Programowanie aplikacji internetowych

2025/26

Instrukcja laboratoryjna

Razor Pages

Prowadzący: Tomasz Goluch

Wersja: 1.0

1. Wstęp

Razor Pages pozwalają na łatwiejsze i bardziej wydajne kodowanie scenariuszy skoncentrowanych niż przy użyciu kontrolerów i widoków. Routing został uproszczony teraz każdy plik Razor Pages znaleziony w katalogu `Pages` definiuje powiązany z jego nazwą endpoint.

2. Instalacja

Narzędzia i środowiska dla Razor Pages są dostępne w instalatorze Visual Studio. Wystarczy pamiętać o włączeniu pakietu „Opracowywanie zawartości dla platformy ASP.NET i sieci Web". Na laboratorium będzie omówiona aplikacja internetowa ASP.NET Core (ASP.NET Core Web App) na platformie .NET 8.0 w środowisku Visual Studio 2022 – to oprogramowanie jest dostępne na komputerach w laboratorium. Dodatkowe informacje na temat omawianej aplikacji można znaleźć pod adresem: https://learn.microsoft.com/en-us/aspnet/core/razor-pages/?view=aspnetcore-7.0. Development z wykorzystaniem kontenerów Docker wymaga zainstalowania środowiska Docker Desktop on Windows.

3. Projekt

Tutorial opisujący utworzenie projektu w edytorze VS: https://learn.microsoft.com/en-us/aspnet/core/tutorials/razor-pages/razor-pages-start.

Uwierzytelnianie:

- Brak – logowanie nie będzie wspierane, identyczny dostęp dla wszystkich użytkowników;
- Pojedyncze konta – logowanie w oparciu o bazę danych przechowywującej informacje o użytkownikach wraz z ze skrótami haseł;
- Platforma tożsamości firmy Microsoft – logowanie przy użyciu swoich poświadczeń usługi Azure Active Directory, Office 365 lub lokalnej usługi Active Directory;
- Windows – preferowane dla wewnętrznej, intranetowej witryny firmowej, użytkownicy podczas logowania będą mogli używać swoich standardowych nazw i haseł z wykorzystaniem modułu IIS uwierzytelniania systemu Windows;

4. Strony Razor[1]

Aby móc korzystać ze stron Razor należy w pliku `Program.cs` dodać ich obsługę:

```
builder.Services.AddRazorPages();
```

należy to zrobić jeszcze przed zbudowaniem aplikacji. Następnie należy dodać punkty końcowe dla Razor Pages co robimy po zbudowaniu aplikacji wołając na jej obiekcie metodę:

```
app.MapRazorPages();
```

[1] https://learn.microsoft.com/en-us/aspnet/core/razor-pages/?view=aspnetcore-6.0&tabs=visual-studio.

Oczywiście obydwie metody będą domyślnie wywołane w każdym nowo utworzonym projekcie obsługującym strony Razor. To co odróżnia strony Razor od „zwykłych ASP.NET" to dyrektywa @page, od której musi rozpoczynać się strona .cshtml. Przekształca ona plik w akcję MVC, co oznacza, że obsługuje żądania bezpośrednio, bez przechodzenia przez kontroler. Oto przykład prostej strony (samotny plik <nazwa_strony>.cshtml):

```
@page

<h1>Hello, world!</h1>
<h2>The time on the server is @DateTime.Now</h2>
```

Po umieszczeniu strony w folderze Pages (domyślnym dla środowiska uruchomieniowego miejscu wyszukiwania stron Razor) powinna być od razu widoczna (w przypadku niepowodzenia sprawdzić czy wywołana została metoda app.MapRazorPages) pod url: https://<adres>:<port>/<nazwa_strony>.

W celu dodania nowej strony wraz z modelem klikamy prawym przyciskiem myszy na folderze Pages i z menu kontekstowego wybieramy: Add → RazorPage… → Razor page – empty. Wygenerowane zostaną dwa pliki:

- strona Razor zawierająca dyrektywę:
  `@model <nazwa_projektu>.Pages.<nazwa_strony>Model;`
- plik z modelem, klasą dziedziczącą po `PageModel`.

Zgodnie z konwencją plik z modelem powinien mieć taką samą nazwę jak odpowiadająca mu strona Razor z dodanym suffixem .cs. i znajdować się w tej samej przestrzeni nazw.

