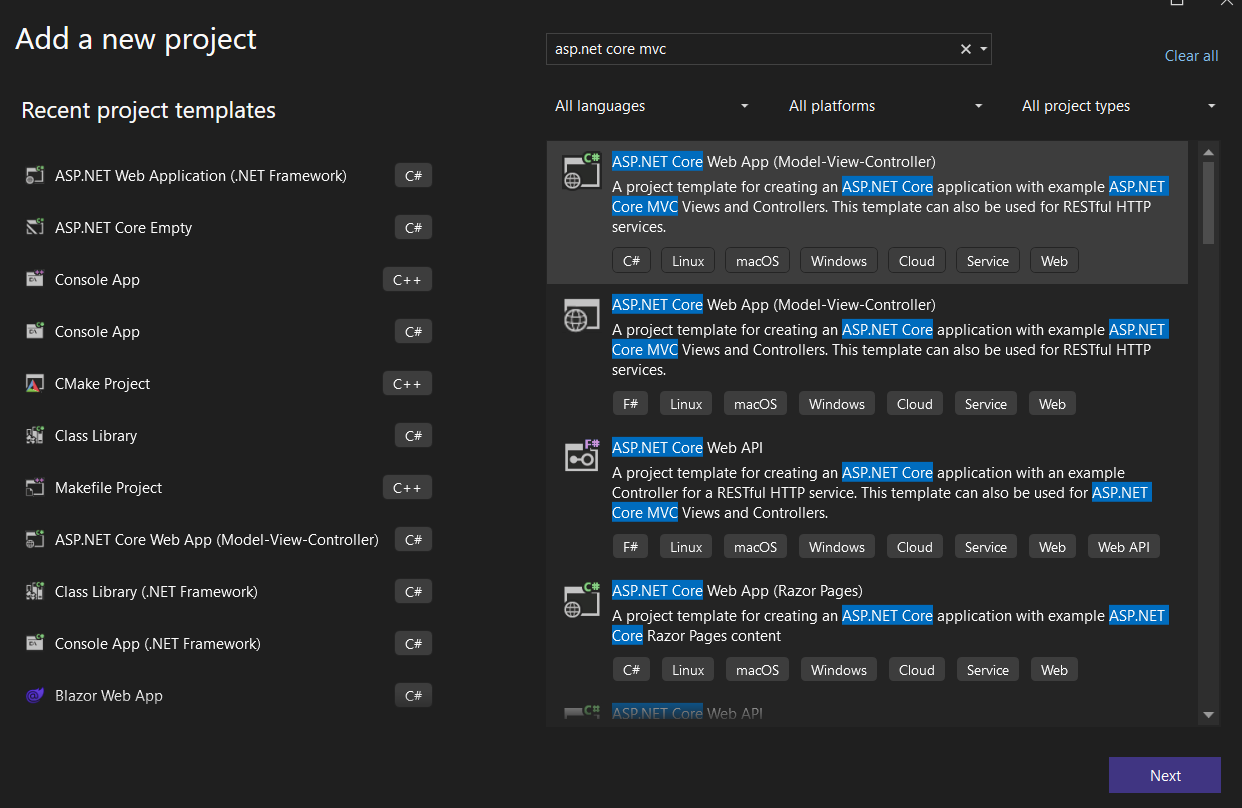


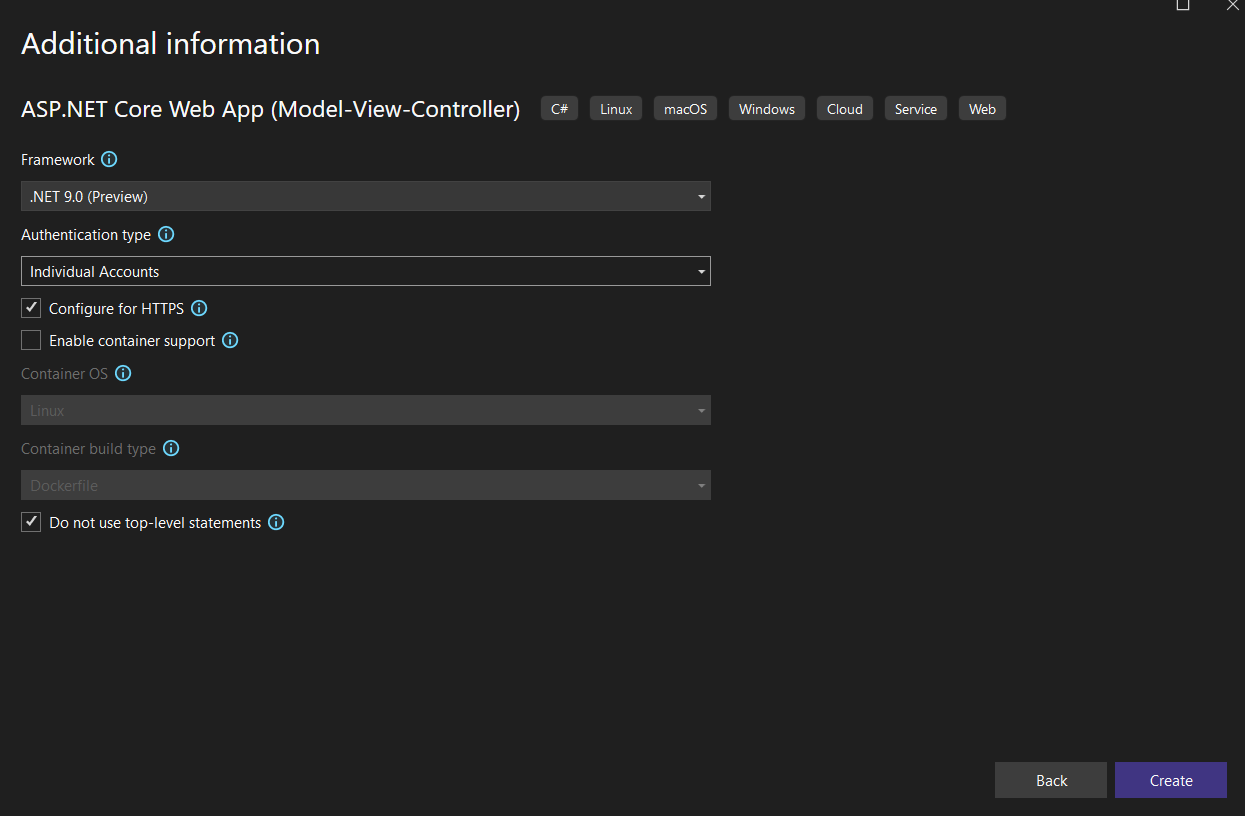
ASP.NET Core Identity — это мощная система управления пользователями и ролями, встроенная в ASP.NET Core, которая предоставляет готовые механизмы для аутентификации, авторизации и управления данными пользователей. Она обеспечивает работу с регистрацией, входом, управлением ролями, а также поддерживает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





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

Update-Database

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;

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

    public UserController(UserManager<IdentityUser> userManager)

    {

        \_userManager = userManager;  // Присваиваем экземпляр UserManager 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();

    }

    // 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 };

        // 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);

    }

    // 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

@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>

Создание представления для сообщений

Чтобы показывать сообщения об успешных или неудачных операциях, можно создать частичное представление. Это будет, например, \_MessagePartial.cshtml.

@model string <!-- Модель для сообщения -->

@if (!string.IsNullOrEmpty(Model))

{

    <div class="alert">

        @Model

    </div>

}

Изменение Index.cshtml для отображения сообщений

Теперь обновим Index.cshtml, чтобы включить частичное представление для сообщений.

@model Tuple<List<Microsoft.AspNetCore.Identity.IdentityUser>, string> <!-- Модель - список пользователей и сообщение -->

<!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>

    <!-- Отображение сообщения -->

    @Html.Partial("\_MessagePartial", Model.Item2)

    <!-- 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.Item1)

            {

                <tr>

                    <td>@user.Email</td>

                    <td>

                        <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>

                        <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>

Если у вас есть кастомные маршруты, убедитесь, что они настроены в методе Configure в файле Program.cs. Обычно маршруты настраиваются по умолчанию, но если вы изменяли настройки, добавьте:

app.UseEndpoints(endpoints =>

{

    endpoints.MapControllerRoute(

        name: "default",

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

});

Карта Url:

Главная страница:

- URL: /                    => HomeController.Index

- URL: /home/index          => HomeController.Index

- URL: /home/privacy        => HomeController.Privacy

- URL: /home/error          => HomeController.Error

Управление пользователями:

- URL: /user/index          => UserController.Index

- URL: /user/create         => UserController.Create (POST)

- URL: /user/update         => UserController.Update (POST)

  - Параметры: userId, newEmail

- URL: /user/delete         => UserController.Delete (POST)

  - Параметры: userId

По желанию можно добавить карту на страницу Home:

@{

    ViewData["Title"] = "Карта сайта";

}

<h1>Карта сайта</h1>

<ul>

    <li><a href="/">Главная страница</a></li>

    <li><a href="/home/privacy">Политика конфиденциальности</a></li>

    <li><a href="/user/index">Управление пользователями</a></li>

    <li><a href="/user/create">Создать пользователя</a></li>

</ul>

Чтобы использовать SignInManager в ASP.NET Core для управления процессом входа и выхода пользователей, вам нужно следовать нескольким шагам. SignInManager отвечает за аутентификацию пользователей, управление их сессиями и настройку политик входа.

Конфигурация Identity

В вашем Program.cs настройте Identity:

Обновление контроллера 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

{

    // Поля для хранения экземпляров UserManager и SignInManager

    private readonly UserManager<IdentityUser> \_userManager;

    private readonly SignInManager<IdentityUser> \_signInManager;

    // Конструктор контроллера. Через Dependency Injection (DI) внедряем UserManager и SignInManager

    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();

    }

    // 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 };

        // 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);

    }

    // POST-метод для входа пользователя

    [HttpPost]

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

    {

        // Проверяем, что email и пароль были переданы

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

        {

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

        }

        // Попытка аутентификации пользователя с помощью SignInManager

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

        // Если вход успешен

        if (result.Succeeded)

        {

            return Ok("Пользователь вошел в систему.");

        }

        // Если вход не удался

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

    }

    // POST-метод для выхода пользователя

    [HttpPost]

    [ValidateAntiForgeryToken]

    public async Task<IActionResult> Logout()

    {

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

        return Ok("Пользователь вышел из системы.");

    }

    // 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.");

    }

}

Program.cs  
using Microsoft.AspNetCore.Identity;  // Подключаем пространство имен для работы с Identity

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

using WebApplication3.Data;           // Подключаем пространство имен для доступа к контексту базы данных

namespace WebApplication3  // Объявляем пространство имен для нашего приложения

{

    public class Program  // Определяем основной класс программы

    {

        public static void Main(string[] args)  // Главный метод, который запускает приложение

        {

            var builder = WebApplication.CreateBuilder(args);  // Создаем строителя приложения

            // Добавляем службы в контейнер.

