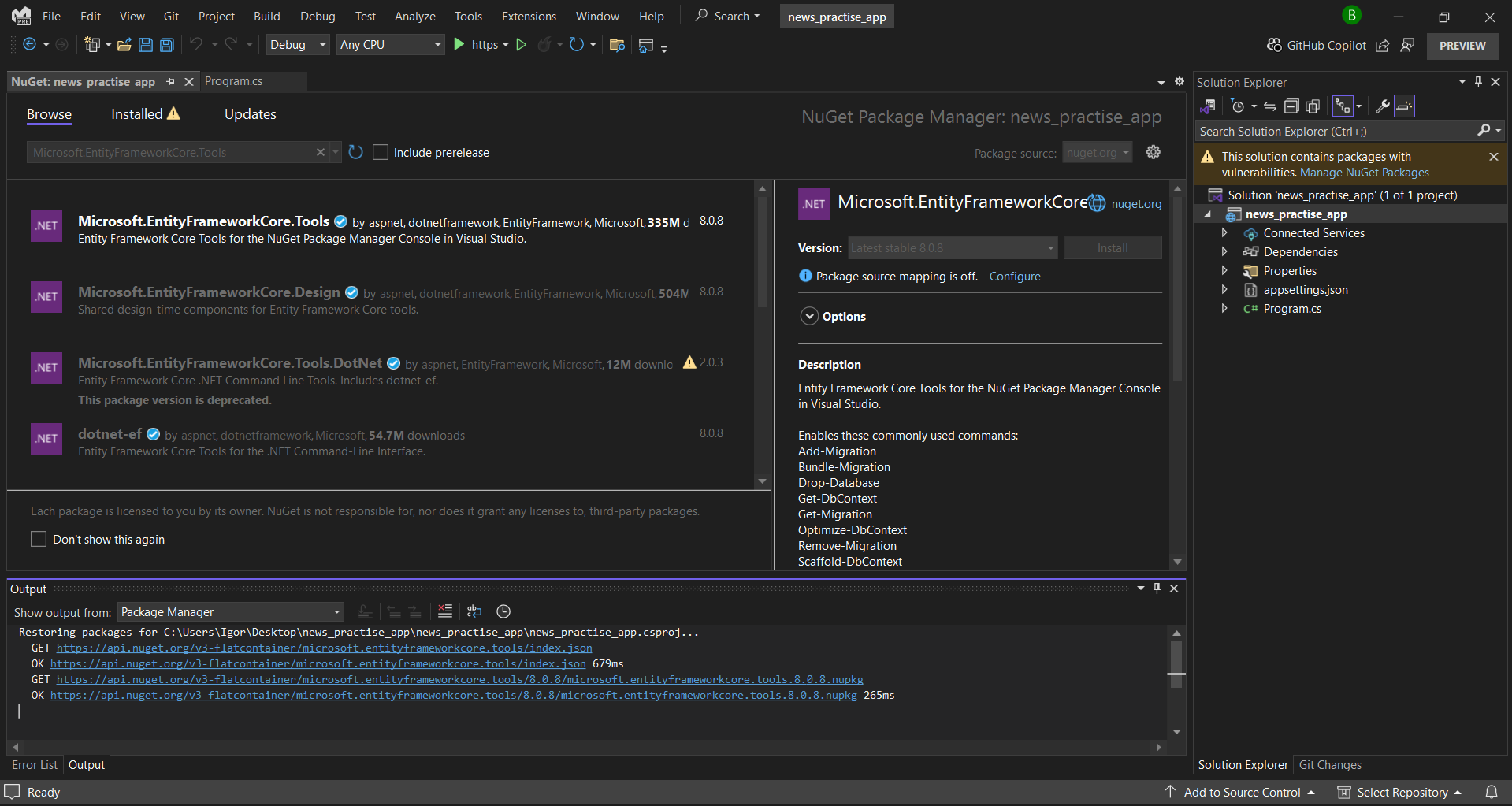
    }

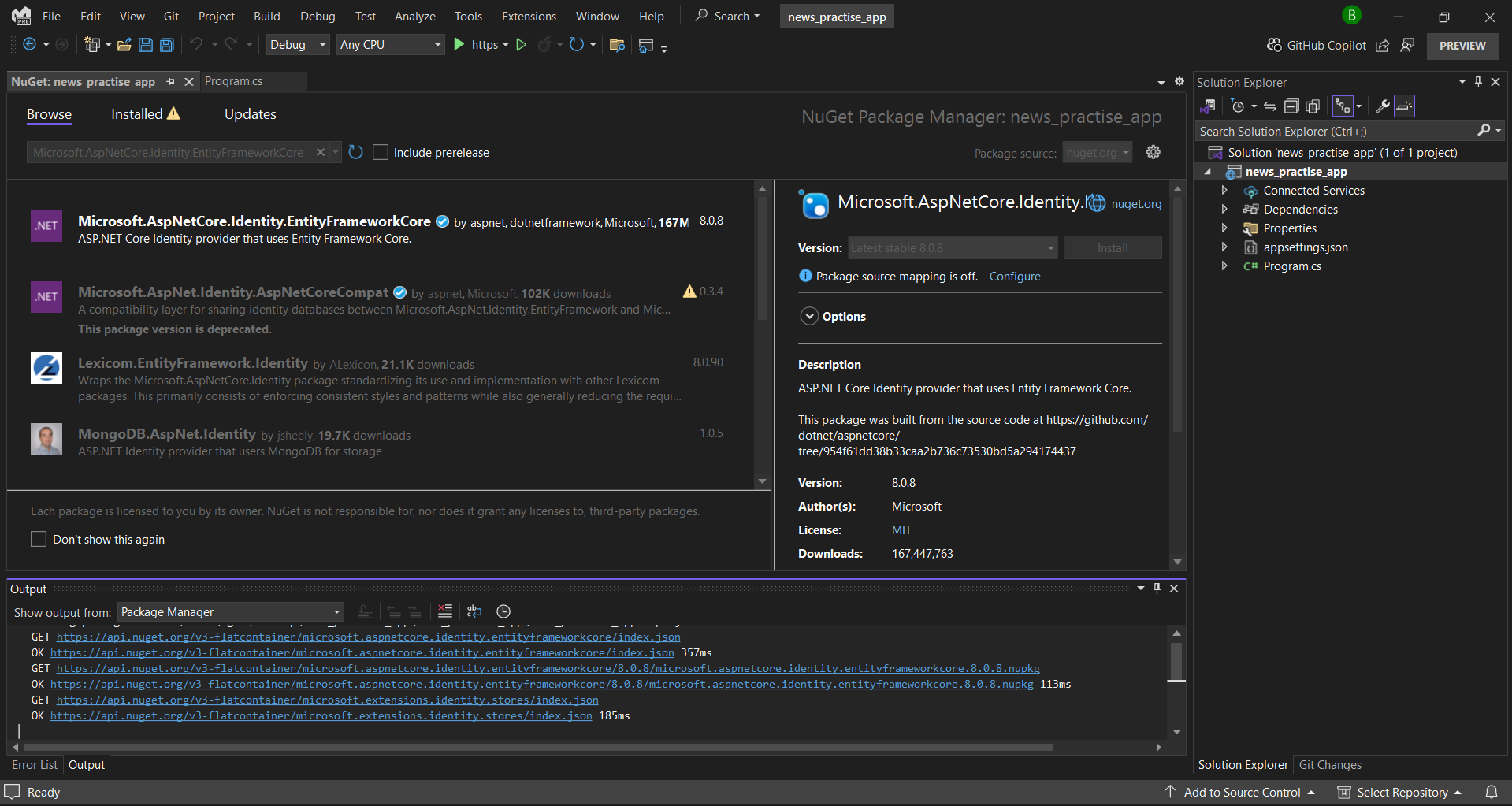
}

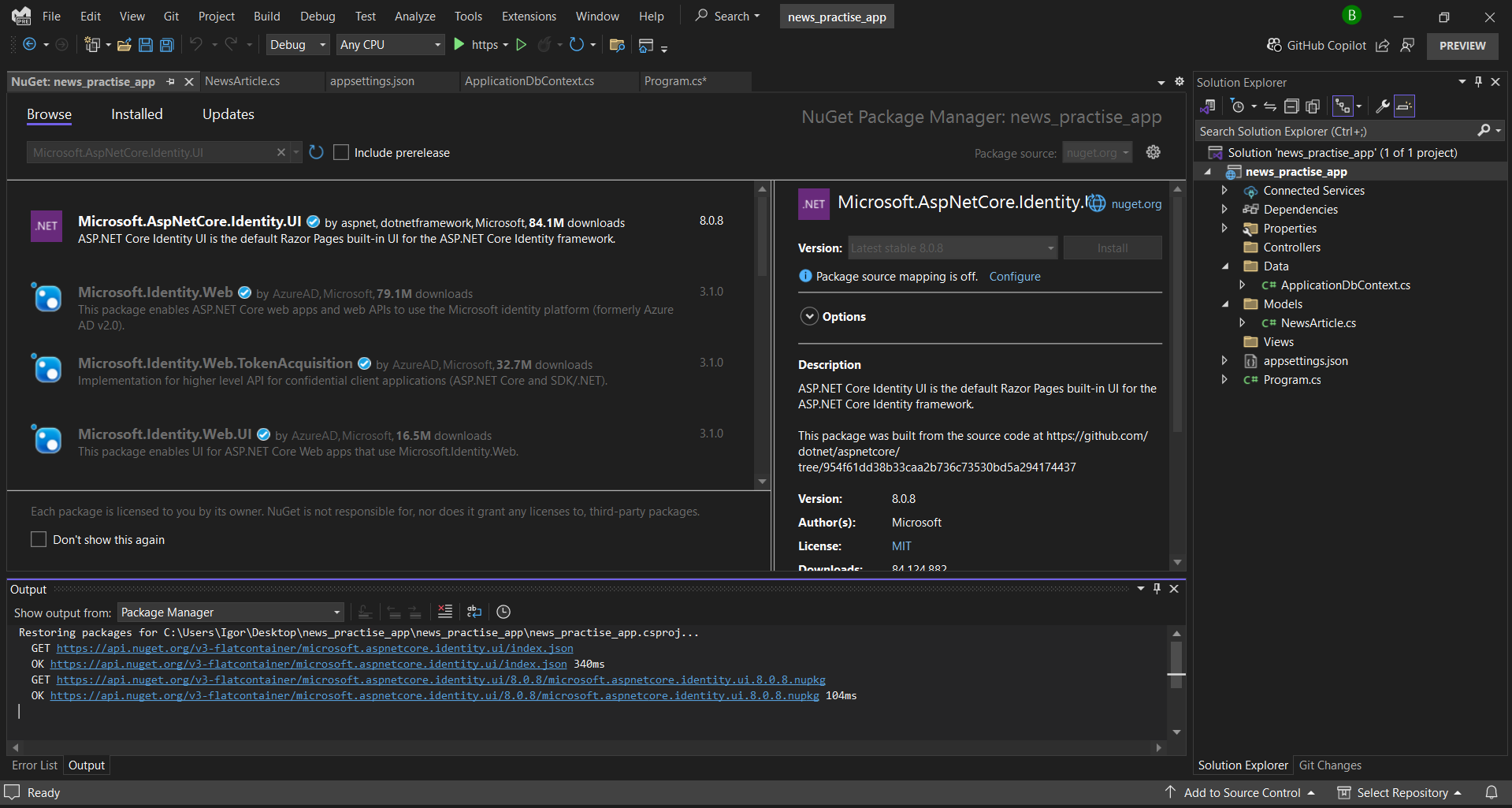
Добавление Entity Framework Core

Теперь установим Entity Framework Core и создадим контекст данных. Выполните команду в терминале для установки пакетов EF Core:









Создание модели NewsArticle

Добавьте папку Model/

Создайте новый класс модели NewsArticle, который будет представлять сущность новости. Этот класс должен быть добавлен в отдельный файл, например, в папке Models.