W pliku strony, dodajmy kod z odwołaniem do właściwości:

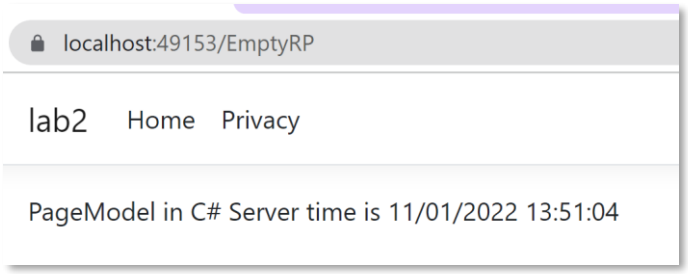
```
@{
    <p>
        @Model.Message
    </p>
}
```

A w modelu dodajmy właściwość `Message` oraz zaimplementujmy metodę `OnGet()`.

```
public string Message { get; private set; } = "PageModel in C#";

public void OnGet()
{
    Message += $" Server time is {DateTime.Now}";
}
```

Strona powinna być widoczna w przeglądarce:



Zadaniem klasy `PageModel` jest oddzielenie logiki strony od jej prezentacji. Definiuje ona kod obsługujący żądania wysyłane do strony i dane używane do renderowania strony. Pozwala to na zastosowanie testów jednostkowych dla wydzielonej logiki oraz zarządzanie zależnościami stron poprzez DI. Metody obsługi żądań nazywane są identycznie jak powiązane z nimi żądania HTTP z przedrostkiem On: obsługi: `OnGet`, `OnPost`, `OnPut`, `OnDelete`, itp. Najczęściej wykorzystywane to:

- `OnGet` – inicjalizuje stan potrzebny do wyświetla strony Razor *.cshtml;
- `OnPost` – wykorzystywana do obsługi zgłoszeń formularzy. Podstawowy przepływ wygląda następująco:
  - sprawdzenie czy wystąpiły błędy walidacji,
  - w przypadku braku błędów, zapisz dane i przekieruj.
  - w przeciwnym przypadku, ponownie pokaż stronę z komunikatami o błędach walidacji. Większość potencjalnych błędów walidacji jest wykrywana na kliencie i nigdy nie trafiają do serwera.

Większość prymitywów MVC, takich jak binding, walidacja, wyniki akcji (action results), działa tak samo ze stronami Razor jak z kontrolerami. Metody obsługi zadań podobnie jak akcje w MVC definiują dalsze działanie aplikacji poprzez zwróconą wartość (najczęściej `PageResult`) i zazwyczaj jest to wynik wołania jednej z poniższych metod:

- `Page` – (domyślny typ, zwraca PageResult) zachowanie podobne do akcji zwracających View() w kontrolerach.
- `RedirectToPage` – zachowanie podobne do `RedirectToAction` lub `RedirectToRoute` (używany w kontrolerach i widokach). Parametr jest łączony ze ścieżką bieżącej strony w celu wyznaczenia nazwy strony docelowej. Aby przekierować do strony w innym obszarze[2], należy podać ten obszar jako drugi parametr: `RedirectToPage("/Index", new { area = "Services" });`

Jeśli Metoda obsługi jedynie renderuje stronę to, zwraca `void`.

Atrybut `[BindProperty]` można zastosować do właściwości publicznej kontrolera lub klasy PageModel, aby spowodować jej powiązanie z modelem. Eliminuje to potrzebę pisania kodu w celu konwersji danych HTTP na typ modelu. Wiązanie redukuje kod, używając tej samej właściwości do renderowania pól formularza: `<input asp-for="<powiązan_właściwość>.<składowa>" class="form-control" />` i akceptowania danych wejściowych.

[2] https://learn.microsoft.com/en-us/aspnet/core/mvc/controllers/areas

5. Scaffolding[3]

Zadaniem scaffolding'u jest automatyczne generowanie części aplikacji. VS realizuje w tym zakresie automatyzację tworzenia stron z podstawową funkcjonalnością CRUD (create, read, update i delete). W tym celu należy w folderze Pages utworzyć folder, który najrozsądniej będzie nazwać identycznie jak nazwę typu encji. Klikamy prawym przyciskiem myszy na utworzonym folderze i wybieramy z menu kontekstowego Add → New Scaffolded Item. W oknie dialogowym Add New Scaffolded Item wybierz Razor Pages using Entity Framework (CRUD) → Add. Wypełnij okno dialogowe Add Razor Pages using Entity Framework (CRUD). Z listy rozwijanej *Model class:* wybierz klasę repezentująca typ encji, a z listy rozwijalnej *Data context class:* odpowiedni kontekt danych. Jeśli odpowiedni kontekst danych nie istnieje można go łatwo wygenerować klikając przycisk ze znakiem +. Ostatecznie należy zaakceptować klikając przycisk Add. W utworzonym przez nas wcześniej folderze zostaną wygenerowane strony CRUD dla danego typu encji modelu, a w przypadku wygenerowania kontekstu w pliku Program.cs dodany zostanie kod odpowiedzialny za jego rejestrację w postaci serwisu. Pozwoli to na wstrzykiwanie kontekstów danych do komponentów które je wykorzystują (np. strony RP) poprzez konstruktor za pomocą mechanizmu DI.

```
builder.Services.AddDbContext<NAZWA_KONTEKSTU>(
            dbContextOptions => dbContextOptions

.UseSqlServer(builder.Configuration.GetConnectionString("NAZWA_KONTEKS
TU")
            ?? throw new InvalidOperationException("Connection string
'NAZWA_KONTEKSTU' not found.")
        ));
```

Wygenerowane strony CRUD dziedziczą po `PageModel` i zgodnie z konwencją powinny posiadać nazwę: `<NAZWA_STRONY>Model`, np. `CreateModel`. Zawierają one następujące dyrektywy:

- `@Html.DisplayNameFor` – wyświetla nazwę właściwości modelu lub nazwę podaną w atrybucie Display tej właściwości.
- `@Html.DisplayFor` – wyświetla zawartość właściwości modelu.