            // Получаем строку подключения к базе данных из конфигурации.

            var connectionString = builder.Configuration.GetConnectionString("DefaultConnection")

                ?? throw new InvalidOperationException("Connection string 'DefaultConnection' not found.");

            // Настраиваем контекст базы данных с использованием SQL Server и полученной строки подключения.

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

                options.UseSqlServer(connectionString));

            // Добавляем фильтр для отладки базы данных в режиме разработки.

            builder.Services.AddDatabaseDeveloperPageExceptionFilter();

            // Добавляем поддержку Identity с настройками по умолчанию.

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

            {

                options.SignIn.RequireConfirmedAccount = true; // Требовать подтверждение аккаунта при входе

            })

            .AddEntityFrameworkStores<ApplicationDbContext>(); // Указываем контекст базы данных для Identity

            // Добавляем поддержку контроллеров и представлений.

            builder.Services.AddControllersWithViews();

            var app = builder.Build();  // Создаем экземпляр приложения

            // Настраиваем конвейер HTTP-запросов.

            if (app.Environment.IsDevelopment())  // Проверяем, находимся ли мы в режиме разработки

            {

                app.UseMigrationsEndPoint();  // Используем конечную точку миграций

            }

            else  // Если не в режиме разработки

            {

                app.UseExceptionHandler("/Home/Error"); // Указываем страницу ошибки

                // Настраиваем HSTS (HTTP Strict Transport Security) для повышения безопасности

                app.UseHsts();

            }

            app.UseHttpsRedirection();  // Перенаправляем все HTTP-запросы на HTTPS

            app.UseRouting();  // Включаем маршрутизацию

            app.UseAuthorization();  // Включаем авторизацию

            // Настраиваем маршрутизацию статических файлов

            app.MapStaticAssets();

            // Настраиваем маршрут для контроллеров

            app.MapControllerRoute(

                name: "default",

                pattern: "{controller=Home}/{action=Index}/{id?}") // Определяем маршрут по умолчанию

                .WithStaticAssets();  // Добавляем статические ресурсы

            app.MapRazorPages()  // Настраиваем маршруты для Razor страниц

               .WithStaticAssets();  // Добавляем статические ресурсы

            app.Run();  // Запускаем приложение

        }

    }

}

Program.cs

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

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

using WebApplication3.Data;           // Подключаем пространство имен для доступа к контексту базы данных

namespace WebApplication3  // Объявляем пространство имен для нашего приложения

{

    public class Program  // Определяем основной класс программы

    {

        public static void Main(string[] args)  // Главный метод, который запускает приложение

        {

            var builder = WebApplication.CreateBuilder(args);  // Создаем строителя приложения

            // Добавляем службы в контейнер.

            // Получаем строку подключения к базе данных из конфигурации.

            var connectionString = builder.Configuration.GetConnectionString("DefaultConnection")

                ?? throw new InvalidOperationException("Connection string 'DefaultConnection' not found.");

            // Настраиваем контекст базы данных с использованием SQL Server и полученной строки подключения.

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

                options.UseSqlServer(connectionString));

            // Добавляем фильтр для отладки базы данных в режиме разработки.

            builder.Services.AddDatabaseDeveloperPageExceptionFilter();

            // Добавляем поддержку Identity с настройками по умолчанию.

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

            {

                options.SignIn.RequireConfirmedAccount = true; // Требовать подтверждение аккаунта при входе

            })

            .AddEntityFrameworkStores<ApplicationDbContext>(); // Указываем контекст базы данных для Identity

            // Добавляем поддержку контроллеров и представлений.

            builder.Services.AddControllersWithViews();

            var app = builder.Build();  // Создаем экземпляр приложения

            // Настраиваем конвейер HTTP-запросов.

            if (app.Environment.IsDevelopment())  // Проверяем, находимся ли мы в режиме разработки

            {

                app.UseMigrationsEndPoint();  // Используем конечную точку миграций

            }

            else  // Если не в режиме разработки

            {

                app.UseExceptionHandler("/Home/Error"); // Указываем страницу ошибки

                // Настраиваем HSTS (HTTP Strict Transport Security) для повышения безопасности

                app.UseHsts();

            }

            app.UseHttpsRedirection();  // Перенаправляем все HTTP-запросы на HTTPS

            app.UseRouting();  // Включаем маршрутизацию

            app.UseAuthorization();  // Включаем авторизацию

            // Настраиваем маршрутизацию статических файлов

            app.MapStaticAssets();

            // Настраиваем маршрут для контроллеров

            app.MapControllerRoute(

                name: "default",

                pattern: "{controller=Home}/{action=Index}/{id?}") // Определяем маршрут по умолчанию

                .WithStaticAssets();  // Добавляем статические ресурсы

            app.MapRazorPages()  // Настраиваем маршруты для Razor страниц

               .WithStaticAssets();  // Добавляем статические ресурсы

            app.Run();  // Запускаем приложение

        }

    }

}

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. Давайте разберём, как это сделать.

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>

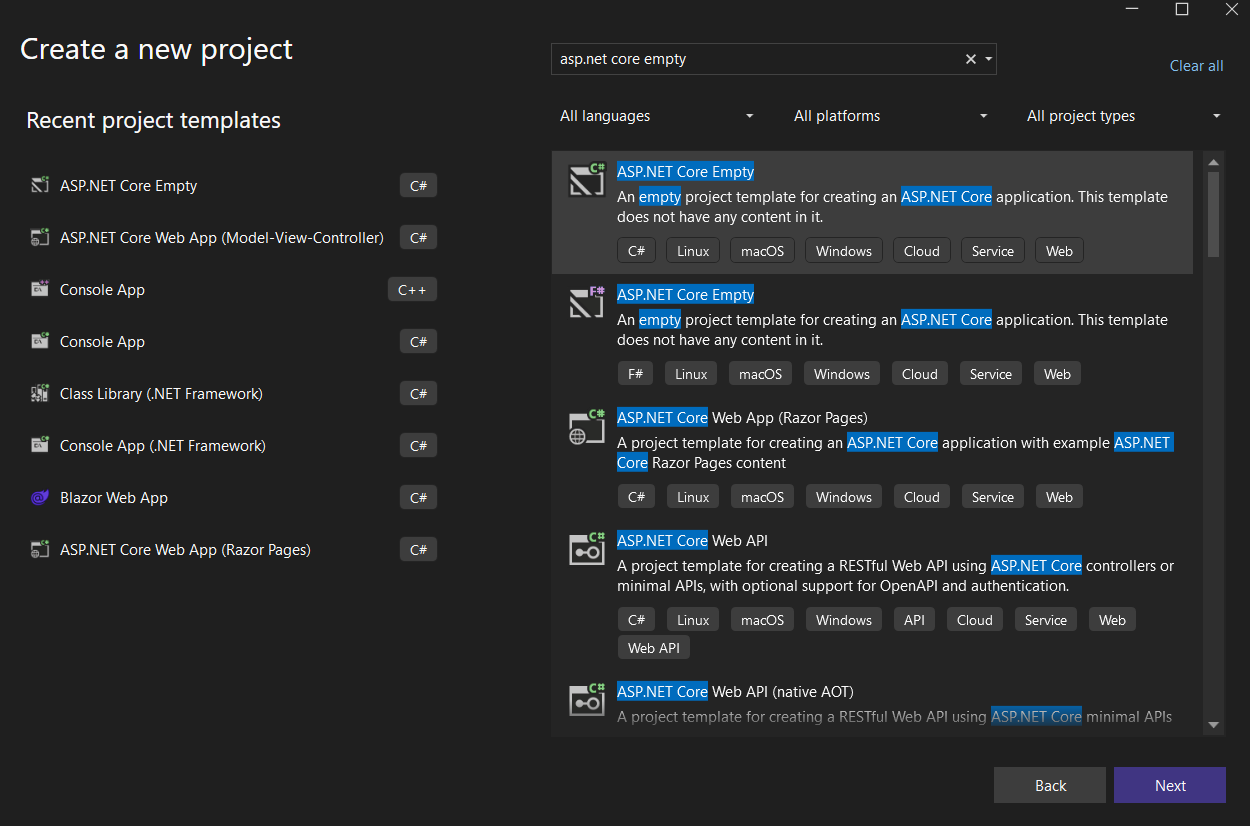
        <input type="text" id="roleName" name="roleName" required />

    </div>

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

</form>

Практика



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();