Пример модели:

using Microsoft.AspNetCore.Identity;

using System;

using System.ComponentModel.DataAnnotations;

namespace news\_practise\_app.Models

{

    public class NewsArticle

    {

        [Key]

        public int ArticleId { get; set; }

        [Required]

        [MaxLength(200)]

        public string Title { get; set; }

        [Required]

        public string Content { get; set; }

        public string AuthorId { get; set; } // Связь с автором (пользователем)

        [DataType(DataType.DateTime)]

        public DateTime CreatedAt { get; set; } = DateTime.Now;

        [DataType(DataType.DateTime)]

        public DateTime? UpdatedAt { get; set; }

        // Навигационное свойство для связи с пользователем

        public IdentityUser Author { get; set; }

    }

}

Настройка базы данных и контекста

Создайте новый файл Data/ApplicationDbContext.cs и добавьте следующий код для контекста базы данных:

using Microsoft.AspNetCore.Identity.EntityFrameworkCore;

using Microsoft.EntityFrameworkCore;

namespace news\_practise\_app.Data

{

    public class ApplicationDbContext : IdentityDbContext

    {

        public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)

            : base(options)

        {

        }

        // Добавьте DbSet для ваших сущностей

        public DbSet<NewsArticle> NewsArticles { get; set; }

    }

}

Настройка строки подключения и конфигурация EF Core

Откройте файл appsettings.json и добавьте строку подключения к базе данных:

{

    "Logging": {

      "LogLevel": {

        "Default": "Information",

        "Microsoft.AspNetCore": "Warning"

      }

    },

    "ConnectionStrings": {

      "DefaultConnection": "Server=DESKTOP-EK8GAKM;Database=NewsWebsiteDb;Trusted\_Connection=True;MultipleActiveResultSets=true"

    },

    "AllowedHosts": "\*"

  }

Теперь в Program.cs добавим регистрацию контекста:

using Microsoft.AspNetCore.Identity;

using Microsoft.EntityFrameworkCore;

using news\_practise\_app.Data;

namespace news\_practise\_app

{

    public class Program

    {

        public static void Main(string[] args)

        {

            var builder = WebApplication.CreateBuilder(args);

            // Добавляем подключение к базе данных

            builder.Services.AddDbContext<ApplicationDbContext>(options =>

                options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));

            // Добавляем Identity для регистрации и авторизации

builder.Services.AddDefaultIdentity<IdentityUser>(options => options.SignIn.RequireConfirmedAccount = true)

                .AddEntityFrameworkStores<ApplicationDbContext>();

            // Добавляем MVC

            builder.Services.AddControllersWithViews();

            // Добавляем поддержку сессий и куки

            builder.Services.AddSession();

            var app = builder.Build();

            // Настройка middleware

            if (!app.Environment.IsDevelopment())

            {

                app.UseExceptionHandler("/Home/Error");

                app.UseHsts();

            }

            app.UseHttpsRedirection();

            app.UseStaticFiles();

            app.UseRouting();

            // Включение сессий

            app.UseSession();

            // Включаем аутентификацию и авторизацию

            app.UseAuthentication();

            app.UseAuthorization();

            app.MapControllerRoute(

                name: "default",

                pattern: "{controller=Home}/{action=Index}/{id?}");

            app.MapRazorPages();

            app.Run();

        }

    }

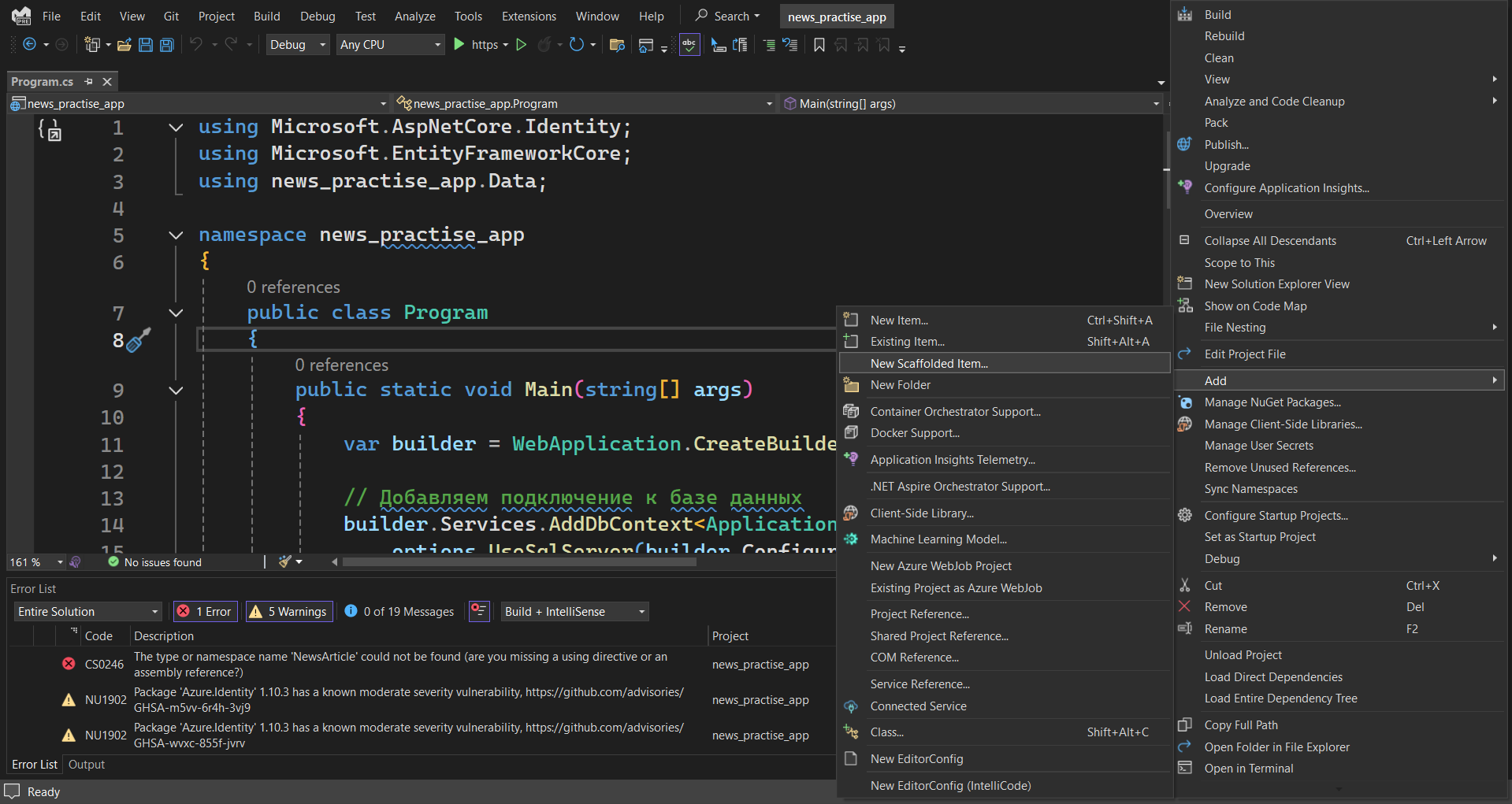
}

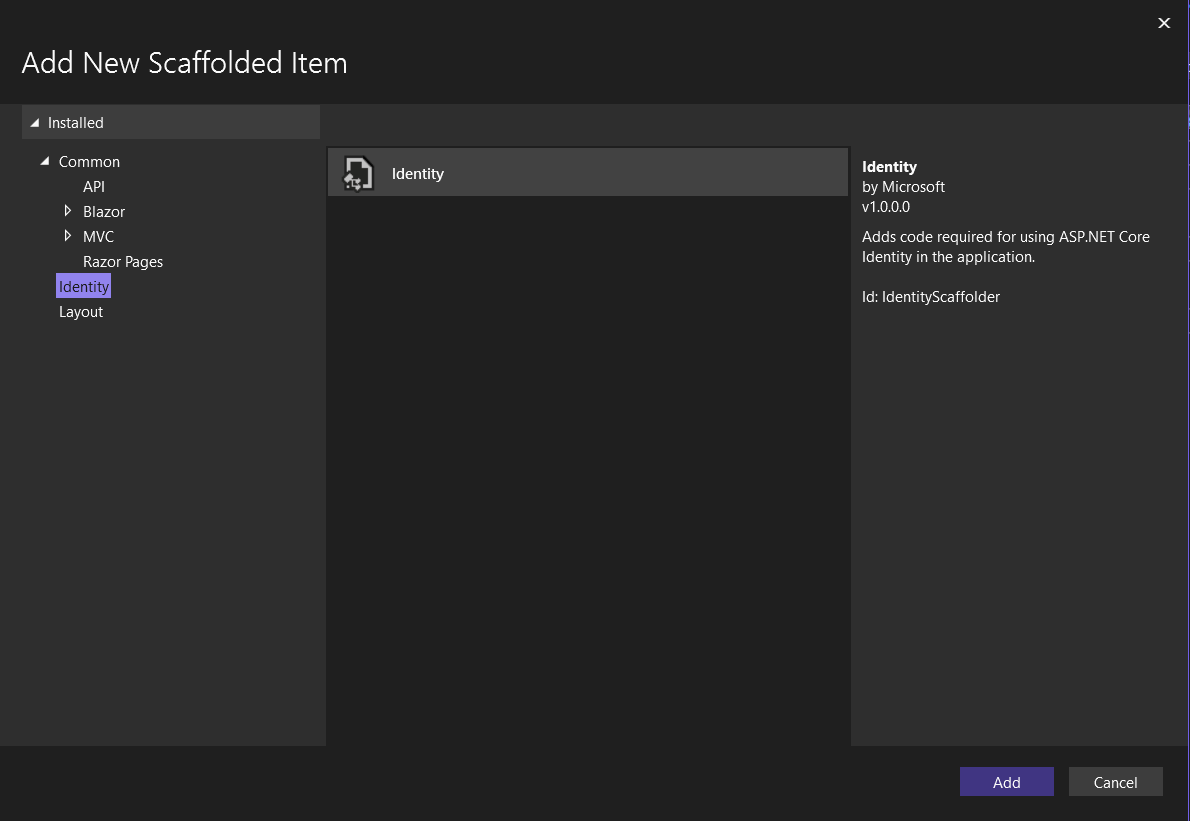
Контроллеры и представления для регистрации и авторизации