Jako parametr do obydwu powyższych dyrektyw przekazywane jest wyrażenie lambda w postaci (`model => model.<właściwość>`). W przypadku `DisplayNameFor` jest ono jedynie sprawdzane więc nie spowoduje naruszenia zasad dostępu w przypadku pojawienia się wartości null, tak jak to będzie miało miejsce w przypadku `DisplayFor`.

- `@RenderBody` – domyślnie wołana na stronie layout'u Pages/Shared/_Layout.cshtml. Reprezentuje placeholder w którym pojawiają się wszystkie widoki specyficzne dla strony, opakowane w ten layout. Przykładowo po wybraniu linku Privacy w miejscu `@RenderBody` zostanie wyświetlony widok Pages/Privacy.cshtml.
- `@*Komentarz Razor*@` – w przeciwieństwie do komentarzy HTML `<!-- -->` komentarze Razor nie są wysyłane do klienta.

[3] https://learn.microsoft.com/en-us/aspnet/core/tutorials/razor-pages/model?view=aspnetcore-7.0&tabs=visual-studio#scaffold-the-movie-model, https://learn.microsoft.com/en-us/aspnet/core/tutorials/razor-pages/page

Jeśli chcemy aby strony Create i Edit zawierały listy rozwijalne wypełnione dostępnymi pozycjami encji zależnych należy poprawnie zdefiniować relacje w modelu danych. W tym celu w klasie reprezentującej encję nadrzędną należy do właściwości typu klasy podrzędnej dodać jeszcze właściwość typu int o nazwie rozbudowanej o suffix ID reprezentującą klucz obcy.

```
namespace Model
{
    public class Product
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public int ProducerID { get; set; }
        public Producer Producer { get; set; }
    }
}
```

Po wygenerowaniu strona Create powinna zawierać w widoku kontrolkę `select`:

```
<div class="form-group">
    <label asp-for="Product.ProducerID" class="control-label"></label>
    <select asp-for="Product.ProducerID" class ="form-control" asp-
items="ViewBag.ProducerID"></select>
</div>
```

A metoda w kodzie behind powinna wypełnić słownik `ViewData` odpowiednimi danymi.

```
public IActionResult OnGet()
{
ViewData["ProducerID"] = new SelectList(_context.Producer, "Id", "Name");
    return Page();
}
```

W przypadku relacji wiele do wielu musimy samodzielnie zaimplementować funkcjonalność stron Create i Edit. W tym celu można wykorzystać tag pomocniczy `<input type="checkbox"/>` omówiony w dalszej części tego dokumentu.

Na stronie przedmiotu dołączono kod prostej aplikacji realizującej operacje dodawania i edycji encji typu wiele do wielu z wykorzystaniem bazy danych w pamięci. Można też przełączyć się na fizyczną bazę danych komentując komendę aby zobaczyć relacje utworzone w bazie danych. Wymagany będzie wtedy update bazy danych.

Tagi pomocnicze[4] w plikach stron Razor umożliwiają tworzenie i renderowanie elementów HTML po stronie serwera. Zastępują helpery HTML – poprzednie podejściem do tworzenia znaczników po stronie serwera w widokach Razor. W większości przypadków wyglądają jak standardowy kod HTML ale wymagają mniejszej liczby znaczników w kodzie źródłowym. Zapewniane jest silne typowanie kontrolek HTML takich jak: **`<label>`** , **`<input>`** z właściwościami modelu. Najczęściej stosowane na stronach CRUD:

- `form` – tworzy formularz HTML z wygenerowaną wartością atrybutu action dla akcji kontrolera MVC lub nazwanej trasy w oparciu o wartości tagów pomocniczych Form

[4] https://learn.microsoft.com/en-us/aspnet/core/mvc/views/tag-helpers/intro?view=aspnetcore-7.0

Action[5] , takich jak: `asp-controller` i `asp-action` (nazwa kontrolera i akcji) lub `asp-route`. Ponadto generuje ukryty token weryfikacji żądania, aby zapobiec atakom CSRF.

