

UWIERZYTELNIANIE, AUTORYZACJA, TOŻSAMOŚĆ

Programowanie wizualne

Wojciech Frohmberg, Instytut Informatyki, Politechnika Poznańska

2023



UWIERZYTELNIANIE

Uwierzytelnianie (ang. authorization) to proces, w którym wyryfikowane jest oświadczenie użytkownika dotyczące jego tożsamości. Proces ten najczęściej odbywa się poprzez wskazanie loginu (tj. nazwy użytkownika) oraz klucza – najczęściej hasła, które powinno być znane tylko i wyłącznie danemu użytkownikowi, czasem (np. podczas korzystania z dwustopniowego uwierzytelniania) klucz może być generowany tylko i wyłącznie na potrzeby pojedynczego uwierzytelniania.

AUTORYZACJA

Autoryzacja jest procesem określającym do jakich działań użytkownik jest uprawniony (np. proces w procesie autoryzacji dowiemy się czy określony użytkownik ma uprawnienia do oglądania określonej strony). Proces ten może wymagać wstępnego uwierzytelnienia użytkownika.

TOŻSAMOŚĆ

Tożsamością użytkownika będziemy określali zbiór poświadczeń (ang. claim) dotyczących uprawnień danego użytkownika.

Poświadczenia powinny być nadane przez odpowiednie jednostki zwane urzędami certyfikacji (ang. certificate authority).

Autoryzacja najczęściej wykonywana jest na podstawie poświadczeń nt. uprawnienia użytkownika.

The background of the entire image is a dark, textured surface covered with a repeating pattern of stylized white and light gray floral and scrollwork motifs. The motifs include large, multi-petaled flowers, smaller circular floral designs, and intricate swirling lines with small leaf-like details.

ASP.NET CORE IDENTITY

DLACZEGO WARTO KORZYSTAĆ Z ASP.NET CORE IDENTITY?

Ponieważ funkcjonalność biblioteki była już niejednokrotnie wykorzystana i jest wszechstronnie przetestowana.

Bezpieczeństwo aplikacji nie powinno stanowić elementu, który podlegałby rozwojowi metodą prób i błędów.

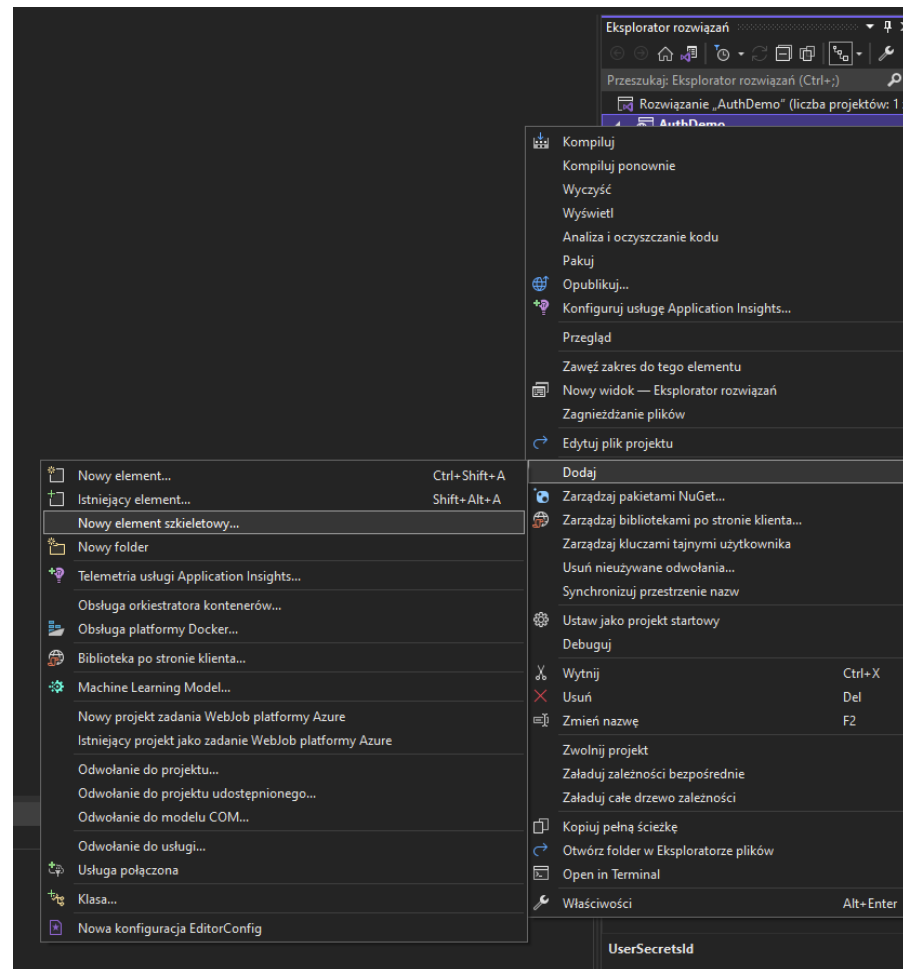
Wbudowane w bibliotekę mechanizmy pozwalają na dowolną adoptowalność do potrzeb aplikacji. Dodatkowo biblioteka posiada ujednolicony interfejs pozwalający na integrację z zewnętrznym urzędem uwierzytelniania np. Google czy Facebook.