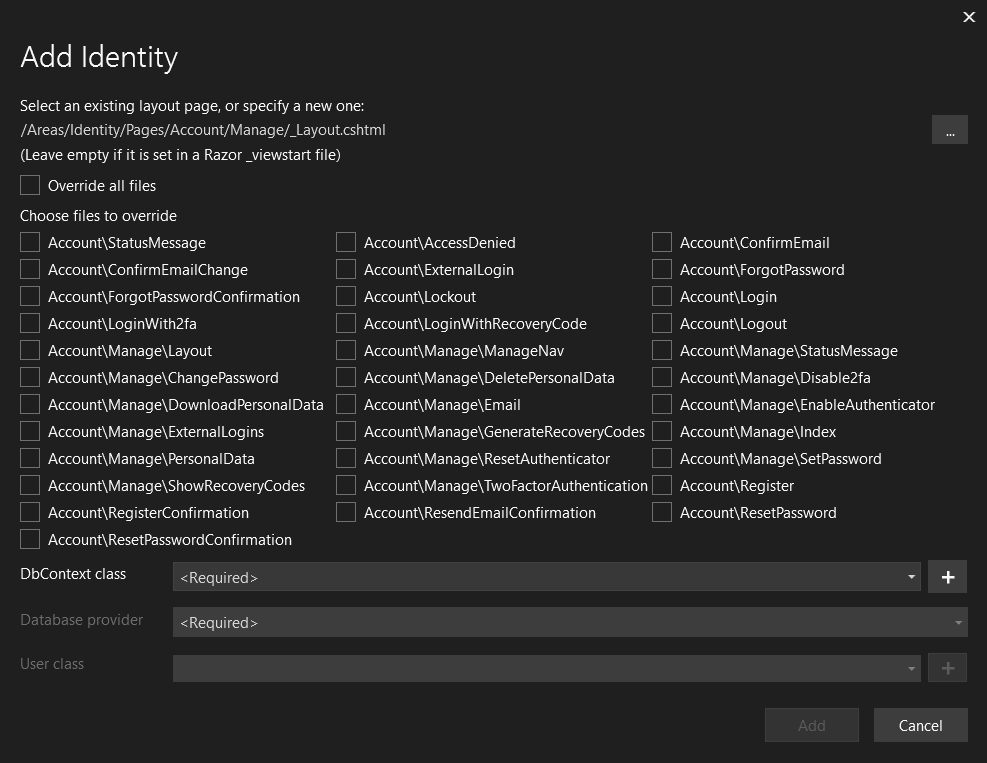
Теперь, когда Identity настроен, можно создать контроллеры и представления для регистрации и авторизации.

Использование Scaffolding для создания контроллеров и представлений

Чтобы сгенерировать контроллеры и представления для Identity, можно использовать встроенный инструмент codegenerator. Выполните следующие команды:





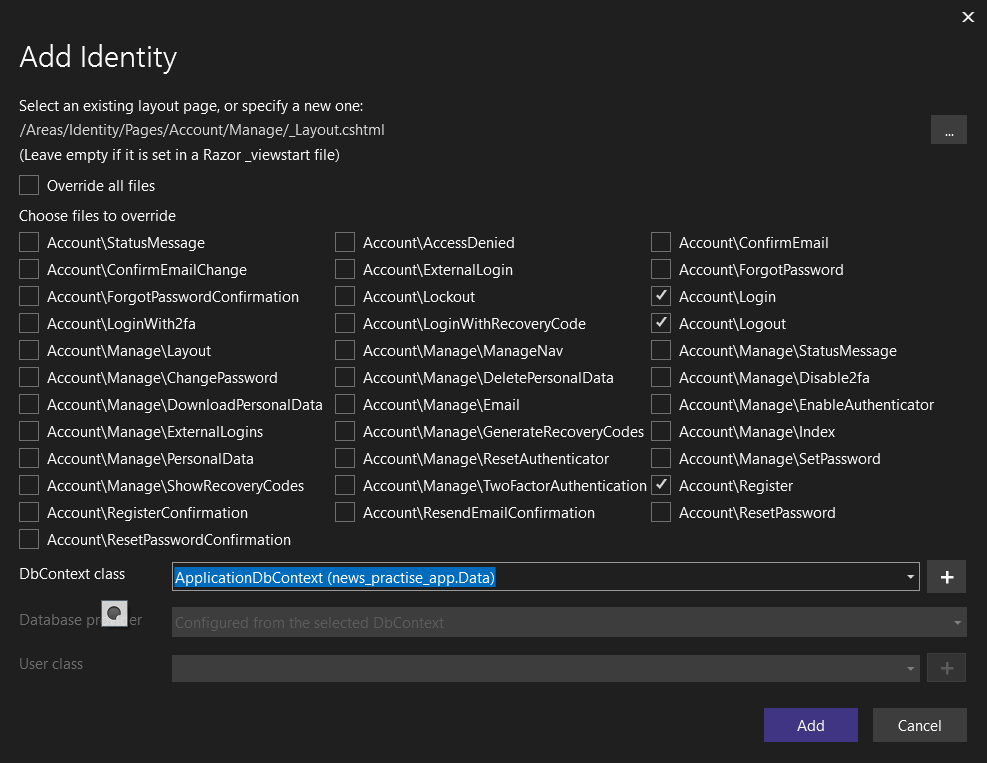


Чтобы минимизировать количество сгенерированных классов Identity для сайта новостей, можно выбрать только необходимые компоненты для базовой функциональности аутентификации и авторизации. Это позволит интегрировать систему пользователей без избыточных функций.

Что выбрать в окне Add Identity:

В окне Add Identity в Visual Studio можно увидеть множество галочек, каждая из которых представляет разные компоненты Identity (например, управление ролями, подтверждение email и т.д.). Для минимальной конфигурации, которая соответствует вашему SQL-скрипту, и для выполнения основных операций (регистрация, вход, выход, хранение данных о пользователях) необходимо выбрать следующие компоненты:

* Account/Login — для страницы авторизации пользователей.
* Account/Register — для регистрации новых пользователей.
* Account/Logout — для выхода из системы.

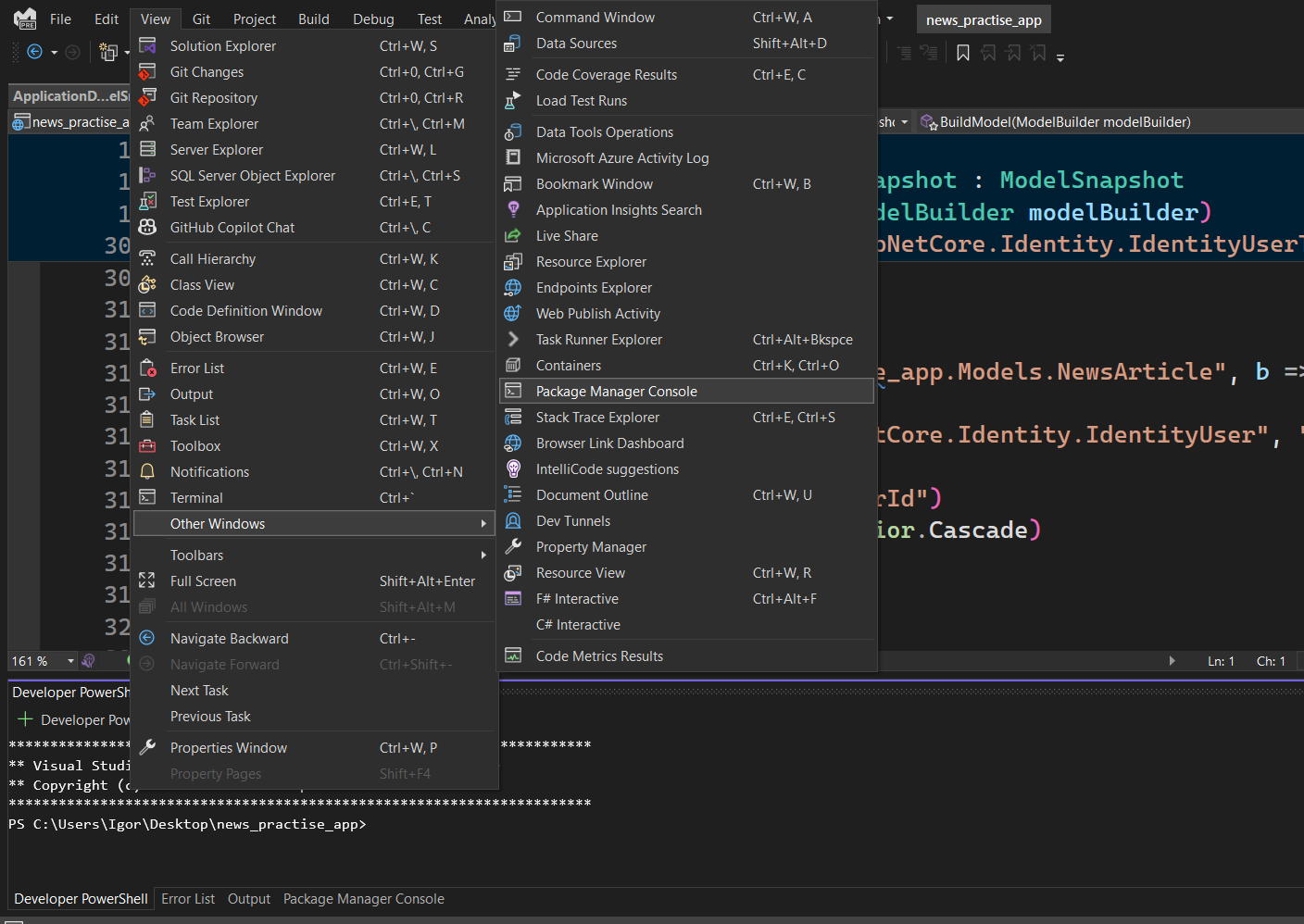


Применение миграций

Если вы еще не создавали и не применяли миграции, сделайте это:

Откройте Package Manager Console в Visual Studio (Tools → NuGet Package Manager → Package Manager Console).

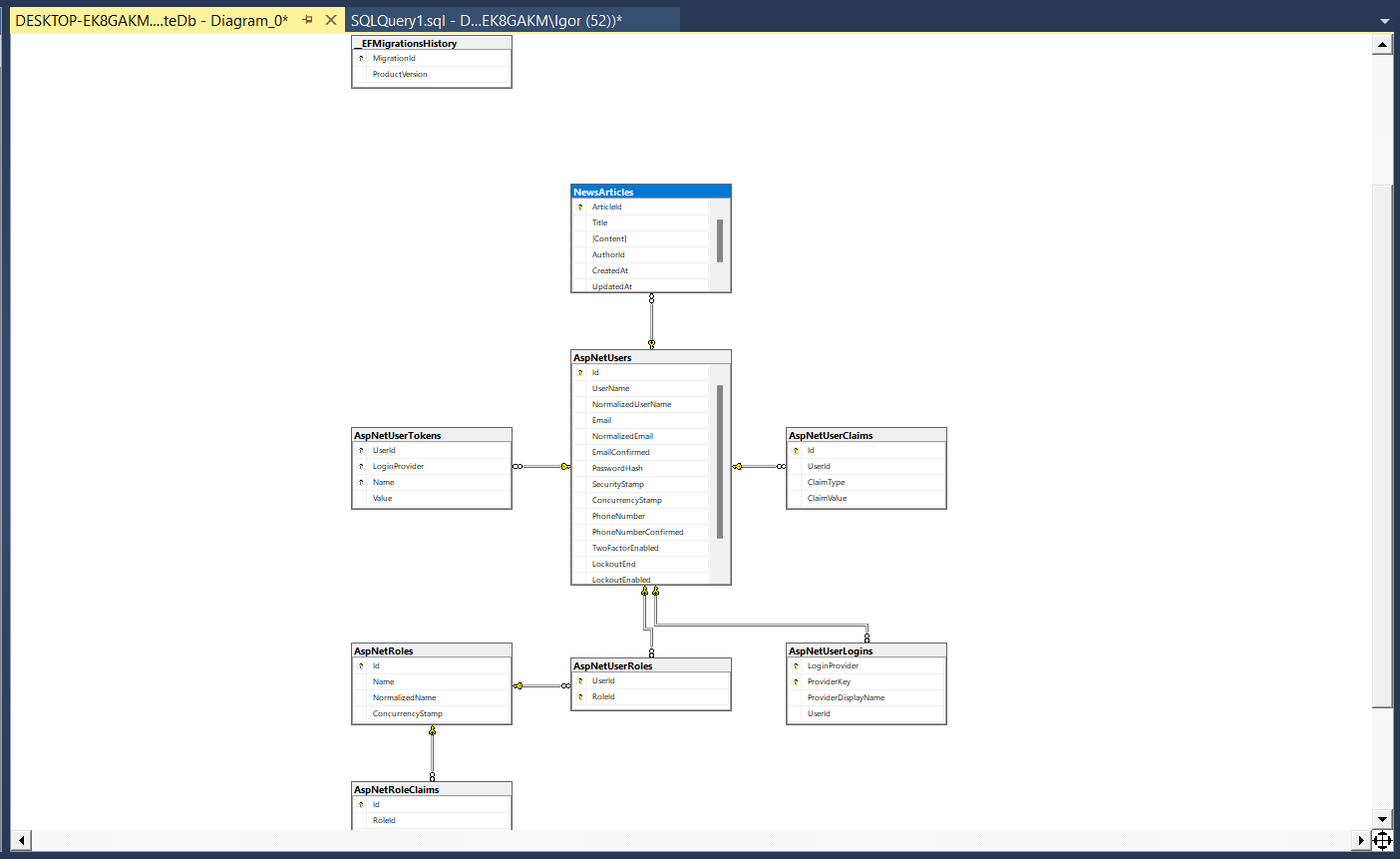
Выполните следующие команды для создания и применения миграций:



Add-Migration InitialCreate

Update-Database

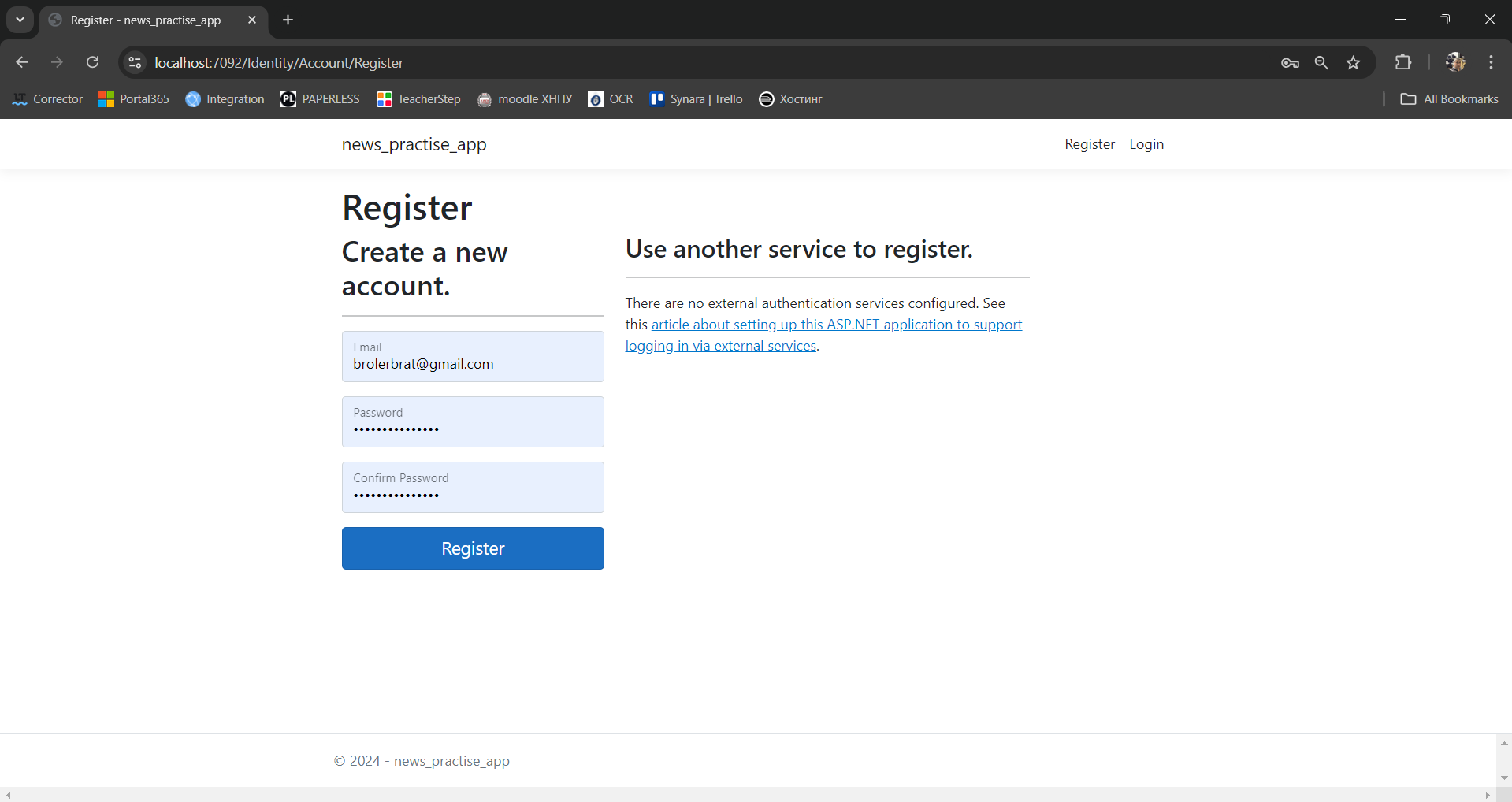
Это создаст таблицы для Identity (пользователей, ролей и их привязки).