- `label` – alternatywa dla helpera HTML: `Html.LabelFor`. Tworzy element HTML z wygenerowanymi wartościami atrybutów `caption` i `for` w oparciu o wartość dostarczoną z `asp-for`. Na podstawie atrybutu `Display` w modelu dostarcza opis etykiety, który może się zmieniać w czasie.
- `input` – wiąże element HTML `<input>` z wyrażeniem modelu w widoku Razor. Generuje wartości atrybutów `id` i `name` w oparciu o wartość dostarczoną z `asp-for`. Generuje atrybuty walidacji HTML5 z atrybutów właściwości modelu.
- `asp-page` – pozwala ustawić wartość atrybutu `href` odnośnika HTML na konkretną stronę. Poprzedzenie strony znakiem `/` tworzy adres URL strony pasujący z katalogu głównego aplikacji.
- `asp-route-{value}` – wartość podana jako {value} jest interpretowana jako potencjalny parametr trasy. Wykorzystywany na stronach Index.cshtml i Details.cshtml Np.:
  `<a asp-page="./Details" asp-route-id="@item.Id">Details</a>`
  dla `item.Id` równego 1 wygeneruje odnośnik kierujący do strony wyświetlającej szczegóły pierwszej encji.
- `asp-validation-for` – stosowany z elementem div, wyświetla komunikat o błędzie walidacji dla pojedynczej właściwości Modelu.
- `asp-validation-summary` – stosowany z elementem `span`, wyświetla podsumowanie komunikatów o błędach walidacji formularza dla wszystkich właściwości. Możliwe wartości to:
  - `All` – podsumowywanie błędów walidacji modelu oraz właściwości,
  - `ModelOnly` – podsumowanie jedynie błędów walidacji modelu,
  - `None` – (domyślny) brak podsumowania walidacji.
- `asp-for`[6] – stosowany z tagami pomocniczymi `label`, `input`. Wyodrębnia nazwę określonej właściwości modelu do renderowanego kodu HTML. Wartość atrybutu to prawa strona wyrażenia lambda, zatem `asp-for="Prop.PropIn"` w wygenerowanym kodzie zostanie rozwinięte do: `m => m.Prop.PropIn` co zwalnia z konieczności stosowania prefixu modelu. Można użyć znaku `@`, aby rozpocząć wyrażenie inline:

```
@{
    var joe = "Joe";
}

<input asp-for="@joe">
```

6. Lokalizacja

Już od początku tworzenia projektu warto pomyśleć o lokalizacji. Wszelkie zasoby które będą zależne od wybranego języka/kultury finalnie są przechowywane w tzw. „satellite assembly”

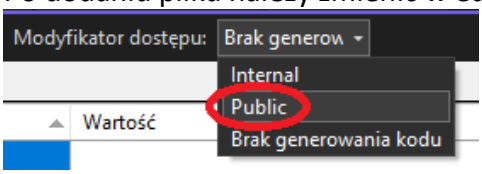
[5] https://learn.microsoft.com/en-us/aspnet/core/mvc/views/working-with-forms?view=aspnetcore-7.0#the-form-action-tag-helper

[6] https://learn.microsoft.com/en-us/aspnet/core/mvc/views/working-with-forms

znajdujących się w odpowiadających im podkatalogach (reprezentowanych kodem wspieranej kultury[7]) katalogu zawierającego dane wyjściowe kompilacji (przeważnie katalog „**bin**\<konfiguracja>\<wersja_frameworka>”). Przykładowo w podkatalogu „pl” będą przechowywane assembly satelickie powiązane z polskim językiem/kulturą. Podczas kompilacji we wspomnianych assembly osadzane są odpowiadające temu samemu językowi/kulturze binarne pliki zasobów (z rozszerzeniami *.<kod_kultury>.resources) widoczne w katalogu zawierającego tymczasowe liki kompilacji (przeważnie katalog „**obj**\<konfiguracja>\<wersja_frameworka>”). Pliki te można generować na wiele sposobów[8] jednak preferowaną metodą jest wykorzystanie edytora zasobów Visual Studio[9] w celu utworzenia plików *.resx będących podstawą do wygenerowania odpowiednich satellite assembly.

Pliki zasobów dla konkretnych stron RP powinny znajdować się katalogu Resources\Pages w odpowiednich podkatalogach analogicznie jak ma to miejsce w folderze Pages. jeśli i powinny mieć nazwy odpowiadające

Po dodaniu pliku należy zmienić w edytorze modyfikator dostępu na publiczny:



Dodać dyrektywę (najlepiej w pliku _ViewImports.cshtml – będzie widoczna na wszystkich stronach):

```
@using Microsoft.Extensions.Localization
@using Microsoft.AspNetCore.Localization
```

Nazewnictwo plików zasobów (.resx)

Kluczowe jest ścisłe przestrzeganie konwencji nazewnictwa, aby platforma ASP.NET Core mogła automatycznie powiązać zasoby z odpowiednimi stronami.