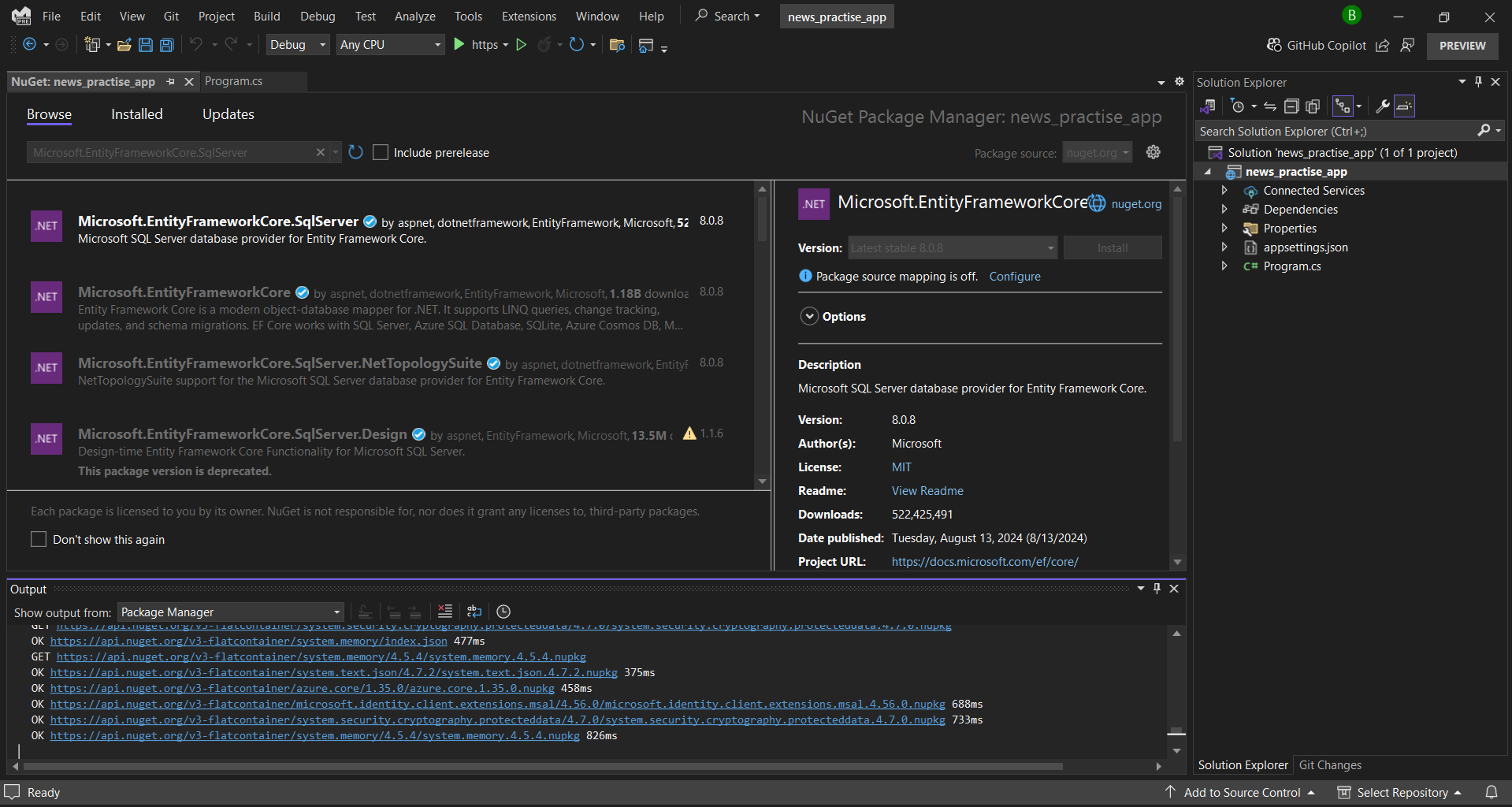
        }

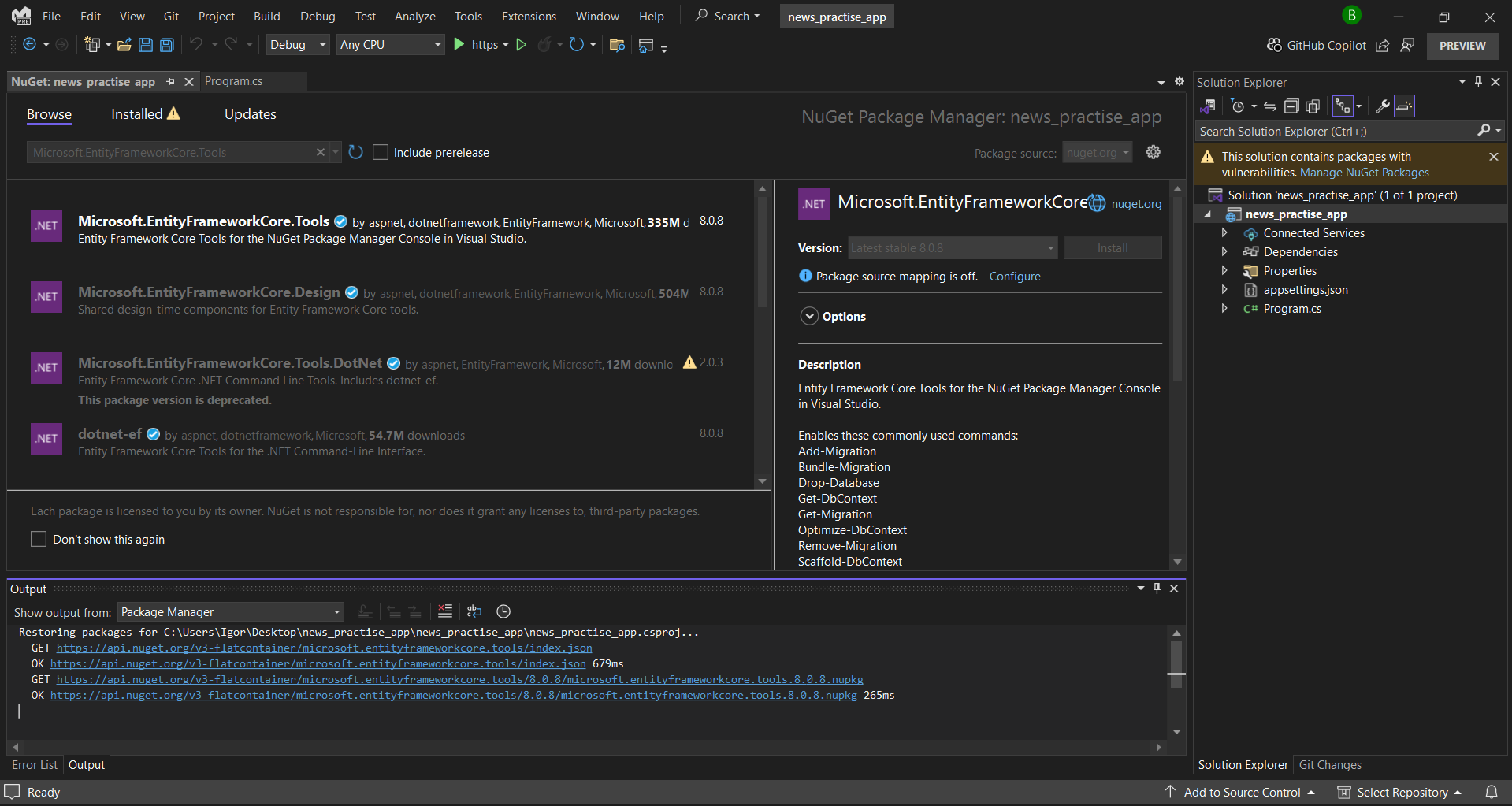
    }

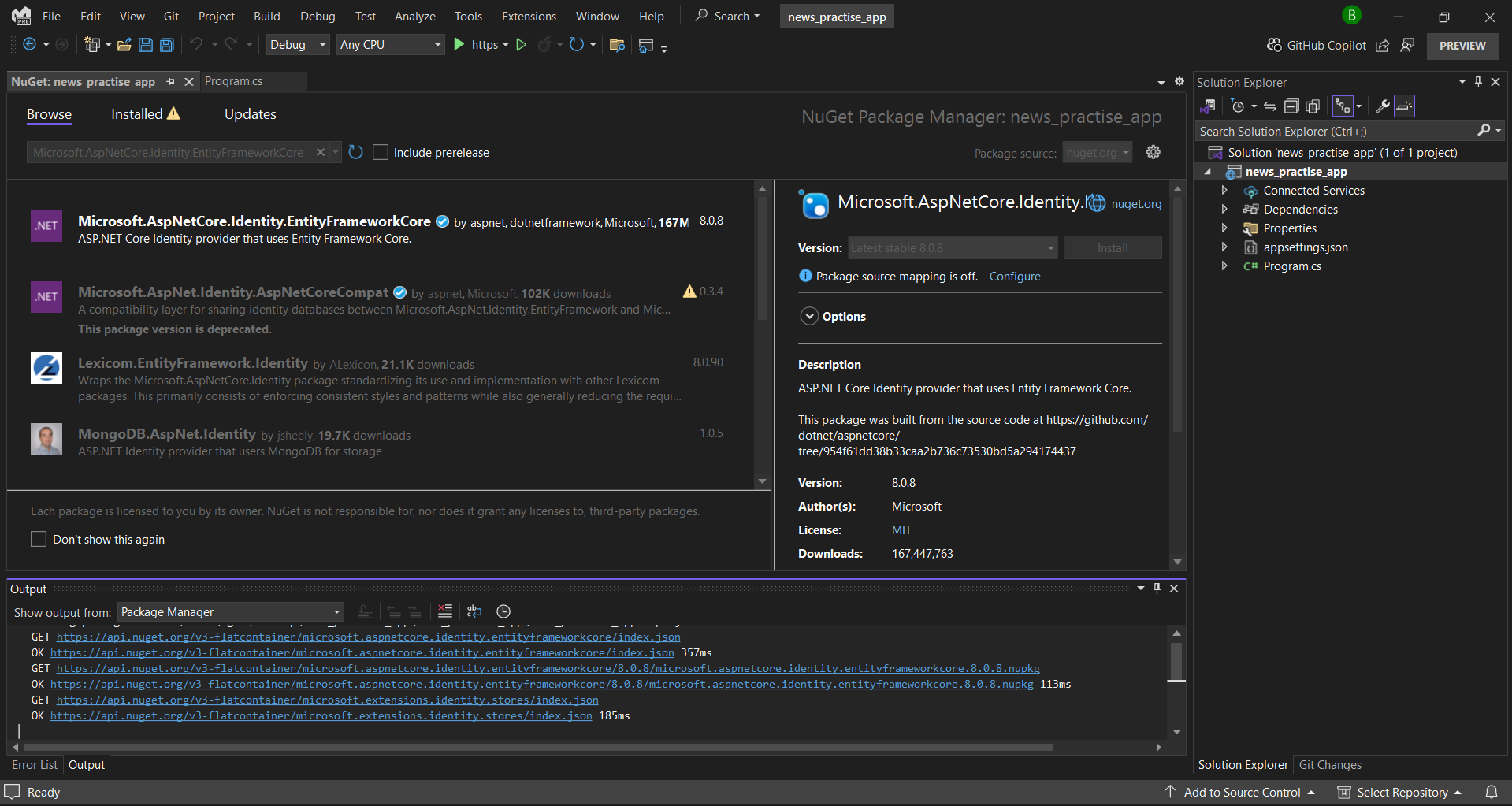
}

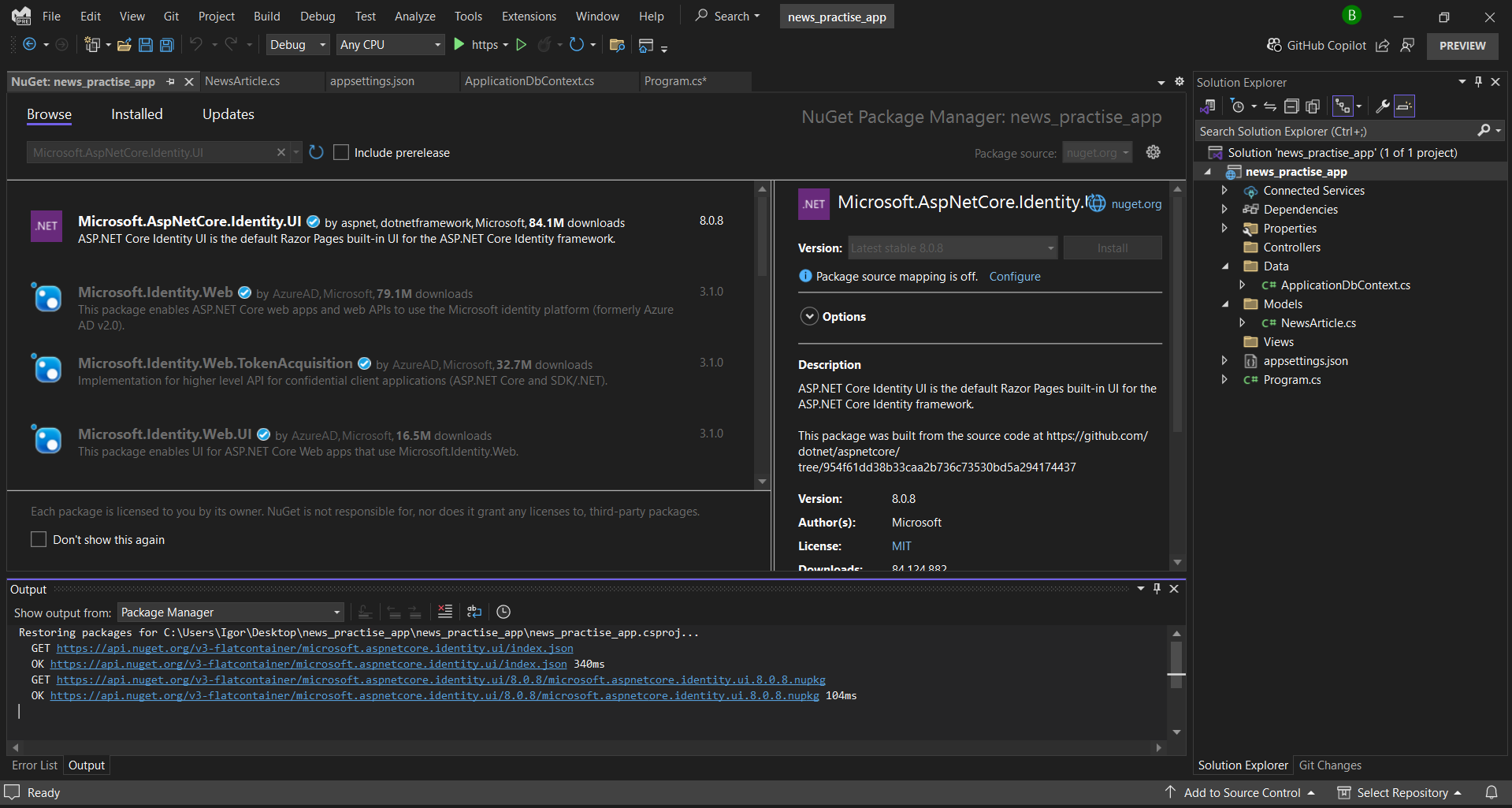
Добавление Entity Framework Core