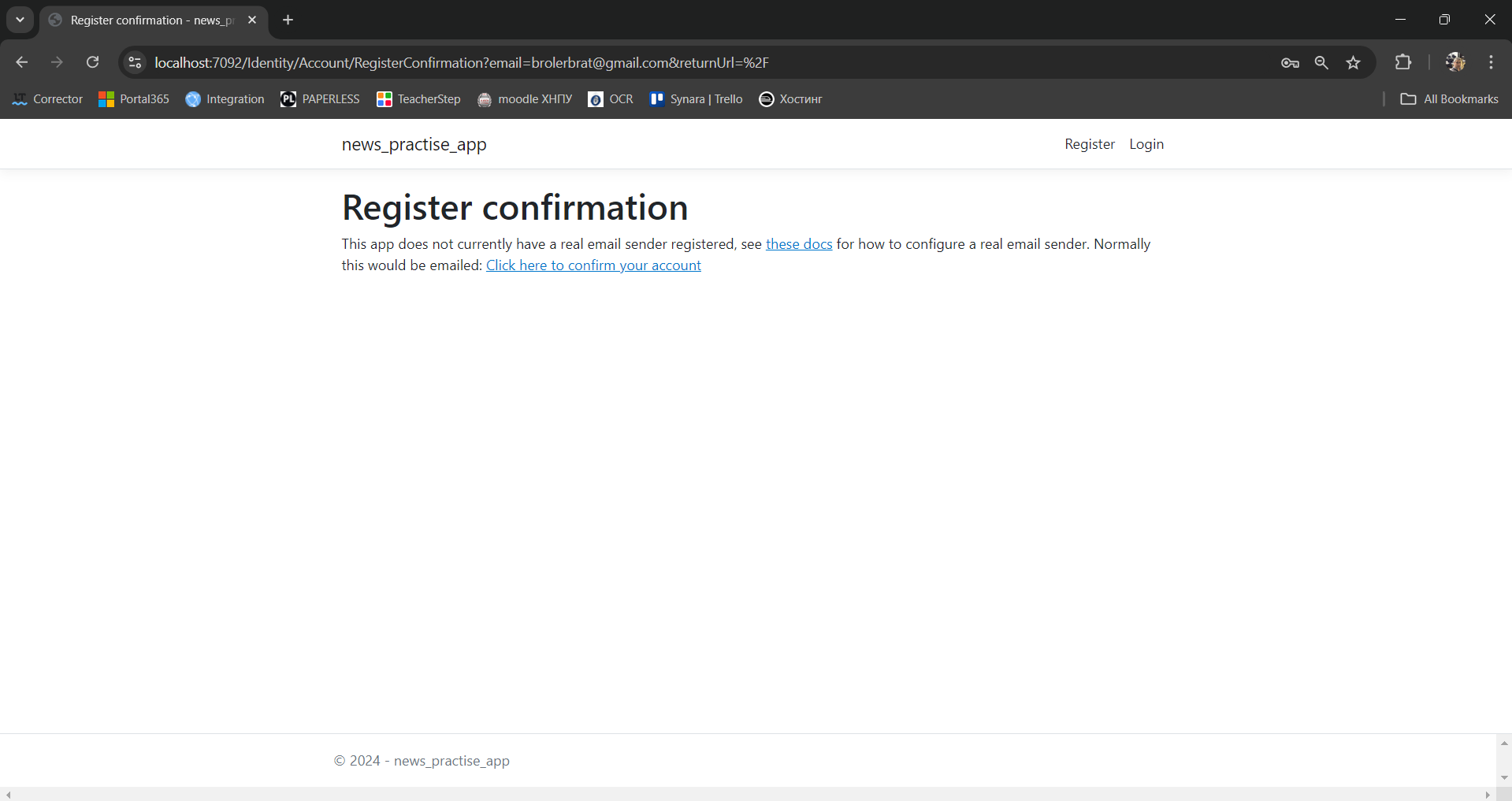


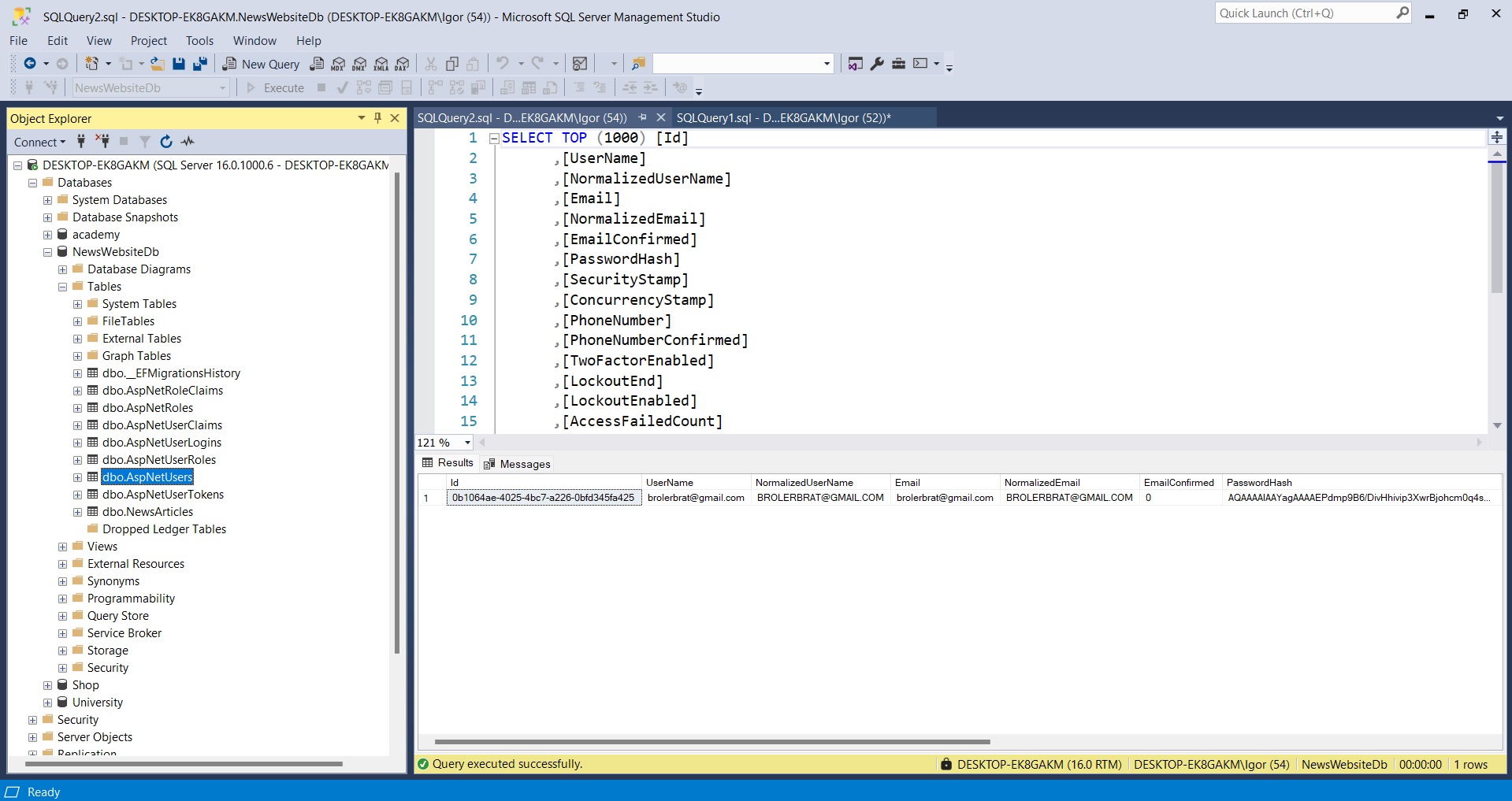
Проверка генерации и работы Identity

Убедитесь, что сгенерированные страницы для регистрации, авторизации и выхода из системы работают корректно. Запустите приложение и проверьте следующие URL:

* Регистрация: /Identity/Account/Register
* Вход: /Identity/Account/Login
* Выход: /Identity/Account/Logout





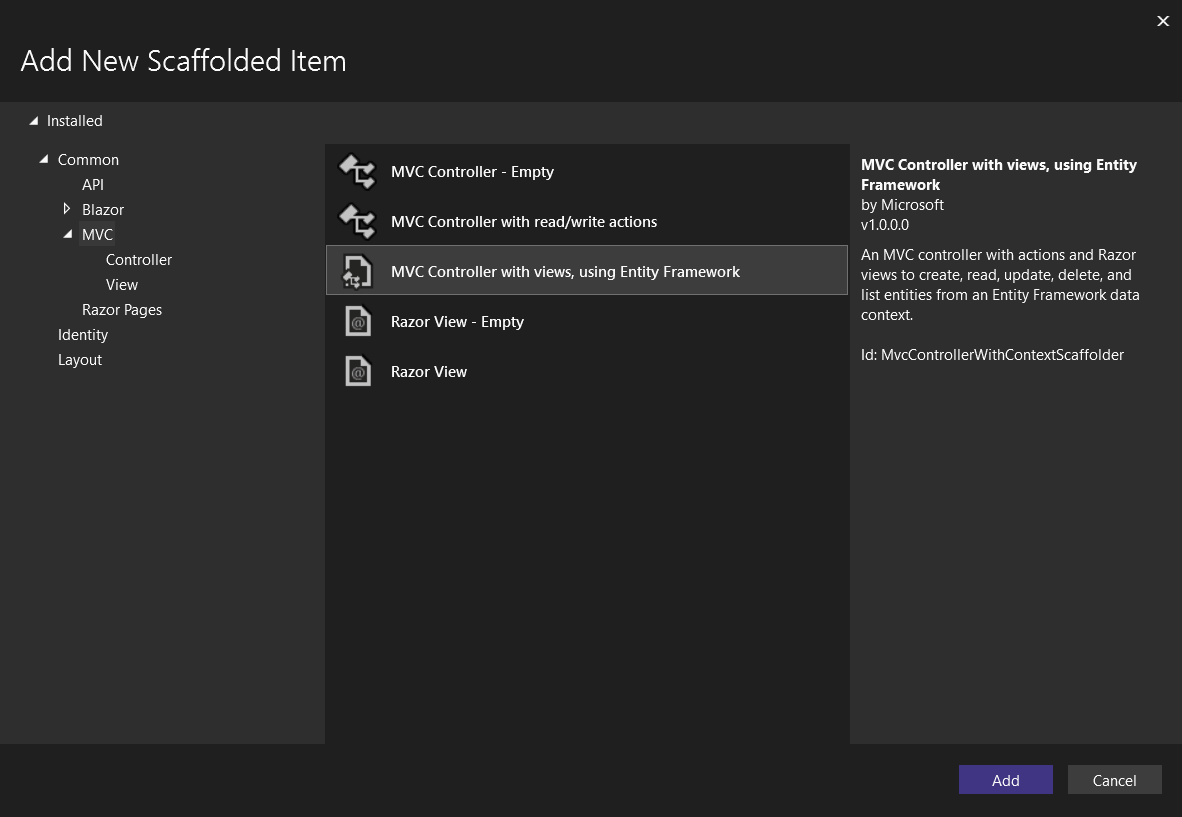


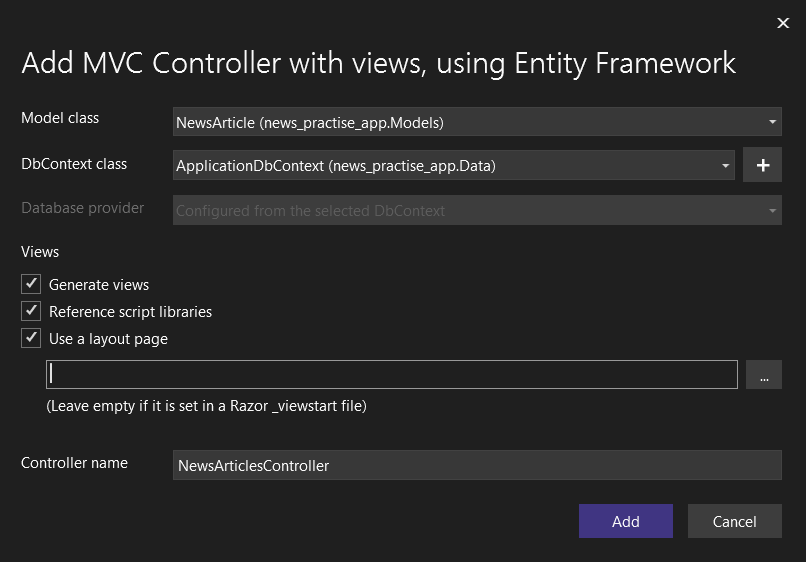
Добавление и настройка контроллеров и представлений:

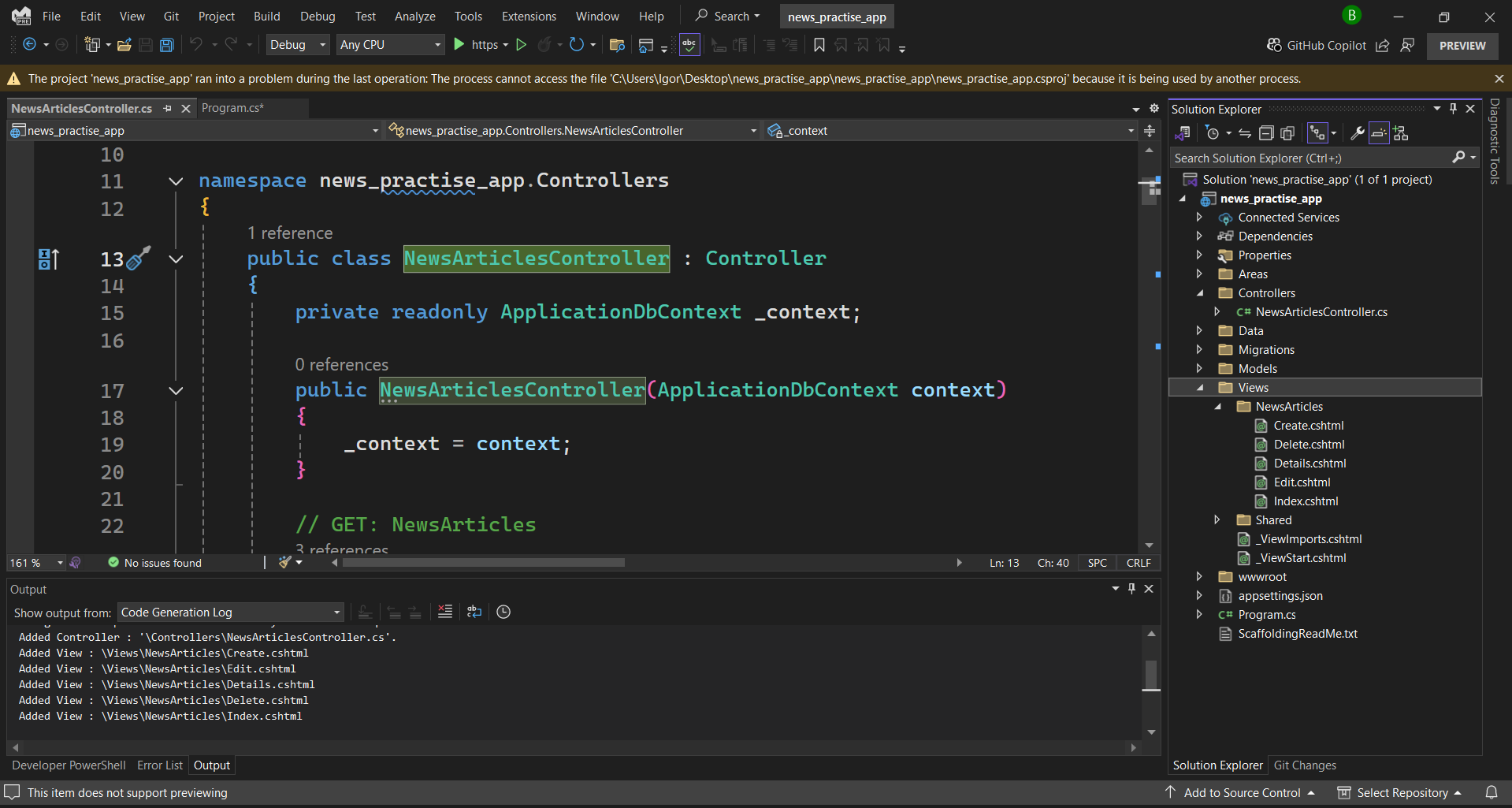
Контроллер, созданный с использованием Scaffolding, автоматически создаст представления для операций CRUD. Эти представления будут находиться в папке Views/News.