- **Dla stron Razor:** Plik zasobów musi odzwierciedlać pełną ścieżkę i nazwę pliku `.cshtml`, względem katalogu `Resources`.
  - **Strona:** `Pages/Products/Index.cshtml`
  - **Zasób (PL):** `Resources/Pages/Products/Index.cshtml.pl.resx`
  - **Zasób (DE):** `Resources/Pages/Products/Index.cshtml.de.resx`
- **Dla zasobów współdzielonych:** Czasami chcemy mieć jeden plik dla zasobów używanych w całej aplikacji (np. "Zapisz", "Anuluj"). Tworzymy wtedy klasę-znacznik (np. `SharedResources.cs` w folderze `Resources`) i pliki `.resx` o tej samej nazwie.
  - **Zasób (PL):** `Resources/SharedResources.pl.resx`

---

[7] https://learn.microsoft.com/en-us/bingmaps/rest-services/common-parameters-and-types/supported-culture-codes

[8] https://learn.microsoft.com/en-us/dotnet/core/extensions/create-resource-files

[9] https://learn.microsoft.com/en-us/dotnet/core/extensions/create-resource-files#resource-files-in-visual-studio

- **Zasób (EN):** `Resources/SharedResources.en.resx`

Aby lokalizacja działała, należy ją skonfigurować w pliku `Program.cs`.

1. **Rejestracja usług:** Musimy poinformować aplikację, gdzie szukać plików zasobów oraz dodać obsługę lokalizacji do stron Razor.

```
// Ustawia domyślny katalog dla plików .resx
builder.Services.AddLocalization(options => options.ResourcesPath =
"Resources");

builder.Services.AddRazorPages()
    // Włącza lokalizację dla widoków i stron Razor
    .AddViewLocalization()
    // Włącza lokalizację dla adnotacji DataAnnotations (np. [Required])
    .AddDataAnnotationsLocalization();
```

2. **Konfiguracja potoku (Pipeline):** Musimy określić, które języki wspieramy i jak aplikacja ma decydować, którego języka użyć (np. na podstawie ciasteczka, nagłówka przeglądarki lub parametru w URL).

```
// Definiujemy wspierane kultury
var supportedCultures = new[] { "en-US", "pl-PL", "de-DE" };

var localizationOptions = new RequestLocalizationOptions()
    .SetDefaultCulture(supportedCultures[0])
    .AddSupportedCultures(supportedCultures)
    .AddSupportedUICultures(supportedCultures);

// Dodaj dostawców określania kultury, np.:
// localizationOptions.AddInitialRequestCultureProvider(new
QueryStringRequestCultureProvider()); // ?culture=pl-PL
// localizationOptions.AddInitialRequestCultureProvider(new
CookieRequestCultureProvider());

// Dodajemy middleware lokalizacji do potoku
app.UseRequestLocalization(localizationOptions);

// ...
app.MapRazorPages();
```

3. **Użycie w kodzie:** Należy dodać dyrektywy `@using` (najlepiej w pliku `_ViewImports.cshtml`, aby były widoczne globalnie):

```
@using Microsoft.Extensions.Localization
@using Microsoft.AspNetCore.Localization
```

Następnie wstrzykujemy usługę `IStringLocalizer` do naszej strony `.cshtml` lub klasy `PageModel`, aby odczytywać przetłumaczone ciągi znaków. W pliku `.cshtml`:

```
@inject IStringLocalizer<IndexModel> localizer

<h1>@localizer["WelcomeHeader"]</h1>
<p>@localizer["WelcomeMessage"]</p>
```

W klasie `PageModel`:

```
private readonly IStringLocalizer<IndexModel> _localizer;

public IndexModel(IStringLocalizer<IndexModel> localizer)
{
    _localizer = localizer;
}

public void OnGet()
{
    // Odczytanie przetłumaczonego ciągu w kodzie C#
    ViewData["Title"] = _localizer["PageTitle"];
}
```

Implementacja ComboBox'a (Listy rozwijalnej)

Listy rozwijane (`<select>`) są kluczowe w formularzach CRUD do wybierania encji powiązanych (np. wybór Producenta dla Produktu). Najlepszą praktyką jest unikanie `ViewData` lub `ViewBag` na rzecz silnie typowanych właściwości w `PageModel`.

Załóżmy, że tworzymy lub edytujemy `Product`, który musi mieć przypisanego `Producer`.