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();

        }

    }

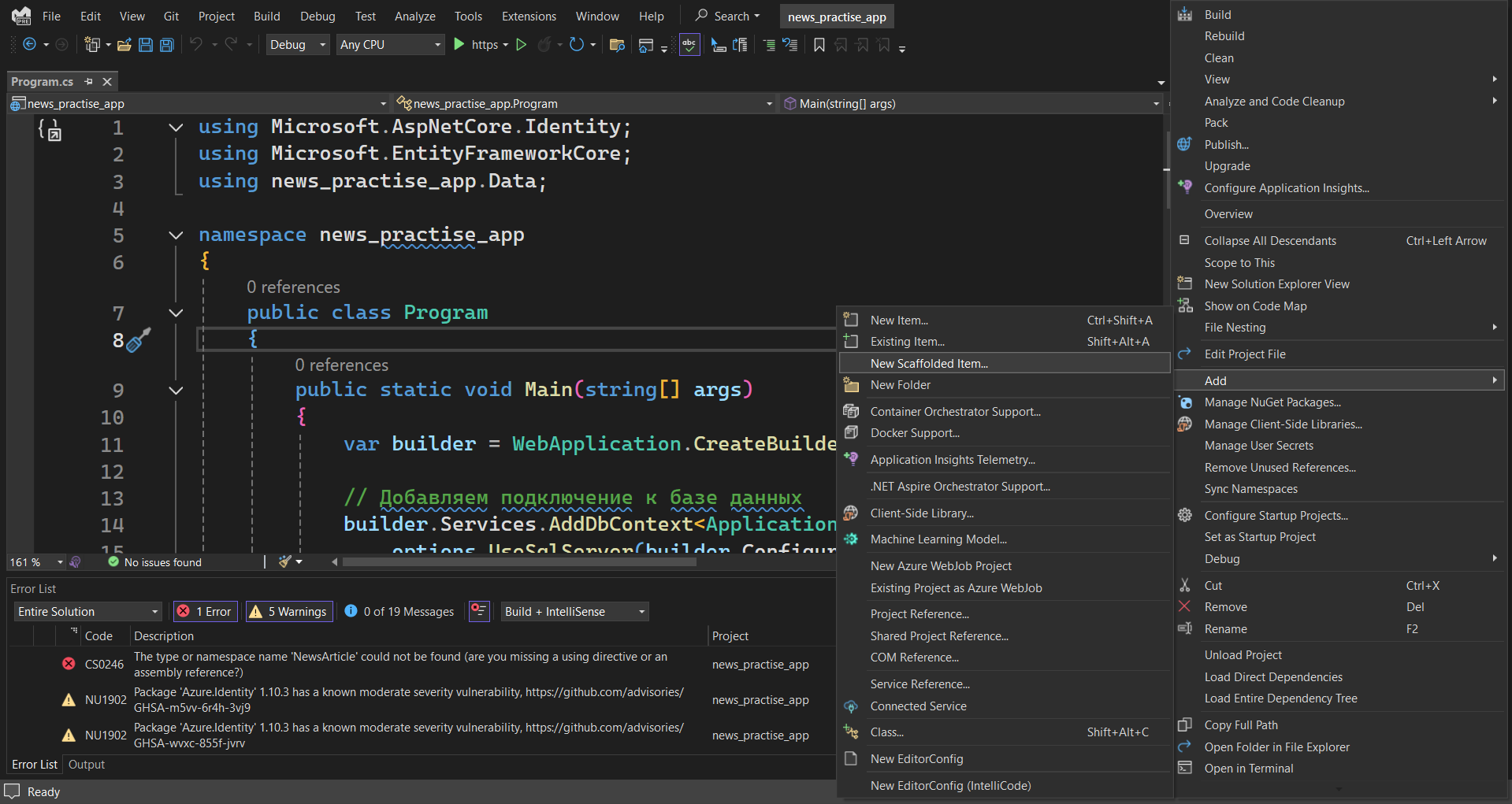
}

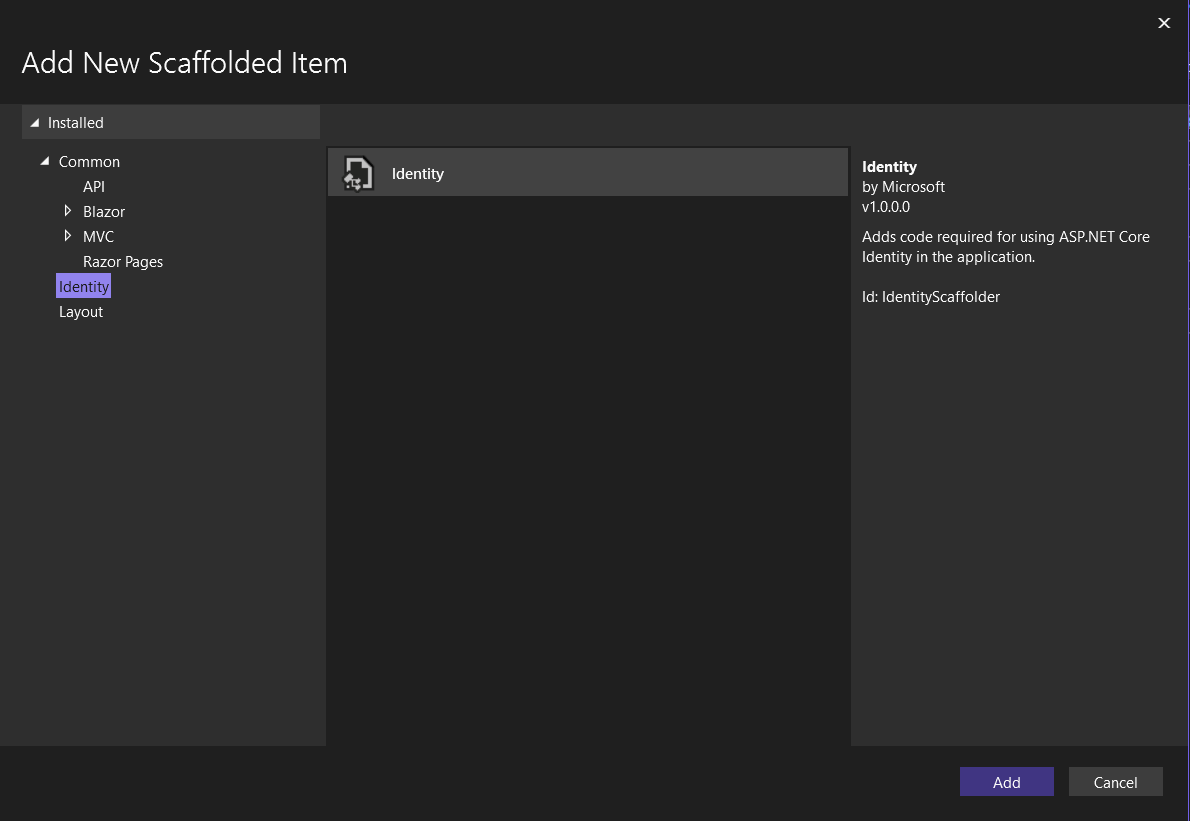
Контроллеры и представления для регистрации и авторизации