Вот краткое описание представлений, которые будут созданы:

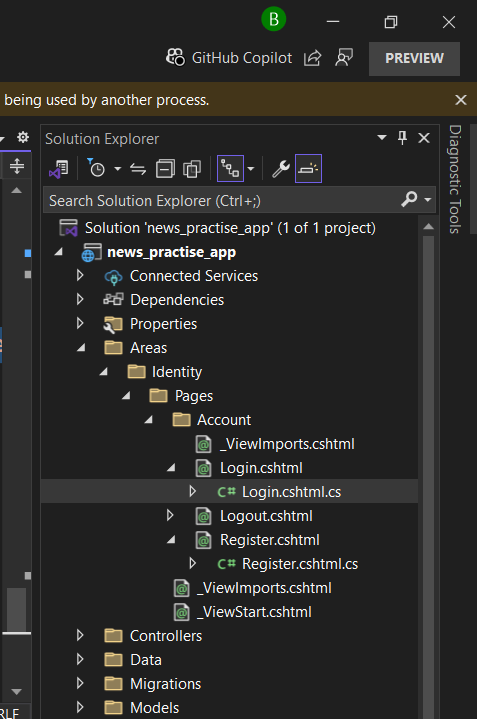
* Index.cshtml: список новостей.
* Details.cshtml: подробности новости.
* Create.cshtml: форма для создания новости.
* Edit.cshtml: форма для редактирования новости.
* Delete.cshtml: подтверждение удаления новости.







Обратите внимание т.к. Identity использует Razor Pages, а значит что вся логика регистрации и авторизации находиться в папке Areas:



Program.cs

using Microsoft.AspNetCore.Identity;

using Microsoft.EntityFrameworkCore;

using news\_practise\_app.Data;

namespace news\_practise\_app

{

    public class Program

    {

        public static void Main(string[] args)

        {

            var builder = WebApplication.CreateBuilder(args);

            // Настройка строки подключения к базе данных

            builder.Services.AddDbContext<ApplicationDbContext>(options =>

                options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));

            // Добавление Identity с настройками

            builder.Services.AddDefaultIdentity<IdentityUser>(options =>

            {

                options.SignIn.RequireConfirmedAccount = true; // Требовать подтверждения аккаунта

                // Дополнительные настройки Identity можно добавить здесь

                options.Password.RequireDigit = true;

                options.Password.RequiredLength = 6;

                options.Password.RequireLowercase = true;

                options.Password.RequireNonAlphanumeric = true;

                options.Password.RequireUppercase = true;

            })

            .AddEntityFrameworkStores<ApplicationDbContext>() // Использовать ApplicationDbContext для хранения данных Identity

            .AddDefaultTokenProviders(); // Добавление провайдеров токенов по умолчанию

            // Настройка MVC

            builder.Services.AddControllersWithViews();

            // Добавление поддержки сессий и куки

            builder.Services.AddSession(options =>

            {

                options.IdleTimeout = TimeSpan.FromMinutes(20); // Время жизни сессии

                options.Cookie.HttpOnly = true; // Защита от JavaScript

                options.Cookie.IsEssential = true; // Обязательные куки

            });

            var app = builder.Build();

            // Настройка промежуточного ПО

            if (!app.Environment.IsDevelopment())

            {

                app.UseExceptionHandler("/Home/Error");

                app.UseHsts();

            }

            else

            {

                app.UseDeveloperExceptionPage(); // Для разработки

            }

            app.UseHttpsRedirection(); // Перенаправление на HTTPS

            app.UseStaticFiles(); // Статические файлы

            app.UseRouting(); // Настройка маршрутизации

            // Включение сессий

            app.UseSession();

            // Включение аутентификации и авторизации

            app.UseAuthentication();

            app.UseAuthorization();

            // Настройка маршрутов

            app.MapControllerRoute(

                name: "default",

                pattern: "{controller=NewsArticles}/{action=Index}/{id?}");

            // Подключение Razor Pages для Identity

            app.MapRazorPages();

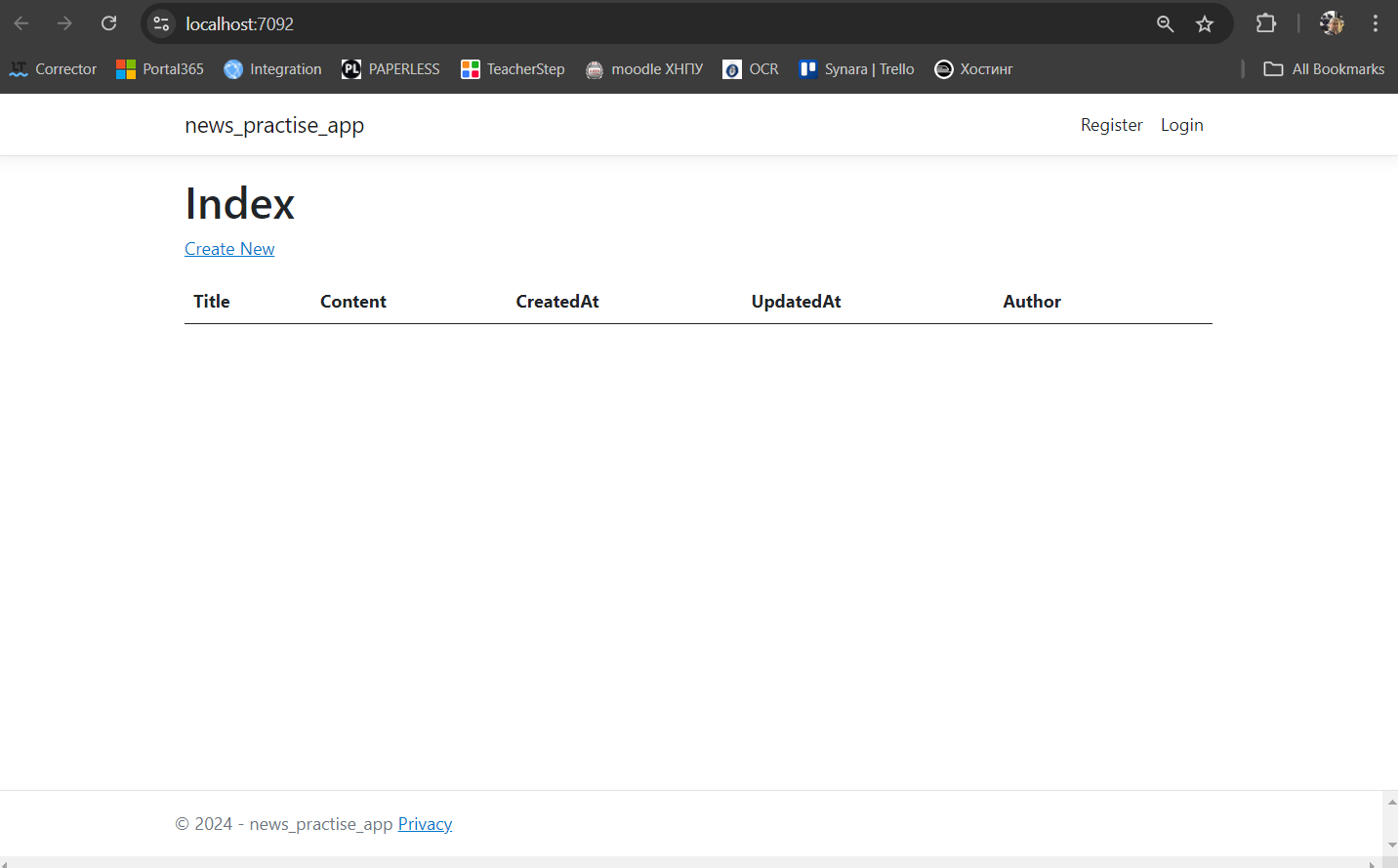
            app.Run();

        }

    }

}

Start:



Логгирование

ASP.NET Core имеет встроенную поддержку логгирования, что позволяет применять логгирование с минимальными вкраплениями кода в функционал приложения.

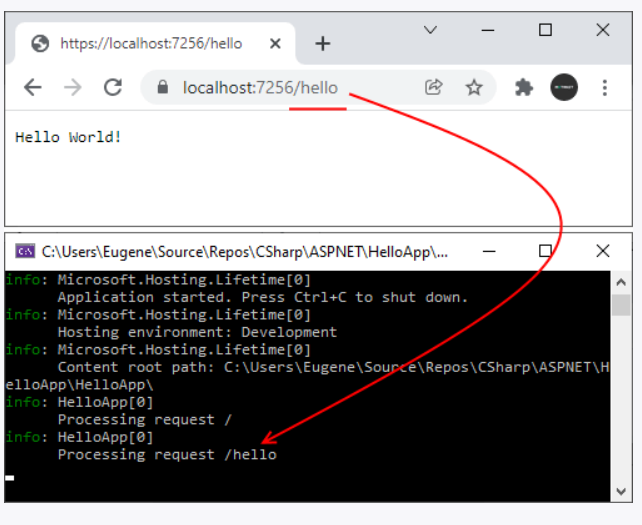
Для логгирования данных нам необходим объект **ILogger<T>**. По умолчанию среда ASP NET Core через механизм внедрения зависимостей уже предоставляет нам такой объект. Его можно получить как и любую другую зависимость в приложении. Также этот объект можно получить через свойство **Logger** объекта WebApplication.

Например, используем встроенный логгер для логгирования на консоль приложения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | var builder = WebApplication.CreateBuilder();  var app = builder.Build();    app.Run(async (context) =>  {      // пишем на консоль информацию      app.Logger.LogInformation($"Processing request {context.Request.Path}");        await context.Response.WriteAsync("Hello World!");  });    app.Run(); |

В данном случае через свойство **app.Logger** получаем встроенный логгер и с помощью его метода logger.LogInformation передаем на консоль некоторую информацию.

При обращении к приложению с помощью следующего запроса *http://localhost:xxxxx/hello* на консоль будет выведена информация, переданная логгером:

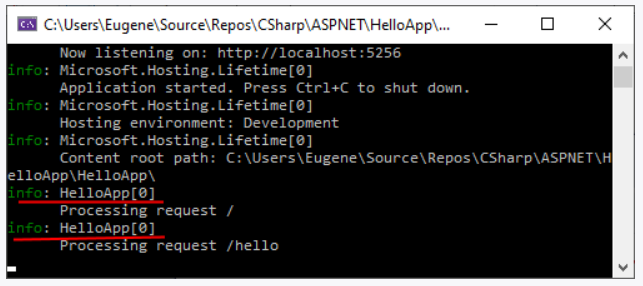


**Категория логгера**

При создании логгера для него указывается категория. Обычно в качестве категории логгера выступает класс, в котором используется логгер. В этом случае логгер типизируется классом-категории. Например, логгер, для которого в качестве категории выступает класс Program:

|  |  |
| --- | --- |
| 1 | ILogger<Program> |

В чем смысл категории? Категория задает текстовую метку, с которой ассоциируется сообщение логгера, и в выводе лога мы ее можем увидеть.



Где это может быть полезно? Например, у нас есть несколько классов middleware, где ведется логгирование. Указывая в качестве категории текущий класс, в последствии в логе мы можем увидеть, в каком классе именно было создано данное сообщение лога. Поэтому, как правило, в качестве категории указывается текущий класс, но в принципе это необязательно.