1. **Klasa `PageModel` (np. `CreateModel.cs`):** Definiujemy właściwość na model (np. `Product`) oraz drugą właściwość trzymającą listę opcji (np. `SelectList`).

```
public class CreateModel : PageModel
{
    private readonly YourDbContext _context;

    public CreateModel(YourDbContext context)
    {
        _context = context;
    }

    [BindProperty]
    public Product Product { get; set; }

    // Właściwość przechowująca listę producentów dla ComboBox'a
    public SelectList ProducerOptions { get; set; }
```

```
    public async Task OnGetAsync()
    {
        // Ładujemy listę opcji podczas ładowania formularza
        // "Id" to wartość (value), "Name" to tekst wyświetlany
        ProducerOptions = new SelectList(
            await _context.Producer.ToListAsync(), "Id", "Name");
    }

    public async Task<IActionResult> OnPostAsync()
    {
        if (!ModelState.IsValid)
        {
            // Jeśli walidacja się nie powiodła, musimy ponownie
            // załadować listę opcji, inaczej będzie pusta!
            ProducerOptions = new SelectList(
                await _context.Producer.ToListAsync(), "Id", "Name");
            return Page();
        }

        // Product.ProducerID zostanie automatycznie powiązane
        // z wartością wybraną w ComboBox'ie
        _context.Product.Add(Product);
        await _context.SaveChangesAsync();

        return RedirectToPage("./Index");
    }
}
```

2. **Strona Razor (`.cshtml`):** Używamy Tag Helperów `asp-for` i `asp-items`, aby powiązać listę z modelem.

```
@page
@model YourProject.Pages.Products.CreateModel

<form method="post">
    <div asp-validation-summary="ModelOnly" class="text-danger"></div>

    <div class="form-group">
        <label asp-for="Product.Name" class="control-label"></label>
        <input asp-for="Product.Name" class="form-control" />
        <span asp-validation-for="Product.Name" class="text-danger"></span>
    </div>

    <div class="form-group">
        <label asp-for="Product.ProducerID" class="control-label"></label>
        <select asp-for="Product.ProducerID"
                class="form-control"
```

```
                asp-items="Model.ProducerOptions">

            <option value="">-- Wybierz producenta --</option>
        </select>
        <span asp-validation-for="Product.ProducerID" class="text-
danger"></span>
    </div>

    <div class="form-group">
        <input type="submit" value="Utwórz" class="btn btn-primary" />
    </div>
</form>
```

To podejście jest preferowane, ponieważ jest silnie typowane, łatwiejsze do testowania i mniej podatne na błędy niż używanie `ViewBag`.

Dostosowywanie nazw wyświetlanych (Metadane modelu)

W ASP.NET Core za generowanie etykiet (np. w `<label asp-for="...">` lub przez `@Html.DisplayNameFor`) odpowiada system metadanych modelu (ang. *Model Metadata*).

Domyślny dostawca metadanych (`IModelMetadataProvider`) automatycznie analizuje nazwy właściwości w formacie **PascalCase** i wstawia między nimi spacje.

- Jeśli właściwość w modelu nazywa się: `public string FirstName { get; set; }`
- To `DisplayNameFor` lub `<label asp-for="FirstName">` domyślnie wygeneruje etykietę: **"First Name"**.
- Podobnie dla `ProducerID` będzie to **"Producer ID"**.

Nie trzeba więc implementować tej funkcjonalności samodzielnie. Jeśli domyślna nazwa (np. "First Name") nam nie odpowiada i chcemy ją zmienić (np. na "Imię" lub "First name (required)"), używamy atrybutu `[Display]` z `System.ComponentModel.DataAnnotations`:

```
using System.ComponentModel.DataAnnotations;

public class Product
{
    public int Id { get; set; }

    [Display(Name = "Nazwa produktu")]
    public string Name { get; set; }

    [Display(Name = "Producent")]
    public int ProducerID { get; set; }
    public Producer Producer { get; set; }
}
```

Teraz, dzięki temu atrybutowi, helper `@Html.DisplayNameFor(model => model.Product.Name)` oraz `<label asp-for="Product.Name">` wygenerują etykietę "**Nazwa produktu**" (lub "**Producent**" dla `ProducerID`), zamiast domyślnych "Name" czy "Producer ID". Jest to również kluczowe dla lokalizacji, gdyż atrybut `[Display]` można połączyć z plikami zasobów `.resx`.

3. Minimal API i Swagger

Jeśli chcemy udostępnić proste API nie wymagające dużej liczby zależności oraz linii kodu to pomocne mogą się okazać następujące funkcje klasy `EndpointRouteBuilderExtensions` rozszerzającej `IEndpointRouteBuilder`:

- `MapGet`, `MapPost`, `MapPut`, `MapDelete` – dodają endpoint pasujący do określonego rodzaju żądania HTTP. Jako pierwszy parametr podajemy szablon trasy w postaci stringu, np.: `"/api/SomeEntity/{id:int}"` dla metody `MapGet` spowoduje przekazanie parametru `id` typu `int`. Jako drugi parametr przekazywany jest delegat obsługujący dane żądanie.
- `MapMethods` – to samo co powyższe ale rodzaje obsługiwanych żądań podajemy w postaci listy stringów jako drugi parametr, a delegat obsługi żądania jako trzeci parametr.

Do powyższych metod jako ostatni parametr przekazywany jest delegat obsługujący dane żądanie. Jest to metoda anonimowa/wyrażenie lambda przyjmująca jako parametry:

- parametry pobrane z szablonu trasy;
- obiekt zarejestrowanej usługi, która zostanie wykorzystana do obsługi żądania.