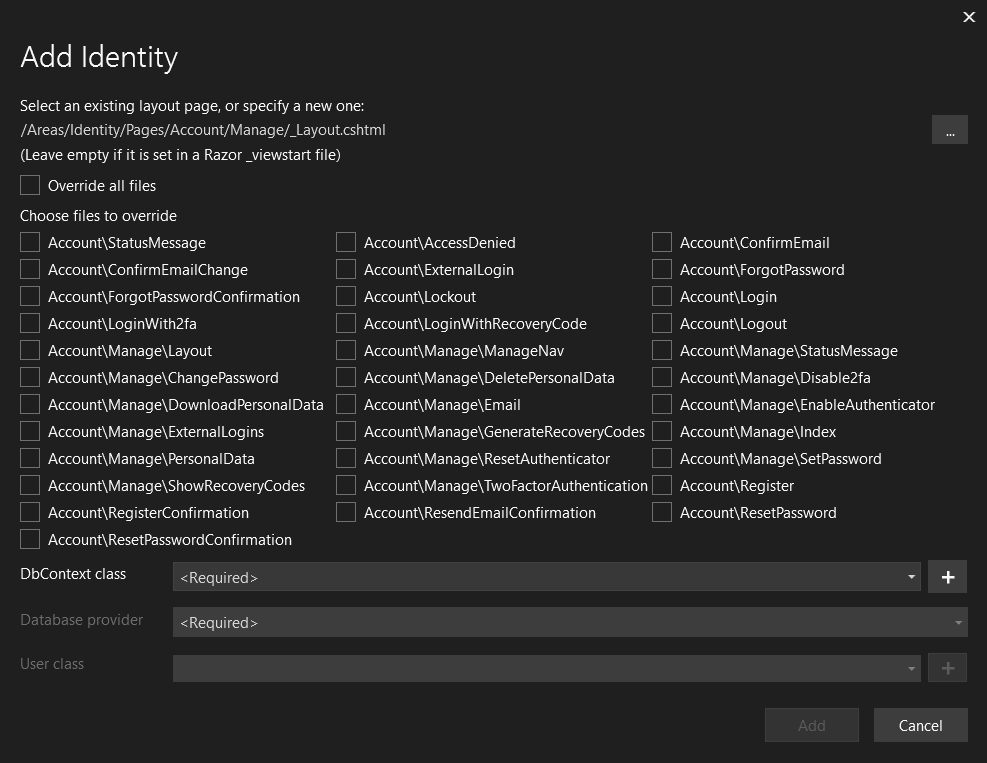
Теперь, когда Identity настроен, можно создать контроллеры и представления для регистрации и авторизации.

Использование Scaffolding для создания контроллеров и представлений

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





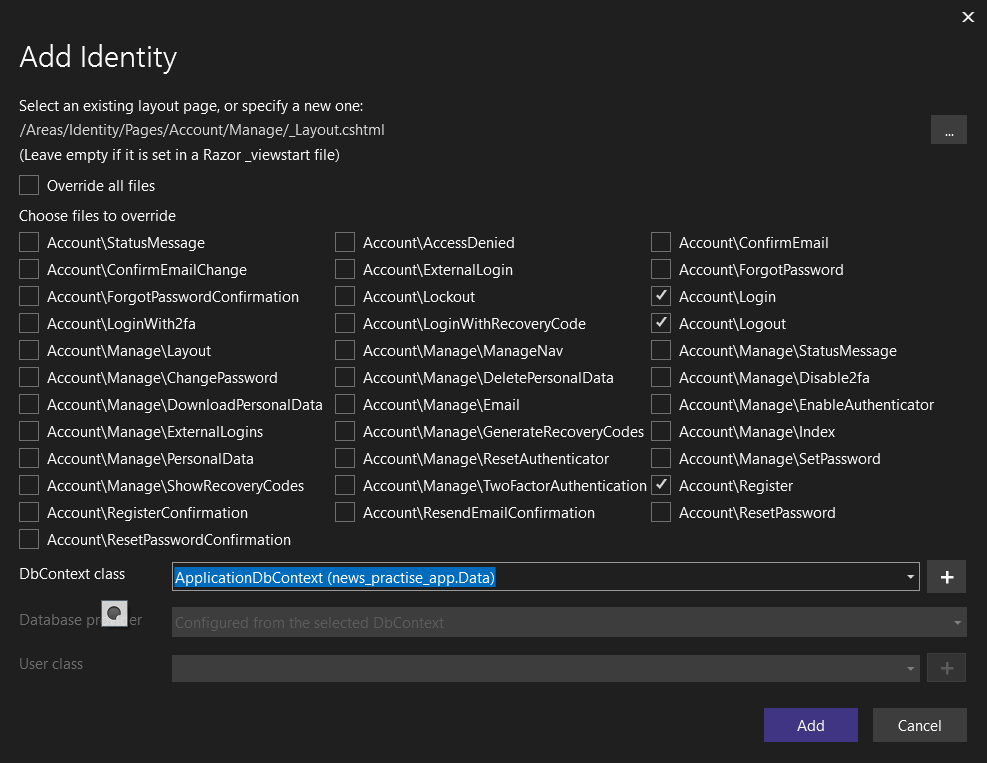


Чтобы минимизировать количество сгенерированных классов Identity для сайта новостей, можно выбрать только необходимые компоненты для базовой функциональности аутентификации и авторизации. Это позволит интегрировать систему пользователей без избыточных функций.

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.

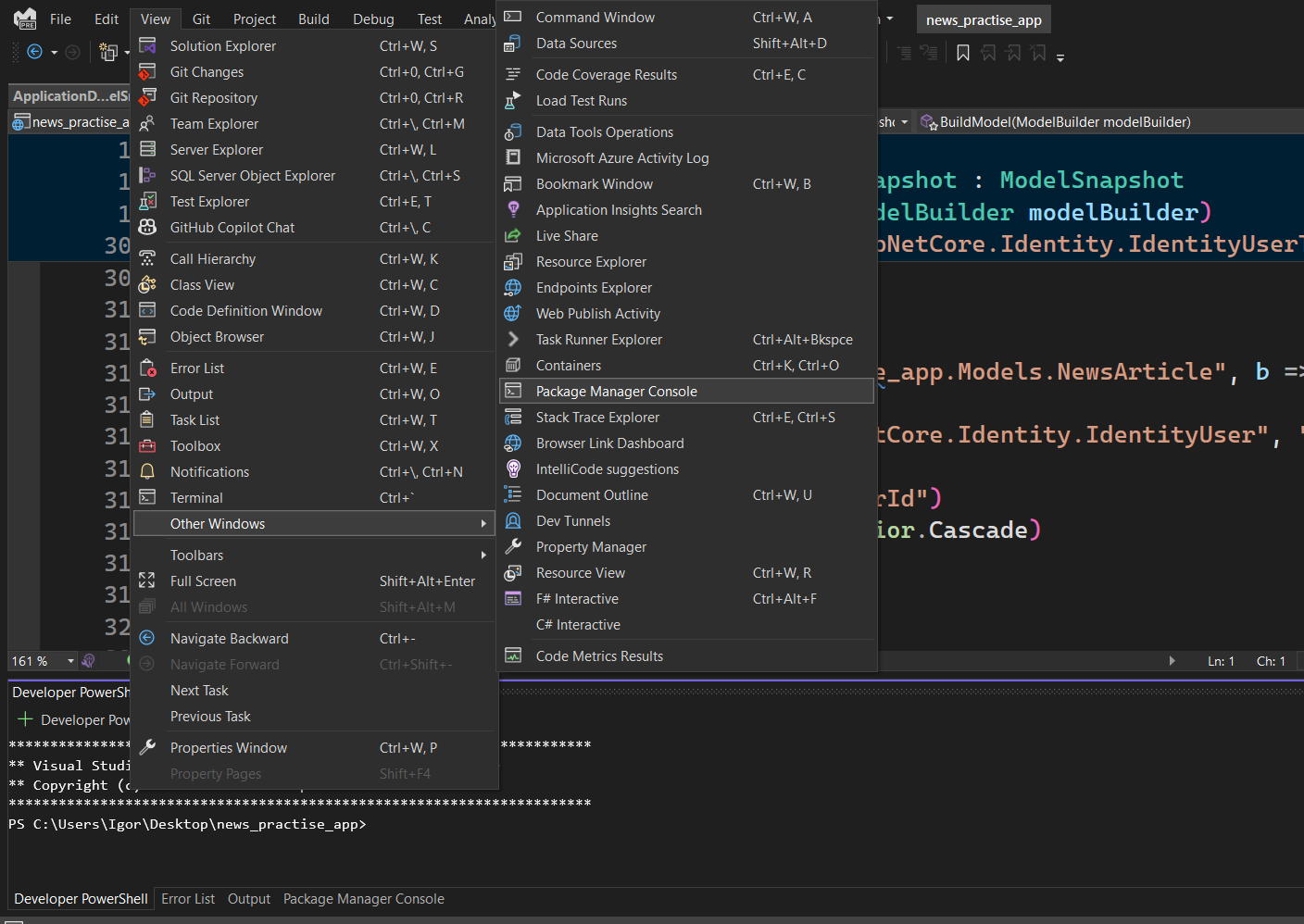


Применение миграций

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).

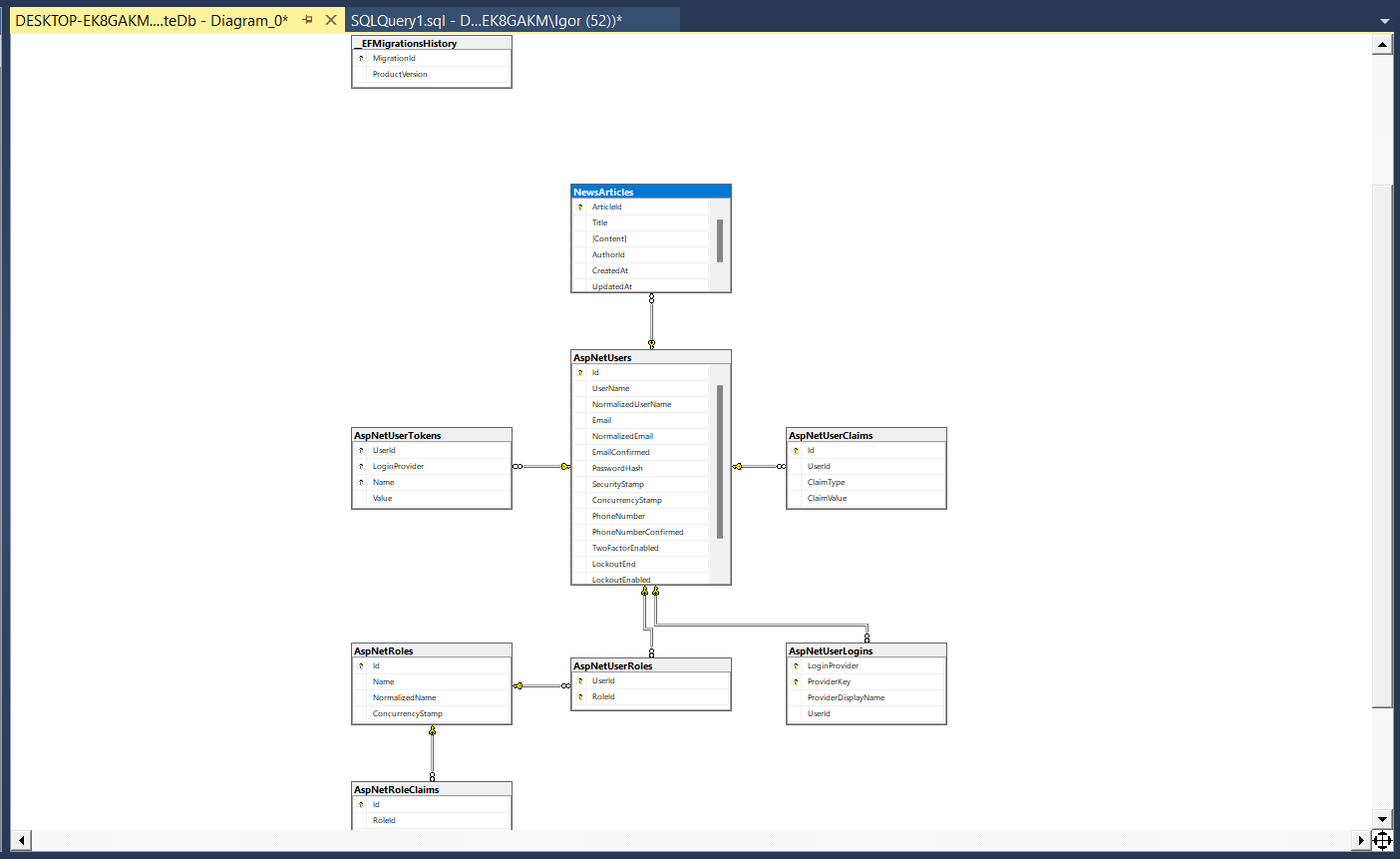
Выполните следующие команды для создания и применения миграций:



Add-Migration InitialCreate

Update-Database

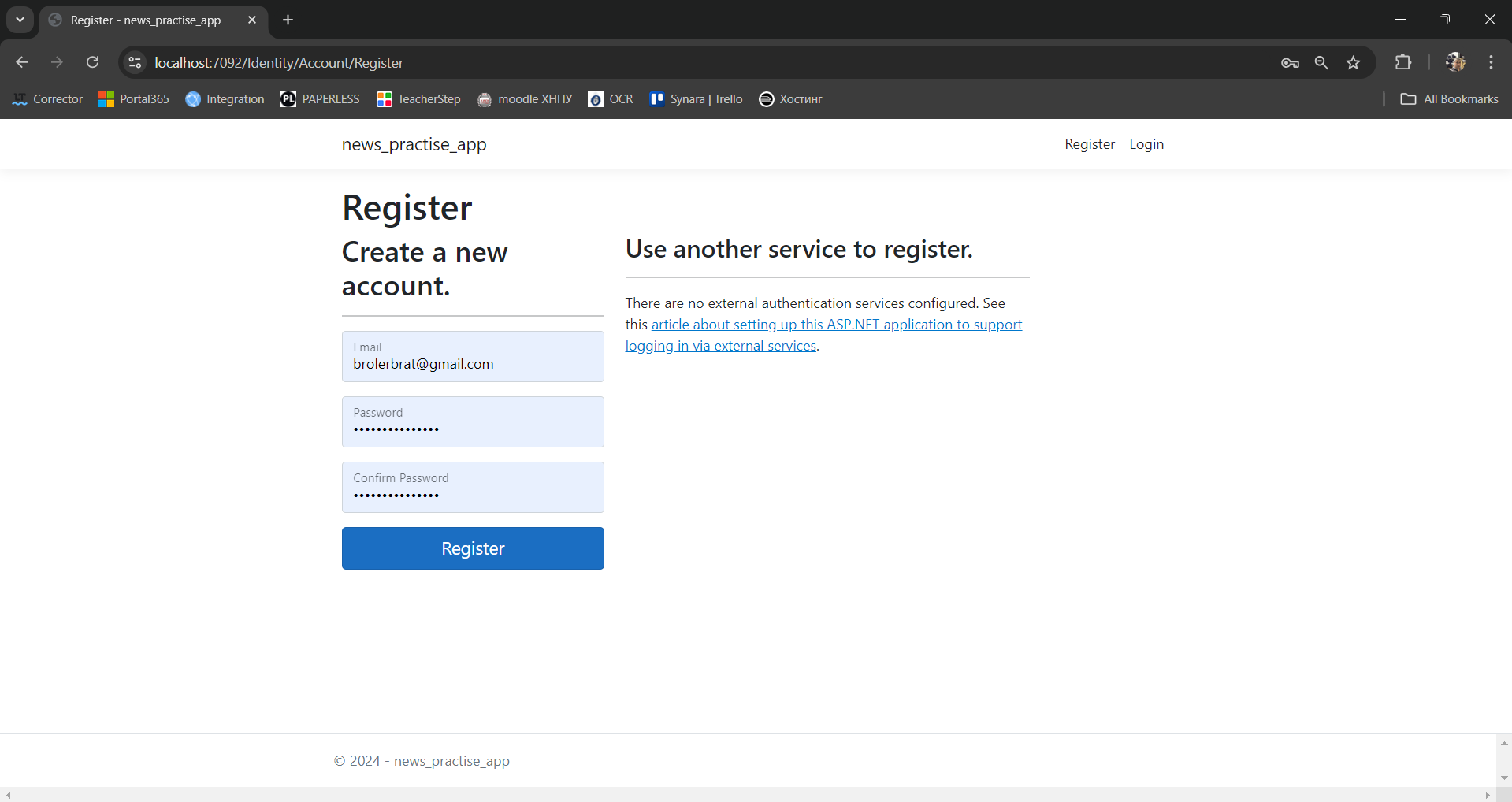
Это создаст таблицы для Identity (пользователей, ролей и их привязки).