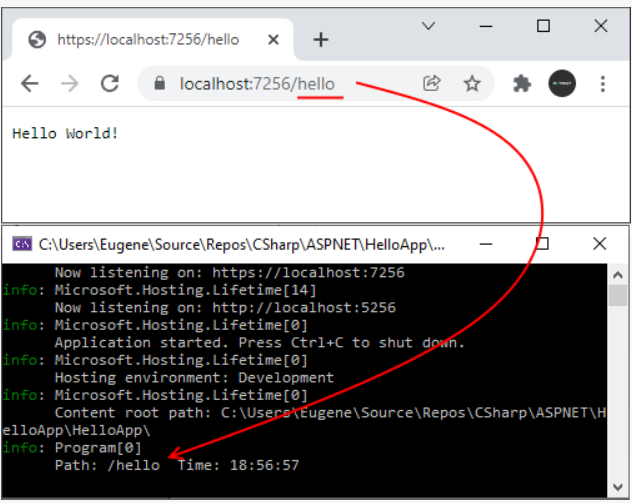
**Получение логгера через внедрение зависимостей**

Поскольку логгер добавляется в сервисы приложения, то мы можем получить его как и любой другой сервис через систему внедрения зависимостей. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | var builder = WebApplication.CreateBuilder();  var app = builder.Build();    app.Map("/hello", (ILogger<Program> logger) =>  {      logger.LogInformation($"Path: /hello  Time: {DateTime.Now.ToLongTimeString()}");      return "Hello World";  });    app.Run(); |

В данном случае при обращении по адресу "/hello" сработает конечная точка, в обработчике которой через механизм внедрения зависимостей можно получить объект логгера. Стоит учитывать, что в этом случае для логгера надо определить категорию. Здесь в качестве категории применяется класс Program (неявный класс, в котором и запускается приложение).

В самом обработчике логгер выводит на консоль путь запроса и время запроса:



### Уровни и методы логгирования

При настройке логгирования мы можем установить уровень детализации информации с помощью одного из значений перечисления **LogLevel**. Всего мы можем использовать следующие значения:

* Trace: используется для вывода наиболее детализированных сообщений. Подобные сообщения могут нести важную информацию о приложении и его строении, поэтому данный уровень лучше использовать при разработке, но никак не при публикации
* Debug: для вывода информации, которая может быть полезной в процессе разработки и отладки приложения
* Information: уровень сообщений, позволяющий просто отследить поток выполнения приложения
* Warning: используется для вывода сообщений о неожиданных событиях, например, ошибках, которые не останавливают выполнение приложения, но в то же время должны быть иследованы
* Error: для вывода информации об ошибках и исключениях, которые возникли при текущей операции и которые не могут быть обработаны
* Critical: уровень критических ошибок, которые требуют немедленной реакции - ошибками операционной системы, потерей данных в бд, переполнение памяти диска и т.д.
* None: вывод информации в лог не применяется

Для вывода соответствующего уровня информации у объекта ILogger определены соответствующие методы расширения:

* LogDebug()
* LogTrace()
* LogInformation()
* LogWarning()
* LogError()
* LogCritical()

Так, в примере выше для вывода информации на консоль использовался метод LogInformation().

Вывод сообщений уровня Trace по умолчанию отключен.

Каждый такой метод имеет несколько перегрузок, которые могут принимать ряд различных параметров:

* string data: строковое сообщение для лога
* int eventId: числовой идентификатор, который связан с логом. Идентификатор должен быть статическим и специфическим для определенной части логгируемых событий.
* string format: строковое сообщения для лога, которое может содержать параметры
* object[] args: набор параметров для строкового сообщения
* Exception error: логгируемый объект исключения

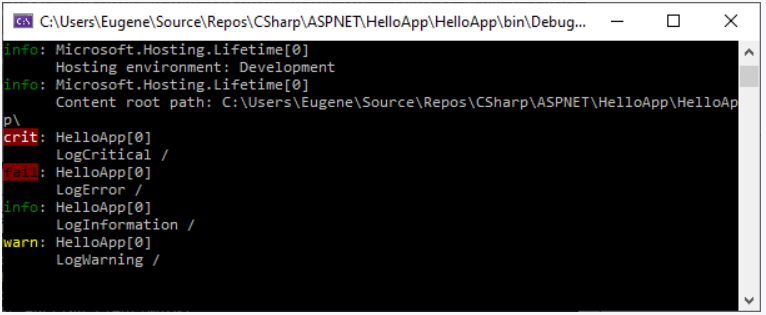
Также для логгирования определен общий метод **Log()**, который позволяет определить уровень логгера через один из параметров:

|  |  |
| --- | --- |
| 1 | logger.Log(LogLevel.Information, $"Requested Path: {context.Request.Path}"); |

При стандартном логгировании на консоль для каждого уровня/метода определен своя метка и цветовой маркер, которые позволяют сразу выделить сообщение соответствующего уровня. Например, при запуске следующего кода:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | var builder = WebApplication.CreateBuilder();  var app = builder.Build();    app.Run(async (context) =>  {      var path = context.Request.Path;      app.Logger.LogCritical($"LogCritical {path}");      app.Logger.LogError($"LogError {path}");      app.Logger.LogInformation($"LogInformation {path}");      app.Logger.LogWarning($"LogWarning {path}");        await context.Response.WriteAsync("Hello World!");  });    app.Run(); |

мы получим следующий лог на консоль:



## Фабрика логгера и провайдеры логгирования

В примерах в прошлой теме мы получали объект логгера, который добавляется через DI. Но мы можем также использовать фабрику логгера для его создания:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | var builder = WebApplication.CreateBuilder();  var app = builder.Build();    ILoggerFactory loggerFactory = LoggerFactory.Create(builder => builder.AddConsole());  ILogger logger = loggerFactory.CreateLogger<Program>();  app.Run(async (context) =>  {      logger.LogInformation($"Requested Path: {context.Request.Path}");      await context.Response.WriteAsync("Hello World!");  });    app.Run(); |

В данном случае с помощью метода **LoggerFactory.Create** создается фабрика логгера в виде объекта **ILoggerFactory**. В качестве параметра метод принимает делегат, который устанавливает некоторые настройки логгирования. В частности, метод **AddConsole()** объекта ILoggingBuilder устанавливает вывод сообщений лога на консоль. Затем метод CreateLogger() фабрики собственно создает логгер:

|  |  |
| --- | --- |
| 1 | ILogger logger = loggerFactory.CreateLogger<Program>(); |

Метод CreateLogger() типизируется классом, который представляет категорию. В данном случае это класс Program, в котором неявно выполняется данный код. Но в качестве альтернативы название категории можно передать в метод в качестве параметра в виде строки:

|  |  |
| --- | --- |
| 1 | ILogger logger = loggerFactory.CreateLogger("WebApplication"); |

В итоге мы получим тот же вывод сообщений на консоль. Но преимущество использования фабрики логгеров состоит в том, что мы можем дополнительно настроить различные параметры логгирования, в частности, провайдер логгирования.

### Получение фабрики логгера через dependency injection

Как и логгер, фабрика логгера доступна в приложении в виде сервиса, соответственно ее можно получит через механизм внедрения зависимостей:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | var builder = WebApplication.CreateBuilder();  var app = builder.Build();    app.Map("/hello", (ILoggerFactory loggerFactory)=>{        // создаем логгер с категорией "MapLogger"      ILogger logger = loggerFactory.CreateLogger("MapLogger");      // логгируем некоторое сообщение      logger.LogInformation($"Path: /hello   Time: {DateTime.Now.ToLongTimeString()}");      return "Hello World!";  });    app.Run(); |

### Провайдеры логгирования

В примере выше логгирование шло на консоль. Вообще путь логгирования определяется провайдером логгирования. По умолчанию ASP.NET Core предоставляет следующие провайдеры:

* **Console**: вывод информации на консоль. Устанавливается методом **AddConsole()**
* **Debug**: использует для ведения записей лога класс **System.Diagnostics.Debug** и в частности его метод Debug.WriteLine. Соответственно все записи лога мы можем увидеть в окне Output в Visual Studio. Устанавливается методом **AddDebug()**. Стоит отметить, что данный способ работает только при запуске проекта в режиме отладки
* **EventSource**: на Windows введет логгирование в лог [ETW](https://docs.microsoft.com/en-us/windows/win32/etw/event-tracing-portal) (Event Tracing for Windows), для просмотра которого может использоваться инструмент PerfView (или аналогичный инструменты). Хотя данный провайдер задумывался как кроссплатформенный, для Linux и MacOS пока назначение лога не определено. Устанавливается методом **AddEventSourceLogger()**
* **EventLog**: записывает в Windows Event Log, соответственно работает только при запуске на Windows. Устанавливается методом **AddEventLog()**

Например, вместо консоли зададим вывод лога в окне Output в Visual Studio:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | var builder = WebApplication.CreateBuilder();  var app = builder.Build();    var loggerFactory = LoggerFactory.Create(builder => builder.AddDebug());  ILogger logger = loggerFactory.CreateLogger<Program>();  app.Run(async (context) =>  {      logger.LogInformation($"Requested Path: {context.Request.Path}");      await context.Response.WriteAsync("Hello World!");  });    app.Run(); |

## Конфигурация и фильтрация логгирования

Для логгера можно задать конфигурацию с помощью одного из следующих источников:

* Файлы (json, xml)
* Аргументы командной строки
* Переменные среды окружения
* Объекты .NET
* Незашифрованное хранилище Secret Manager

Также можно создать свой провайдер конфигурации логгера.

Например, по умолчанию в проект по типу Empty добавляется файл **appsettings.json**, который предназначен для конфигурации проекта и в том числе может определять и по умолчанию уже содержит конфигурацию логгера:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | {    "Logging": {      "LogLevel": {        "Default": "Information",        "Microsoft.AspNetCore": "Warning"      }    },    "AllowedHosts": "\*"  } |

Узел **"Logging"** задает настройки логгирования. Далее узел **LogLevel** задает **минимальный уровень** логгирования для двух категорий. Категория "Default" представляет универсальное применение ко всем категориям. То есть по умолчанию будут логгироваться все сообщения уровня Information. Но для логгера с категорией "Microsoft.AspNetCore" будут логгироваться только сообщения уровня Warning.

Мы можем, например, убрать логгирование для категории "Microsoft.AspNetCore":

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | {    "Logging": {      "LogLevel": {        "Default": "Information",      }    },    "AllowedHosts": "\*"  } |

Или установить логгирование связки различных уровней и категорий для различных провайдеров:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | {    "Logging": {      "Debug": {        "LogLevel": {          "Default": "Debug"        }      },      "Console": {        "LogLevel": {          "Default": "Information",          "Microsoft.AspNetCore": "Warning"        }      },      "LogLevel": {        "Default": "Error"      }    },    "AllowedHosts": "\*"  } |

Здесь определяются четыре правила. Одно правило определено для провайдера Debug:

|  |  |
| --- | --- |
| 1  2  3  4  5 | "Debug": {      "LogLevel": {          "Default": "Debug"      }  } |

То есть для провайдера Debug определено логгирование сообщения уровня Debug для всех категорий.

Для провайдера Console определено два правила:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | "Console": {      "LogLevel": {          "Default": "Information",          "Microsoft.AspNetCore": "Warning"      }  } |

В данном случае для всех категорий будут логироваться сообщения уровня "Information". Для категории "Microsoft.AspNetCore" логируются сообщения уровня "Warning".

И также определено одно общее правило для всех провайдеров:

|  |  |
| --- | --- |
| 1  2  3 | "LogLevel": {      "Default": "Error"  } |

Логгирование сообщений уровня "Error" для всех категорий.