W przypadku parametru zwracanego należy wykorzystać odpowiednie metody klasy `Results`[10] . np. (podano tylko dwie najczęściej wykorzystywane):

- `Ok` – odpowiedź HTTP 200 OK z zawartością przekazanego do niej parametru zawartą w treści odpowiedzi http,
- `NoContent` – odpowiedź HTTP 204 (No Content).

Jeśli chcemy aby metoda nie wymagała autoryzacji możemy wykorzystać atrybut `[AllowAnonymous]`:

```
app.MapMethods("/api/SomeEntity/Add", new[] { "POST", "PUT" },
[AllowAnonymous] (SomeEntity se, ISomeService service) =>
{
    try
    {
        service.Add(se);
    }
    catch (Exception ex)
    {
        return Results.Problem(
            detail: ex.StackTrace,
            title: ex.Message );
    }
    return Results.Ok();
});
```

[10] https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.http.results

Wspomniany obiekt usługi, obsługującej żądanie musi zostać zarejestrowany z wykorzystaniem jednej z metod `AddTransient`, `AddScoped`, `AddSingleton`, klasy `ServiceCollectionServiceExtensions` rozszerzającej interfejs `IServiceCollection`, przykładowo:

```
builder.Services.AddScoped<ISomeService, SomeService>();
```

Rejestrują one usługę typu `ISomeService` z implementacją w `SomeService`. Różnią się między sobą czasem życia:

- `AddTransient` – zawsze nowy obiekt,
- `AddScoped` – ten sam obiekt w ramach jednego żądania,
- `AddSingleton` – zawsze ten sam obiekt.

Tak udostępnione API możemy łatwo przeglądać, wywoływać i dokumentować wykorzystując toolset Swagger. Wystarczy wywołać odpowiednie metody klasy `SwaggerBuilderExtensions` rozszerzającej interfejs `IApplicationBuilder` z biblioteki `Swashbuckle.AspNetCore`:

```
app.UseSwagger();
app.UseSwaggerUI();
```

Tak udostępniona dokumentacja w postaci *.json opisująca dany endpoint powinna być domyślnie widoczna pod adresem:
http://<adres_usługi>:<port>/swagger/v1/swagger.json

a UI do wizualnego jej renderowania oraz pozwalające do interakcji z naszym API pod adresem:
http:// <adres_usługi>:<port>/swagger.

W celu wykorzystania naszego API z poziomu aplikacji klienckiej można posłużyć się instancją klasy `HttpClient`[11] i wykorzystać jej metody: `GetAsync`, `PostAsync`, itp. W przypadku `PostAsync` utworzony obiekt należy wcześniej zserializować. Można w tym celu wykorzystać metodę `SerializeObject` klasy statycznej `JsonConvert` z biblioteki `Newtonsoft.Json`:

```
HttpResponseMessage? httpResponseMessage = null;
try
{
    string serializedSomeEntity = JsonConvert.SerializeObject(SomeEntity);
    StringContent byteContentJSON = new StringContent(
        serializedSomeEntity,
        Encoding.UTF8,
        "application/json");
    httpResponseMessage = await client.PostAsync("/api/SomeEntity/Add",
byteContentJSON);
}
catch (ArgumentNullException uex)
{
    string infoMessage = " | URL missing or invalid.";
    Console.WriteLine("Error", new { msg = uex.Message + infoMessage });
    return ("Error" + uex.Message.ToString() + infoMessage, false);
}
```

[11] https://learn.microsoft.com/pl-pl/dotnet/api/system.net.http.httpclient

```
catch (JsonReaderException jex)
{
    string infoMessage = " | Json data could not be read.";
    Console.WriteLine("Error", new { msg = jex.Message + infoMessage });
    return ("Error" + jex.Message.ToString() + infoMessage, false);
}
catch (Exception ex)
{
    string infoMessage = " | Are you missing some Json keys and values?
Please check your Json data.";
    Console.WriteLine("Error", new { msg = ex.Message + infoMessage });
    return ("Error" + ex.Message.ToString() + " | Are you missing some Json
keys and values? Please check your Json data.", false);
}
if (httpResponseMessage.IsSuccessStatusCode)
{
    return ("SomeEntity added succesfully", true);
}
else
{
    return (httpResponseMessage.Content.ToString(), false);
}
```

Do komunikacji między procesami, np. poprzez API tak jak w naszym przypadku należy zdefiniować nowe obiekty transferu danych tzw. DTO. Są to tak samo jak w przypadku modelu Encji (DAO – Data Access Object) zwykłe klasy. Pozwoli to na oddzielenie modelu domeny od warstwy prezentacji pozwalając na niezależną zmianę tych obydwu.