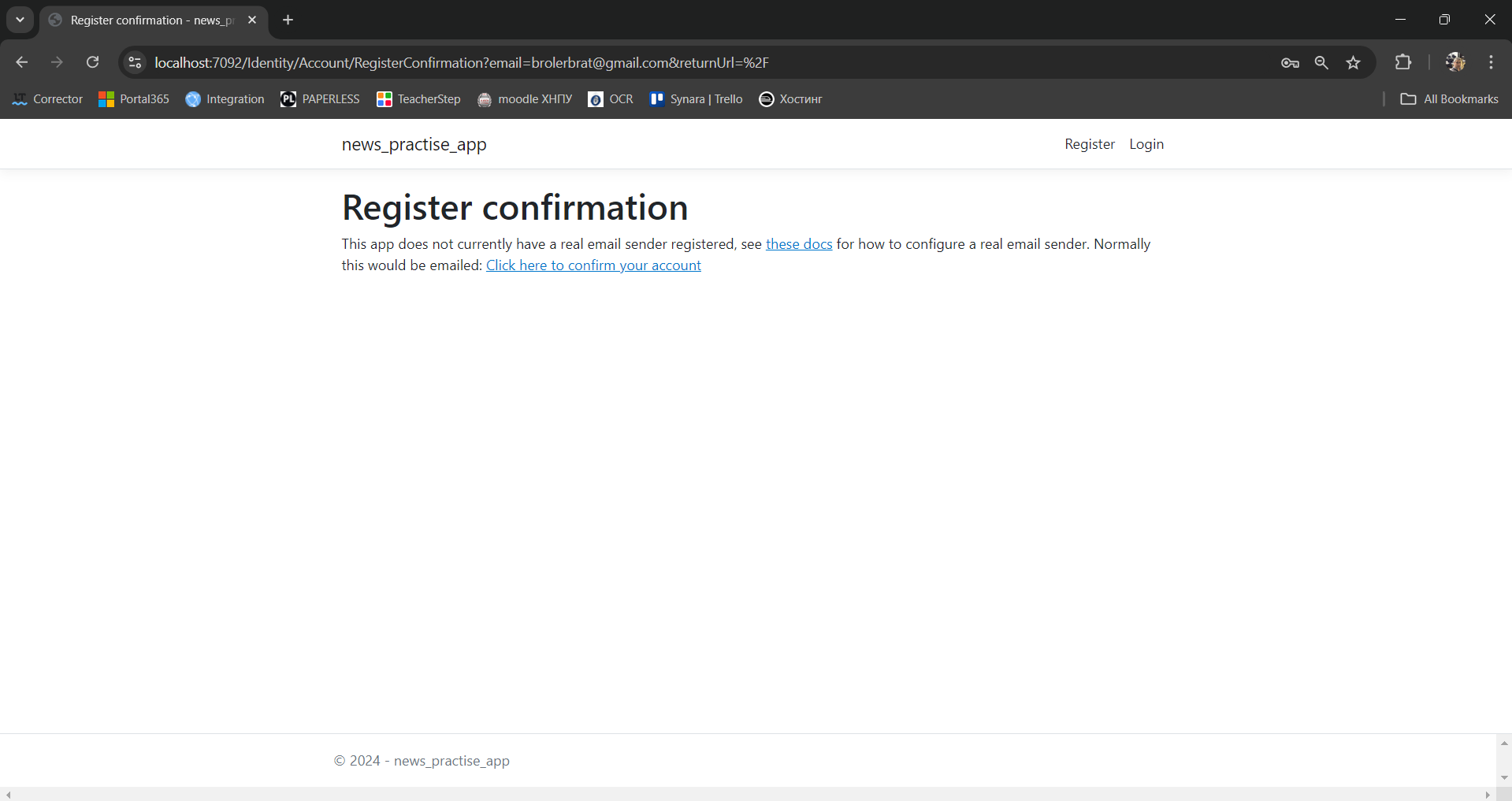


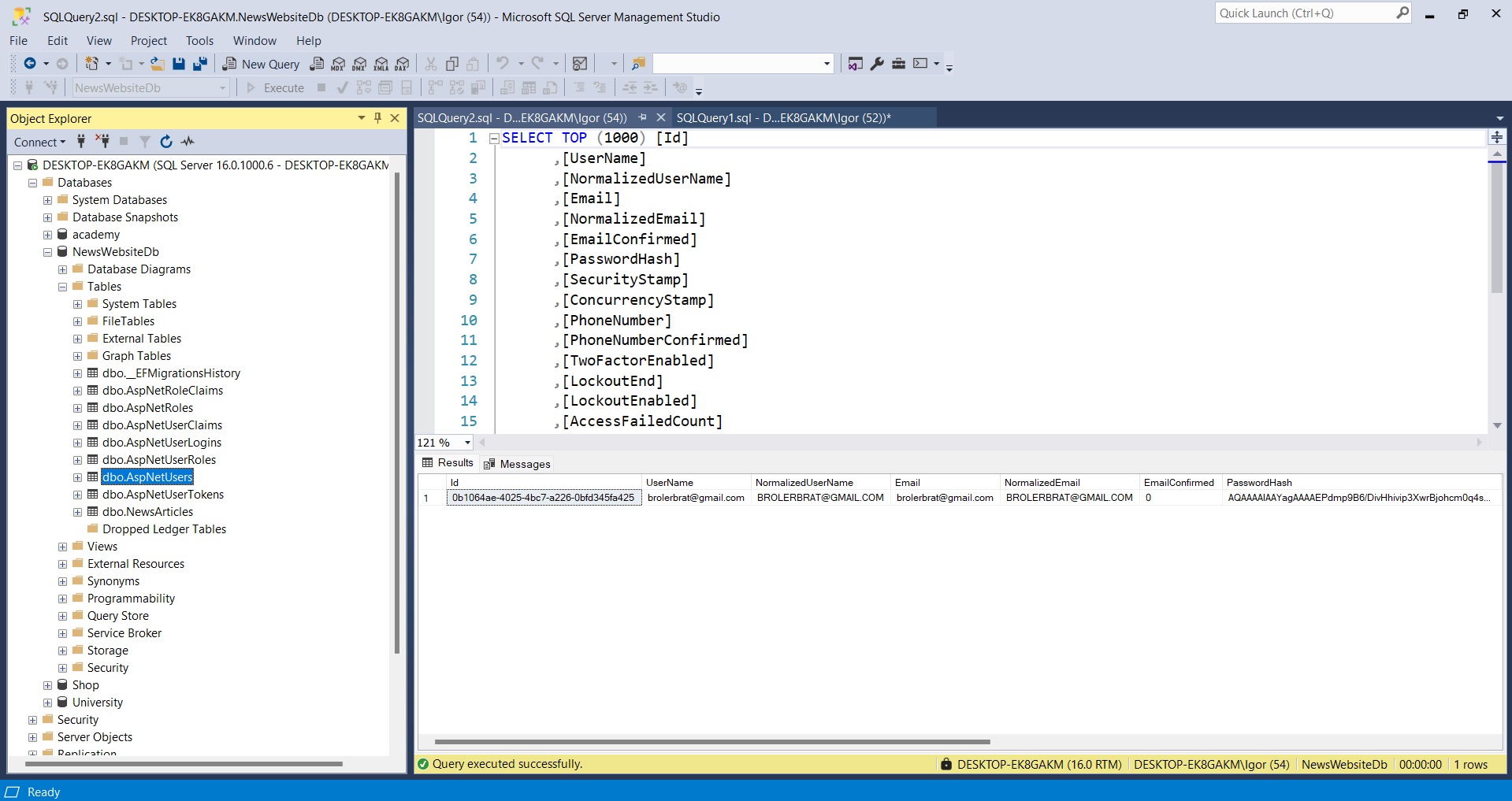
Проверка генерации и работы Identity

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





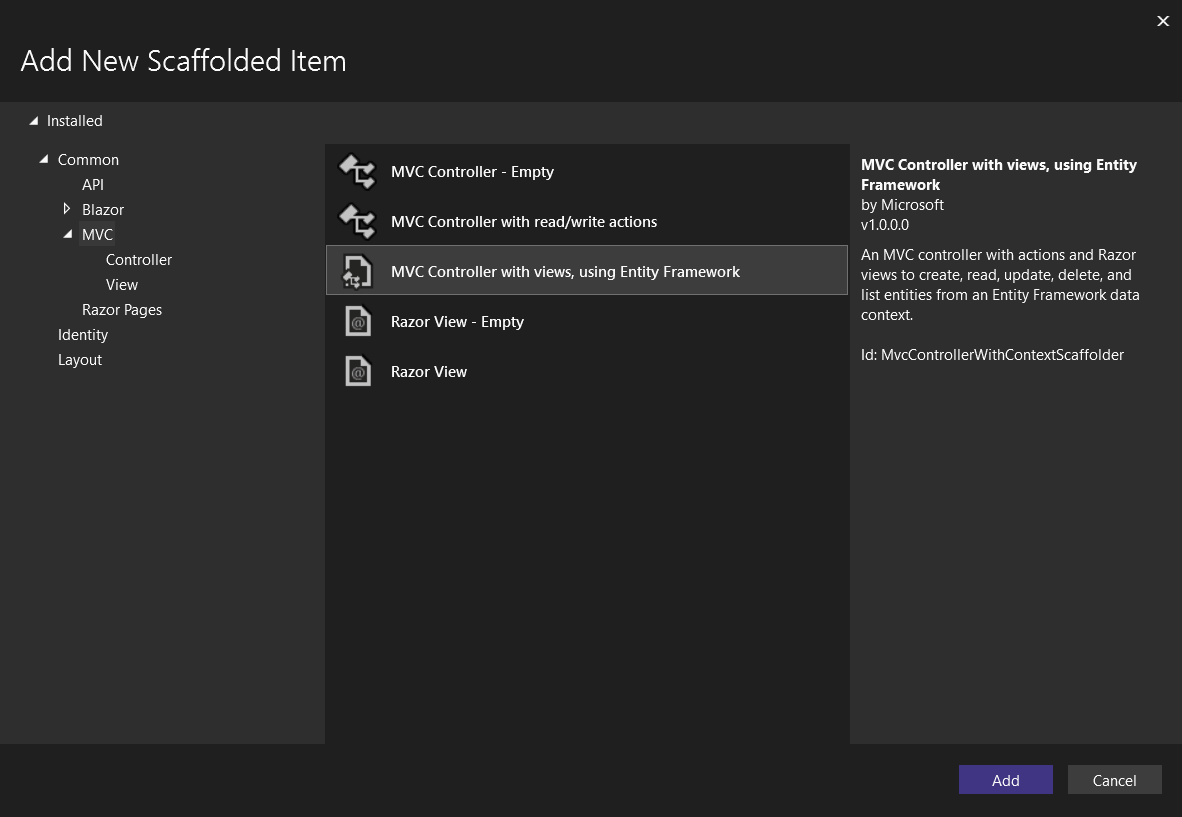


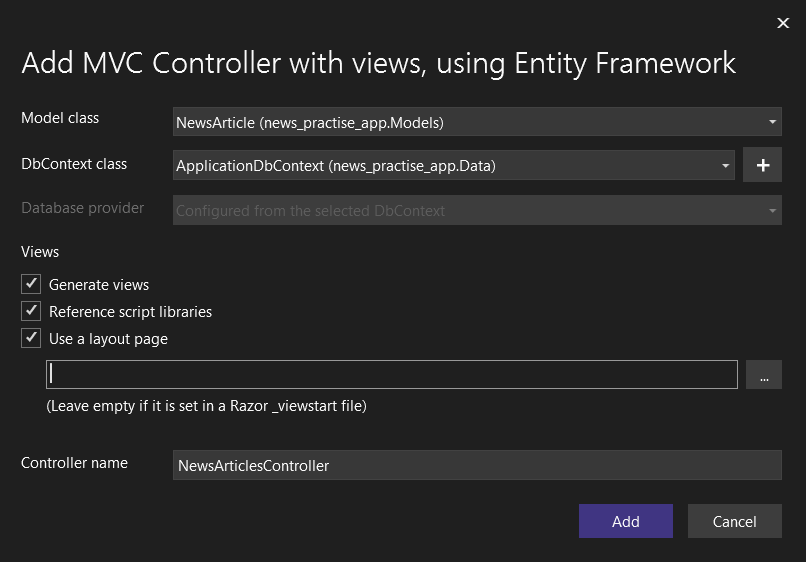
Добавление и настройка контроллеров и представлений:

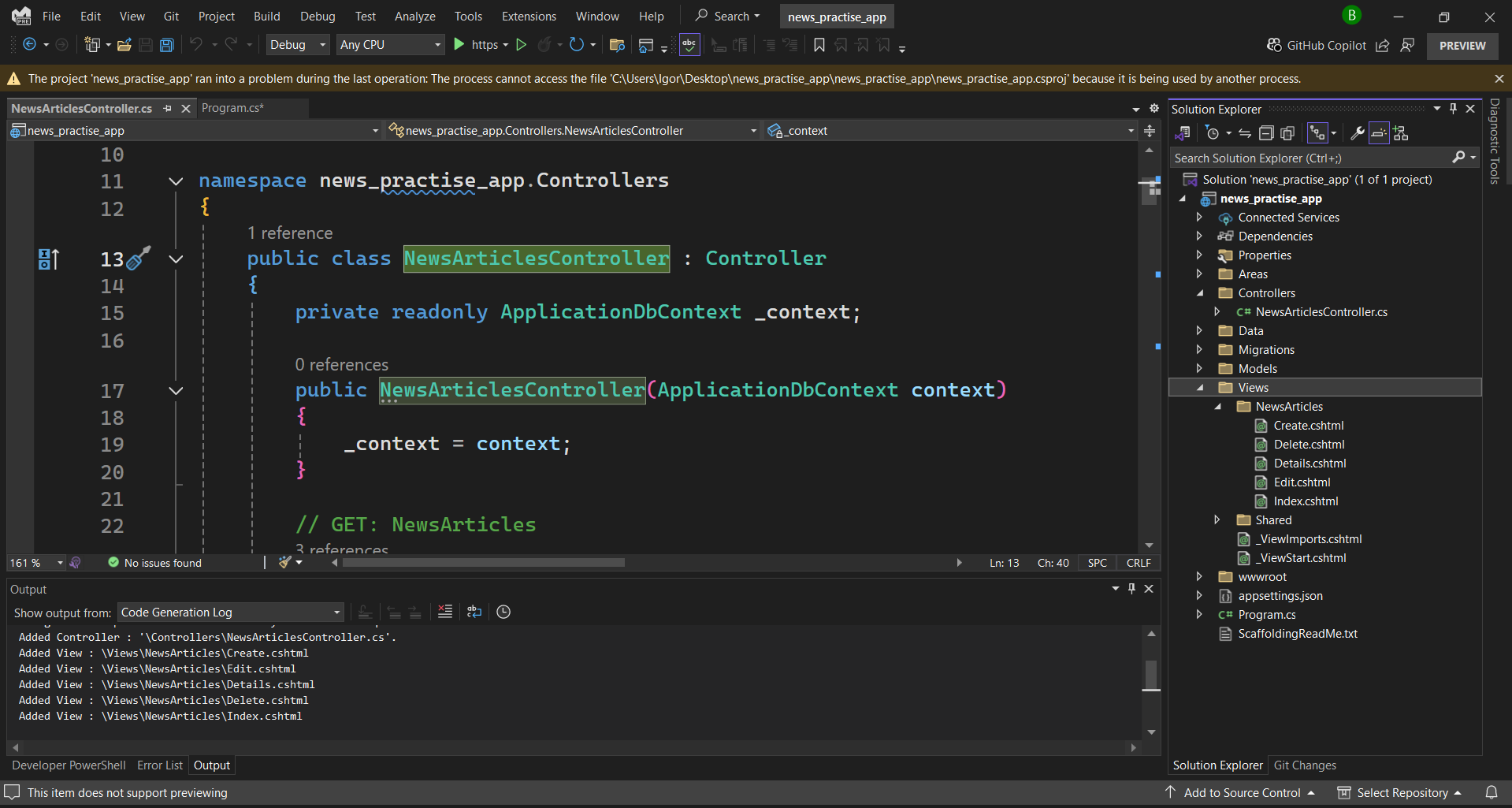
Контроллер, созданный с использованием Scaffolding, автоматически создаст представления для операций CRUD. Эти представления будут находиться в папке Views/News.

