Projektowanie aplikacji ASP.NET

Wykład 14/15
Razor pages, Playwright

Wiktor Zychla 2024/2025

Spis treści

1 Razor Pages

Technologia Razor Pages w zamierzeniu zasypuje wyrwę jaką w .NET Core wprowadza brak wsparcia dla Web Forms – czyli brak technologii w której routowanie obsługuje nie „kontroler" ale „strona".

W teorii, Razor Pages nadaje się więc być może do scenariuszy migracji spadkowego kodu Web Forms. W praktyce – Razor Pages i tak nie obsługuje zaawansowanych formantów Web Forms takich jak GridView czy ListView.

Aplikacja Razor Pages składa się więc ze stron (deklaratywnych) oraz ich modeli (imperatywnych). Strona zawiera widok. Model zawiera dane i logikę.

Przenosząc te pojęcia do intuicji z MVC:

- **Strona** w Razor Pages to widok z MVC. Obsługiwany jest Razor taki jak w MVC (wolno używać strony layoutowej, obsługiwane są metody rozszerzające @Html…. Itd.)
- **Model** w Razor Pages to równocześnie kontroler i model z MVC. Kontroler – bo zawiera metody OnGet/OnGetAsync/OnPost/OnPostAsync itd. Model – bo zawiera dane przekazywane do widoku i bindowane przy POST formularzy. Wstrzykiwanie usług jest obsługiwane przez parametry konstruktora (jak w przypadku kontrolerów MVC).

Z technologią zapoznamy się budując prostą aplikację ze stroną logowania i stroną główną.

1.1 Program.cs

Rozruch aplikacji zawiera rejestrację stron i middleware stron. To na co należy zwrócić uwagę to konwencja wskazywania które strony wymagają autoryzacji (**AuthorizePage**).

```
using Microsoft.AspNetCore.Authentication.Cookies;

var builder = WebApplication.CreateBuilder(args);

builder.Services
    .AddAuthentication( CookieAuthenticationDefaults.AuthenticationScheme )
    .AddCookie( CookieAuthenticationDefaults.AuthenticationScheme, options
=>
    {
        options.LoginPath = "/Logon";
        options.SlidingExpiration = true;
    } );

builder.Services.AddAuthorization();

builder.Services.AddRazorPages( options =>
{
    options.Conventions.AuthorizePage( "/Index" );
} );

var app = builder.Build();

app.UseRouting();

app.UseAuthentication();
app.UseAuthorization();
```

```
app.MapRazorPages();

app.Run();
```

1.2 /Logon

Middleware autentykacji (znane już nam z wykładu o autentykacji) odpowiada za przekierowanie nieautoryzowanego żądania do /Logon, z adresem powrotu.

Model poniżej. Proszę zwrócić uwagę na wskazanie tych właściwości które podlegają bindowaniu (**BindProperty**) oraz obsługę logiki (**OnPostAsync**).

```
public class LogonModel : PageModel
{
    [Required( ErrorMessage = "Pole jest wymagane" )]
    [BindProperty]
    public string Username { get; set; }

    [Required( ErrorMessage = "Pole jest wymagane")]
    [BindProperty]
    public string Password { get; set; }

    [ViewData]
    public string Message { get; set; }

    public void OnGet()
    {
    }

    public async Task<IActionResult> OnPostAsync()
    {
        if ( this.ModelState.IsValid )
        {
            if ( !string.IsNullOrEmpty( Username ) &&
                 !string.IsNullOrEmpty( Password ) &&
            Username == Password
                )
            {
                List<Claim> claims = new List<Claim>
                {
                    new Claim(ClaimTypes.Name, Username)
                };

                // create identity
                ClaimsIdentity identity   = new ClaimsIdentity(claims, Cook
ieAuthenticationDefaults.AuthenticationScheme);
                ClaimsPrincipal principal = new ClaimsPrincipal(identity);

                await this.HttpContext.SignInAsync( CookieAuthenticationDef
aults.AuthenticationScheme, principal );

                return Redirect( "/" );
            }
            else
            {
```

```
                Message = "Zła nazwa użytkownika lub hasło";
            }

            return Page();
        }

        return Page();
    }
}
```

Widok:

```
@page
@model WebApplication1.Pages.LogonModel
@{
}

<form method="post">

    <div>
        <label>Username:</label>
        @Html.TextBoxFor( m => m.Username )
        @Html.ValidationMessageFor( m => m.Username )
    </div>
    <div>
        <label>Password:</label>
        <input asp-for="Password" />
        <span asp-validation-for="Password"></span>
    </div>
    <div>
        <button>Logon</button>
    </div>
    @if ( this.ViewData["Message"] != null )
    {
        <div>@this.ViewData["Message"]</div>
    }

</form>
```

1.3 /Index

W Razor Pages, /Index to domyślna strona, zwracana do przeglądarki przy pustym pasku adresowym. W naszej aplikacji

```
public class IndexModel : PageModel
{
    public void OnGet()
    {
    }

    public async Task<IActionResult> OnPostAsync()
    {
        await this.HttpContext.SignOutAsync();
```

```
        return Page();
    }
}
```

oraz

```
@page
@model WebApplication1.Pages.IndexModel
@{
}

Witaj, @this.User.Identity.Name

<div>
    <form method="post">
        <button>Wyloguj</button>
    </form>
</div>
```

2 Testowanie interfejsu użytkownika

Spośród różnych możliwości testowania interfejsu użytkownika warto zwrócić uwagę na framework Playwright, łączący elementy do tej pory występujące fragmentarycznie we wcześniejszych podejściach, m.in.:

- Wsparcie różnych przeglądarek
- Tryb headless
- Generator kodu

Aby utworzyć pusty projekt, należy w konsoli wydać polecenie

```
npm init playwright@latest
```

Testy znajdują się w folderze /tests. W zależności od wyboru, tworzy się je w jednym ze wspieranych języków.