4. Uwierzytelnianie i autoryzacja

W ASP.NET Core uwierzytelnianie jest obsługiwane przez usługę uwierzytelniania IAuthenticationService, która jest używana przez oprogramowanie pośredniczące uwierzytelniania. Wybierając, podczas tworzenia projektu uwierzytelnianie na poziomie Pojedyncze konta, zostanie wygenerowany nowy obszar Identity zawierający klasę kontekstu danych dla użytkowników `IdentityContext : IdentityDbContext<IdentityUser>` oraz strony Razor służące do rejestracji, logowania itp. (folder Pages/Account) oraz służące do zarządzania kontem (folder Pages/Account/Manage). Jedyne co musimy zrobić to dodać zarejestrować nowy kontekst danych dla użytkowników:

```
builder.Services.AddDbContext<IdentityContext>(
            dbContextOptions => dbContextOptions
                .UseMySql(connectionStringForIdentity, serverVersion)
                // The following three options help with debugging, but
should
                // be changed or removed for production.
                .LogTo(Console.WriteLine, LogLevel.Information)
                .EnableSensitiveDataLogging()
                .EnableDetailedErrors()
        );
```

W przypadku wybrania polityki kontroli ról użytkowników opartej na zasadach. Deweloper rejestruje zasady podczas uruchamiania aplikacji w ramach konfiguracji usługi autoryzacji. Dodanie usługi polityki autoryzacji (metoda `AddAuthorization` i `AddPolicy`)

```
builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("RequireAdministratorRole",
         policy => policy.RequireRole("Administrator"));
});
```

Zasady są stosowane przy użyciu właściwości Policy w atrybucie [Authorize], lub z wykorzystaniem metod `AuthorizeFolder` i `AuthorizePage`:

```
builder.Services.AddRazorPages(options =>
{
    options.Conventions.AuthorizePage("/SomePage");
    options.Conventions.AuthorizeFolder("/SomeFolder");
});
```

Użycie mechanizmów uwierzytelniania i autoryzacji należy włączyć:

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

Podczas rejestracji wymagana jest możliwość wysłania maila do użytkownika w tym celu należy zaimplementować interfejs `IEmailSender`, który udostępnia metodę `SendEmailAsync` warto skorzystać z jednej z szeroko dostępnych usług np.: SendGrid, SendInBlue, EmailLabs, MailGun. Poniżej przykład implementacji z wykorzystaniem klasy `MailgunSender`.

```
public EmailSender(ILogger<EmailSender> logger)
{
    _logger = logger;
}

public async Task SendEmailAsync(string toEmail, string subject, string message)
{
    var fromEmail = "sender@email.adress";
    var sender = new MailgunSender(
        "mg.devteam.net.pl", // Mailgun Domain
        "Mailgun_API_Key", // Mailgun API Key
        MailGunRegion.EU
    );
    Email.DefaultSender = sender;
    // glitch in asp.net core
    if (message.Contains("amp;"))
    {
        message = message.Replace("amp;", "");
    }
    var email = Email
        .From(fromEmail)
        .To(toEmail)
        .Subject(subject)
        .Body(message);

    var response = await email.SendAsync();
    _logger.LogInformation(response.Successful
                ? $"Email to {toEmail} queued successfully!"
                : $"Failure Email to {toEmail}");
}
```

W wygenerowanych linkach potwierdzających rejestracje użytkownika najprawdopodobniej pojawią się ciągi `&` które mogą zostać nieprawidłowo zinterpretowane przez Asp.NET

Core. Rozwiązaniem w przypadku zaistnienia błędów z uwierzytelnieniem jest pozbycie się nadmiarowej części `amp;` ze stringa aby uniknąć wielokrotnej konwersji `&` na `&` podczas kodowania ciągów URL w HTML.

Alternatywnie w aplikacji https://app.sendgrid.com/ po zarejestrowaniu się (dostępne jest darmowe konto Free 100 emails/day) i zalogowaniu należy wygenerować klucze do API (zakładka Settings → API keys → Create API Key). Klucz należy zachować ponieważ – ze względów bezpieczeństwa – zostanie wyświetlony tylko raz po wygenerowaniu.
W celu wysyłania maila potrzebna będzie bibliotek udostępniana przez twórców o nazwie SendGrid (najłatwiej dodać ją menedżerem pakietów Nuget). Ciało funkcji `EmailSender` wygląda bardzo podobnie.

```
var client = new SendGridClient(
    "SendGrid_API_Key ");
var to = new EmailAddress(email);
var from = new EmailAddress("sender@email.adress");
if (htmlMessage.Contains("amp;"))
{
    htmlMessage = htmlMessage.Replace("amp;", "");
}
var message = MailHelper.CreateSingleEmail(from, to, subject, null, htmlMessage);
var response = client.SendEmailAsync(message);
_logger.LogInformation(response.IsCompletedSuccessfully ? $"Email to {email}
queued successfully!" : $"Failure Email to {email}");
```