Следует учитывать, что настройки узла Logging.{имя\_провайдера}.LogLevel переопределяют настройки в узле Logging.LogLevel

Также можно настроить применяемые фильтры программно с помощью фабрики логгера:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | using Microsoft.Extensions.Logging.Debug;  //....................    var loggerFactory = LoggerFactory.Create(builder =>  {      builder.AddDebug();      builder.AddConsole();      // настройка фильтров      builder.AddFilter("System", LogLevel.Information)              .AddFilter<DebugLoggerProvider>("Microsoft", LogLevel.Trace);  });  ILogger logger = loggerFactory.CreateLogger("WebApplication"); |

Фильтр логгирования задается с помощью метода **AddFilter()**, который позволяет задать для определенной категории определенный уовень логгирования. Например:

|  |  |
| --- | --- |
| 1 | builder.AddFilter("System", LogLevel.Information) |

устанавает для категории "System" уровень LogLevel.Information для всех провайдеров. Это все равно, если бы мы написали в файле appsettings.json:

|  |  |
| --- | --- |
| 1  2  3  4 | "Logging": {      "LogLevel": {        "System": "Information"  } |

Второй вызов AddFilter обобщенный - обобщенный класс указывает на провайдер, для которого задается фильтр. То есть для провайдера DebugLoggerProvider для категории "Microsoft" устанавливается уровень LogLevel.Trace:

|  |  |
| --- | --- |
| 1 | AddFilter<DebugLoggerProvider>("Microsoft", LogLevel.Trace); |

Этот вызов аналогичен следующему определению в файле конфигурации

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | "Logging": {      "Debug": {        "LogLevel": {          "Microsoft": "Trace"      }  } |

Можно определять фильтры сразу и в коде C#, и в файле конфигурации. При создании логгера объект ILoggerFactory выбирает и применяет одно правило для каждого провайдера. Выбор нужного правила состоит из ряда этапов:

1. Выбираются все правила, которые соответствуют провайдеру. Если таких правил нет, то выбираются все правила, общие для всех провайдеров.
2. Из выбранных на предыдущем шаге правил выбирается правило, которое имеет наиболее длинное соответствие имени категории. Если такого нет, то выбираются все правила, общие для всех категорий.
3. Если на предыдущем шаге выбрано несколько правил, из них выбирается самое последнее (Порядковый номер правила соответствует его порядку определения в файле конфигурации или в коде. Причем правила из файла конфигурации предшествуют правилам, определенным в коде).
4. Если на предыдущем шаге не выбрано никаких правил, то применяется настройка MinimumLevel. Данную опцию также можно определить в коде, например, при настройке фабрики логгера:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | var loggerFactory = LoggerFactory.Create(builder =>  {      builder.AddDebug();      builder.AddFilter("System", LogLevel.Debug)              .SetMinimumLevel(LogLevel.Debug);   // Определение MinimumLevel    }); |

### Глобальная настройка логгирования

Для глобальной установки настроек логгирования у объекта WebApplicationBuilder определено свойсво **Logging**, которое представляет тип **ILogingBuilder**. Для управления логгирования он предоставляет ряд методов:

* AddConfiguration(): добавляет конфигурацию логгера в виде объекта IConfiguration
* AddConsole(): добавляет консольный логгер
* AddConsoleFormatter(): добавляет объект форматирования для консольного вывода
* AddJsonConsole(): добавляет форматирование сообщений консольного логгера в формат "json"
* AddSimpleConsole(): добавляет простое форматирование для логгирования на консоль
* AddDebug(): добавляет логгер отладки
* AddEventLog(): добавляет логгер для вывод в журнал Windows Event Log
* AddEventSourceLogger(): добавляет логгер для вывода в лог Event Tracing for Windows
* AddFilter(): добавляет фильтрацию для сообщений логгера
* AddProvider(): добавляет провайдер логгирования
* ClearProviders(): удаляет все зарегистрированные провайдеры логгирования
* SetMinimumLevel(): устанавливает минимальный уровень логгирования

Например:

|  |  |
| --- | --- |
| 1  2  3  4 | var builder = WebApplication.CreateBuilder();  builder.Logging.ClearProviders();   // удаляем все провайдеры  builder.Logging.AddConsole();   // добавляем провайдер для логгирования на консоль  var app = builder.Build(); |

## Создание провайдера логгирования

Стандартная инфраструктура ASP NET Core предоставляет, возможно, не самые удобные способы логгирования - на консоль, в окне Output в Visual Studio. Однако в то же время ASP.NET Core позволяет полностью определить свою логику ведения лога. Допустим, мы хотим сохранять сообщения в текстовом файле.

Вначале добавим в проект новый класс **FileLogger**:

|  |  |
| --- | --- |
| 1  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 | public class FileLogger : ILogger, IDisposable  {      string filePath;      static object \_lock = new object();      public FileLogger(string path)      {          filePath = path;      }      public IDisposable BeginScope<TState>(TState state)      {          return this;      }        public void Dispose() { }        public bool IsEnabled(LogLevel logLevel)      {          //return logLevel == LogLevel.Trace;          return true;      }        public void Log<TState>(LogLevel logLevel, EventId eventId,                  TState state, Exception? exception, Func<TState, Exception?, string> formatter)      {          lock (\_lock)          {              File.AppendAllText(filePath, formatter(state, exception) + Environment.NewLine);          }      }  } |

Класс логгера должен реализовать интерфейс **ILogger**. Этот интерфейс определяет три метода:

* BeginScope: этот метод возвращает объект IDisposable, который представляет некоторую область видимости для логгера. В данном случае нам этот метод не важен, поэтому возвращаем значение this - ссылку на текущий объект класса, который реализует интерфейс IDisposable.
* IsEnabled: возвращает значения true или false, которые указыват, доступен ли логгер для использования. Здесь можно здать различную логику. В частности, в этот метод передается объект LogLevel, и мы можем, к примеру, задействовать логгер в зависимости от значения этого объекта. Но в данном случае просто возвращаем true, то есть логгер доступен всегда.
* Log: этот метод предназначен для выполнения логгирования. Он принимает пять параметров:
  + LogLevel: уровень детализации текущего сообщения
  + EventId: идентификатор события
  + TState: некоторый объект состояния, который хранит сообщение
  + Exception: информация об исключении
  + formatter: функция форматирования, которая с помощью двух предыдущих параметов позволяет получить собственно сообщение для логгирования

И в данном методе как раз и производится запись в текстовый файл. Путь к этому файлу передается через конструктор

Далее добавим в проект класс **FileLoggerProvider**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public class FileLoggerProvider : ILoggerProvider  {      string path;      public FileLoggerProvider(string path)      {          this.path = path;      }      public ILogger CreateLogger(string categoryName)      {          return new FileLogger(path);      }        public void Dispose() {}  } |

Этот класс представляет провайдер логгирования. Он должен реализовать интерфейс **ILoggerProvider**. В этом интерфейсе определны два метода:

* **CreateLogger**: создает и возвращает объект логгера. Для создания логгера используется путь к файлу, который передается через конструктор
* **Dispose**: управляет освобождение ресурсов. В данном случае пустая реализация

Теперь создадим вспомогательный класс **FileLoggerExtensions**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public static class FileLoggerExtensions  {      public static ILoggingBuilder AddFile(this ILoggingBuilder builder, string filePath)      {          builder.AddProvider(new FileLoggerProvider(filePath));          return builder;      }  } |

Этот класс добавляет к объекту ILoggingBuilder метод расширения AddFile, который будет добавлять наш провайдер логгирования.

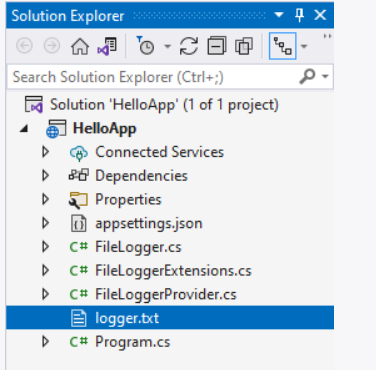
Теперь используем провайдер в файле **Program.cs**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | var builder = WebApplication.CreateBuilder();  // устанавливаем файл для логгирования  builder.Logging.AddFile(Path.Combine(Directory.GetCurrentDirectory(), "logger.txt"));  // настройка логгирования с помошью свойства Logging идет до  // создания объекта WebApplication  var app = builder.Build();    app.Run(async (context) =>  {      app.Logger.LogInformation($"Path: {context.Request.Path}  Time:{DateTime.Now.ToLongTimeString()}");      await context.Response.WriteAsync("Hello World!");  });    app.Run(); |

За глобальную установку настроек логгирования отвечает свойство **Logging** класса WebApplicationBuilder. Это свойство представляет объект ILoggingBuilder и предоставляет ряд методов для управления логгированием. И в данном случае с помошью выше определенного метода AddFile добавляем логгирование в файл.

Стоит отметить, что глобальная настройка логгирования должна идти до создания объекта WebApplication.

Теперь для логгирования также будет использоваться файл logger.txt, который будет создаваться в папке проекта.



Материал к уроку:

1. <https://learn.microsoft.com/ru-ru/aspnet/core/security/authentication/identity?view=aspnetcore-8.0&tabs=visual-studio>
2. <https://learn.microsoft.com/en-us/aspnet/core/security/authentication/identity?view=aspnetcore-8.0&tabs=visual-studio>

Домашнее задание:

Домашнее задание: Разработка Приложения для Управления Проектами

Цели:

Создать приложение для управления проектами с использованием ASP.NET Core Identity и Entity Framework.

Реализовать расширенные функции для работы с проектами и задачами.

Настроить безопасность и права доступа в приложении.

Задание:

Создание проекта:

Создайте новый проект ASP.NET Core MVC с использованием шаблона "Web Application" (с поддержкой Razor Pages).

Настройка ASP.NET Core Identity:

Добавьте и настройте ASP.NET Core Identity для управления пользователями и ролями. Убедитесь, что в проекте есть страницы для регистрации, входа в систему и выхода из системы.

Модель данных:

Создайте модели данных для управления проектами и задачами. Модели должны включать:

Project:

ProjectId (int, ключ)

Name (string, обязательное поле, максимальная длина 100 символов)

Description (string)

CreatedAt (DateTime, устанавливается автоматически при создании проекта)

UpdatedAt (DateTime?)

OwnerId (string, внешний ключ, связывающий проект с пользователем)

Task:

TaskId (int, ключ)

Title (string, обязательное поле, максимальная длина 100 символов)

Description (string)

DueDate (DateTime?)

IsCompleted (bool, по умолчанию false)

ProjectId (int, внешний ключ, связывающий задачу с проектом)

AssignedUserId (string, внешний ключ, связывающий задачу с пользователем)

User (пользователь из Identity):

Связь с проектами и задачами через OwnerId и AssignedUserId.

Работа с Entity Framework:

Создайте контекст базы данных (ApplicationDbContext), который будет включать таблицы для пользователей, проектов и задач.

Реализуйте миграции для создания и обновления схемы базы данных. Убедитесь, что таблицы создаются корректно.

Функциональность управления проектами и задачами:

Реализуйте CRUD (создание, чтение, обновление, удаление) операции для проектов и задач. Добавьте соответствующие действия в контроллеры и представления для работы с проектами и задачами.

Для задач реализуйте возможность назначения их пользователям и установки сроков выполнения.

Настройка ролей и прав доступа:

Добавьте роли (например, "Admin", "Project Manager", "Team Member"). Реализуйте возможность управления ролями и правами доступа.

Настройте права доступа так, чтобы только владельцы проектов могли редактировать и удалять свои проекты, а также назначать задачи другим пользователям.

Интерфейс и UX:

Создайте удобный интерфейс для управления проектами и задачами с использованием Bootstrap или другой CSS библиотеки.

Реализуйте функционал для фильтрации и сортировки проектов и задач.

Уведомления:

Добавьте функционал для отправки уведомлений пользователям о новых задачах, изменениях в проектах и приближающихся сроках выполнения задач.