Вот краткое описание представлений, которые будут созданы:

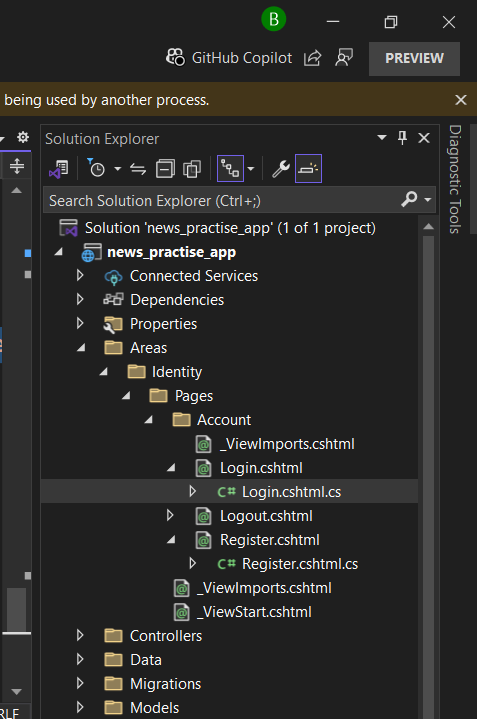
* Index.cshtml: список новостей.
* Details.cshtml: подробности новости.
* Create.cshtml: форма для создания новости.
* Edit.cshtml: форма для редактирования новости.
* Delete.cshtml: подтверждение удаления новости.







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();

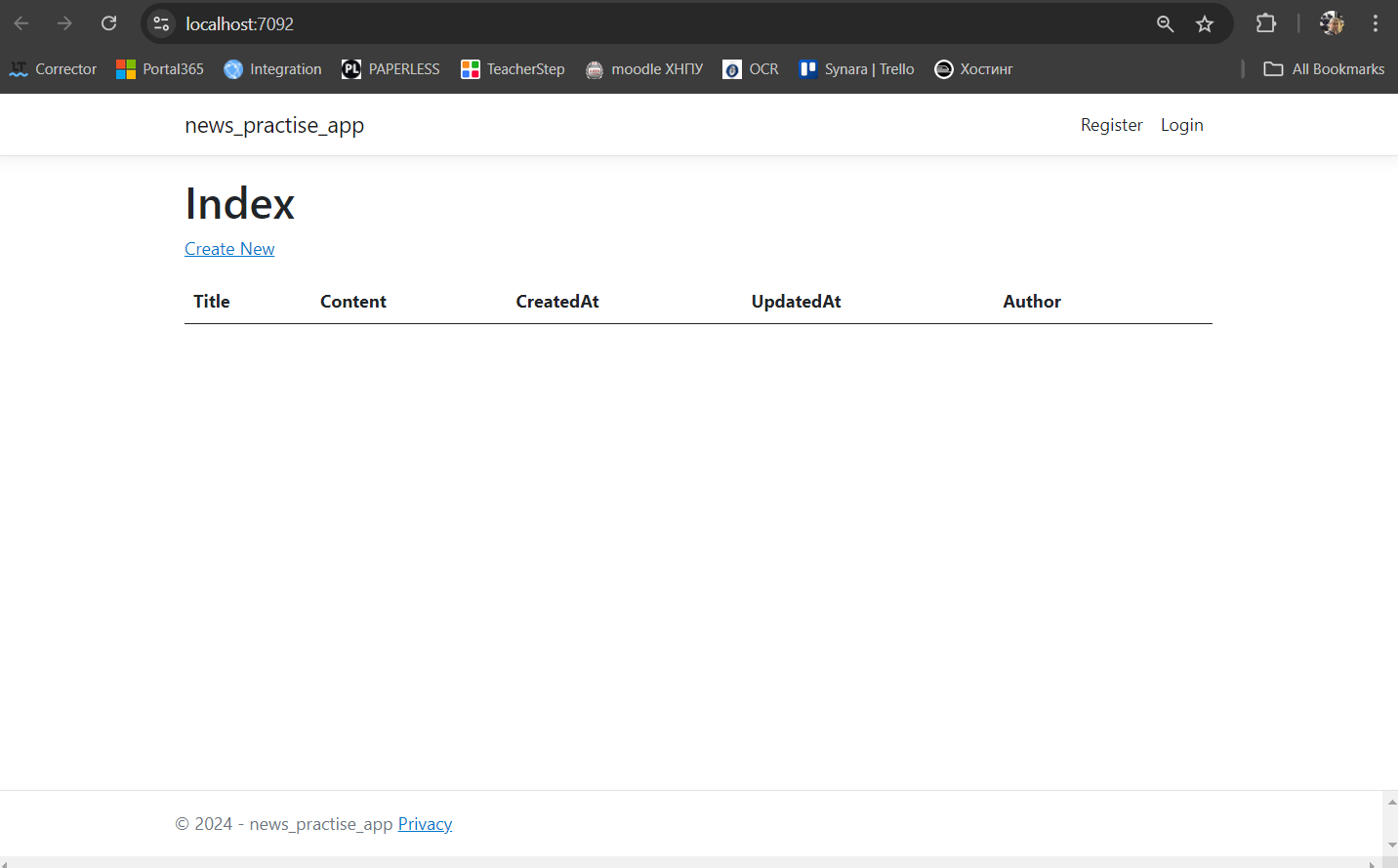
            app.Run();

        }

    }

}

Start:



Логгирование

ASP.NET Core имеет встроенную поддержку логгирования, что позволяет применять логгирование с минимальными вкраплениями кода в функционал приложения.

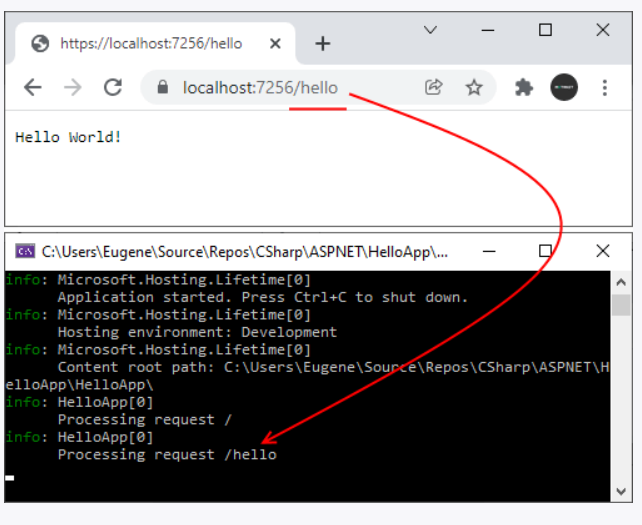
Для логгирования данных нам необходим объект **ILogger<T>**. По умолчанию среда ASP NET Core через механизм внедрения зависимостей уже предоставляет нам такой объект. Его можно получить как и любую другую зависимость в приложении. Также этот объект можно получить через свойство **Logger** объекта WebApplication.

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:

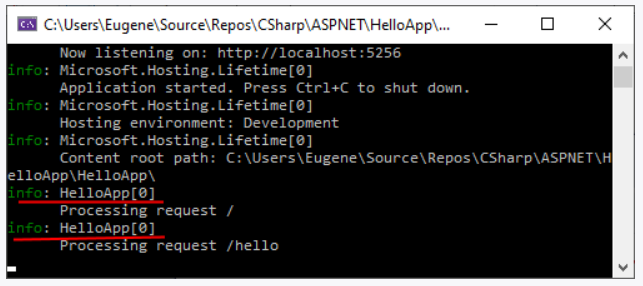


**Категория логгера**

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.

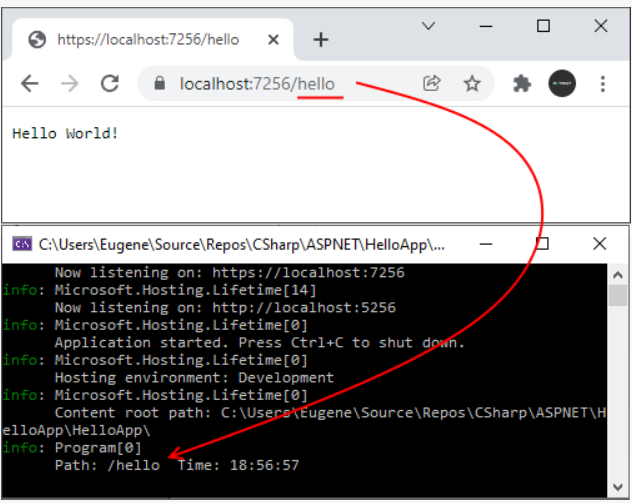
**Получение логгера через внедрение зависимостей**

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

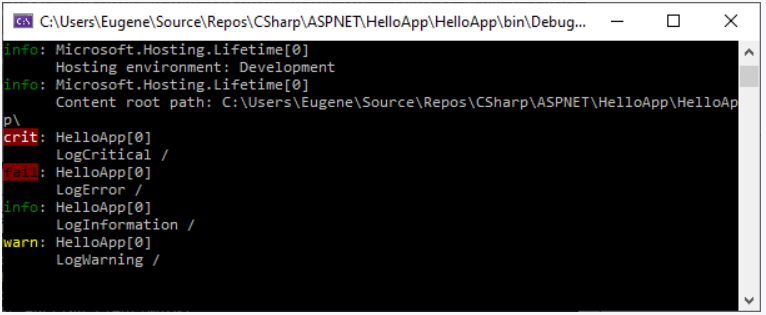
Также для логгирования определен общий метод **Log()**, который позволяет определить уровень логгера через один из параметров:

|  |  |
| --- | --- |
| 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.

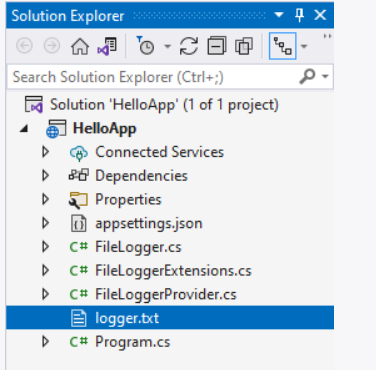
Теперь используем провайдер в файле **Program.cs**:

|  |  |
| --- | --- |
| 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.