Warto przestudiować ważniejsze fragmenty dokumentacji, m.in.:

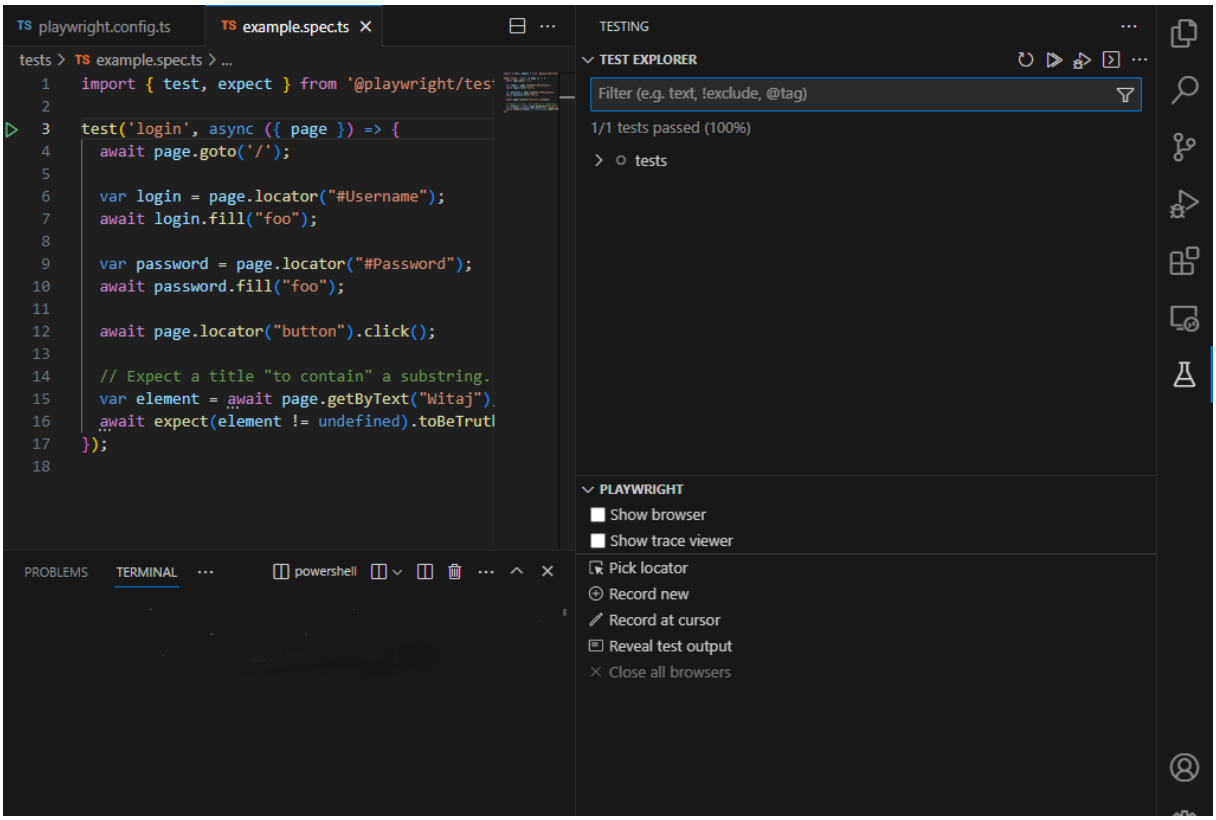
- Akcje
- Asercje

Przykładowy test do aplikacji z poprzedniego rozdziału poniżej. Przed uruchomieniem testu należy w **playwright.config.ts** zmienić **baseURL** na url odpowiadający ścieżce aplikacji.

```
import { test, expect } from '@playwright/test';

test('login', async ({ page }) => {
  await page.goto('/');

  var login = page.locator("#Username");
  await login.fill("foo");

  var password = page.locator("#Password");
  await password.fill("foo");

  await page.locator("button").click();

  // Expect a title "to contain" a substring.
  var element = await page.getByText("Witaj");
  await expect(element != undefined).toBeTruthy();
});
```

Tworzenie własnych testów możliwe jest w trybie pisania kodu, lub przez użycie generatora.

Generator dostępny jest w VS Code na zakładce testów:


playwright.config.ts
example.spec.ts
tests > example.spec.ts > ...
import { test, expect } from '@playwright/tes
test('login', async ({ page }) => {
await page.goto('/');
var login = page.locator("#Username");
await login.fill("foo");
var password = page.locator("#Password");
await password.fill("foo");
await page.locator("button").click();
// Expect a title "to contain" a substring.
var element = await page.getByText("Witaj")
await expect(element != undefined).toBeTrut
});
PROBLEMS
TERMINAL
powershell
TESTING
TEST EXPLORER
Filter (e.g. text, !exclude, @tag)
1/1 tests passed (100%)
tests
PLAYWRIGHT
Show browser
Show trace viewer
Pick locator
Record new
Record at cursor
Reveal test output
Close all browsers