NUGETY NIEZBĘDNE DO KORZYSTANIA Z ASP.NET CORE IDENTITY

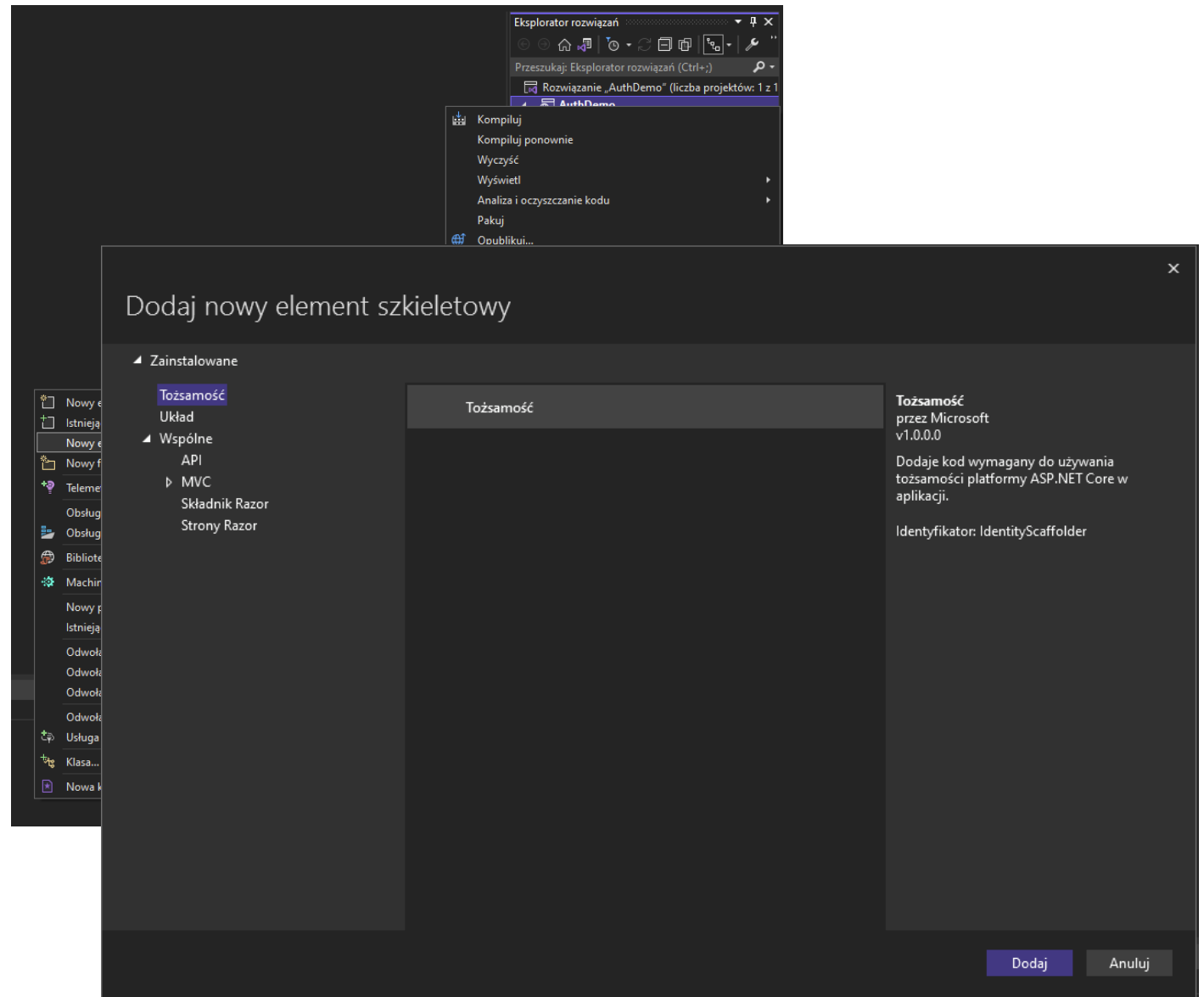
W celu korzystania z funkcjonalności niezbędne będzie zainstalowanie następujących Nugetów:

- `Microsoft.AspNetCore.Identity.UI` – trzon biblioteki udostępniającej zarządzanie uprawnieniami i tożsamością,
- `Microsoft.VisualStudio.Web.CodeGeneration.Design` – pakiet, przy użyciu którego wygenerujemy widoki rejestracji, logowania czy zarządzania kontem,
- `Microsoft.AspNetCore.Identity.EntityFrameworkCore` – znana nam biblioteka służąca do mapowania obiektów na wpisy w bazie danych, przyda nam się do abstrahowania od fizycznego zapisu informacji dot. uprawnień i tożsamości w tabelach bazodanowych,
- `Microsoft.EntityFrameworkCore.Sqlite` oraz `Microsoft.EntityFrameworkCore.SqlServer` – providery połączeń z bazą danych,

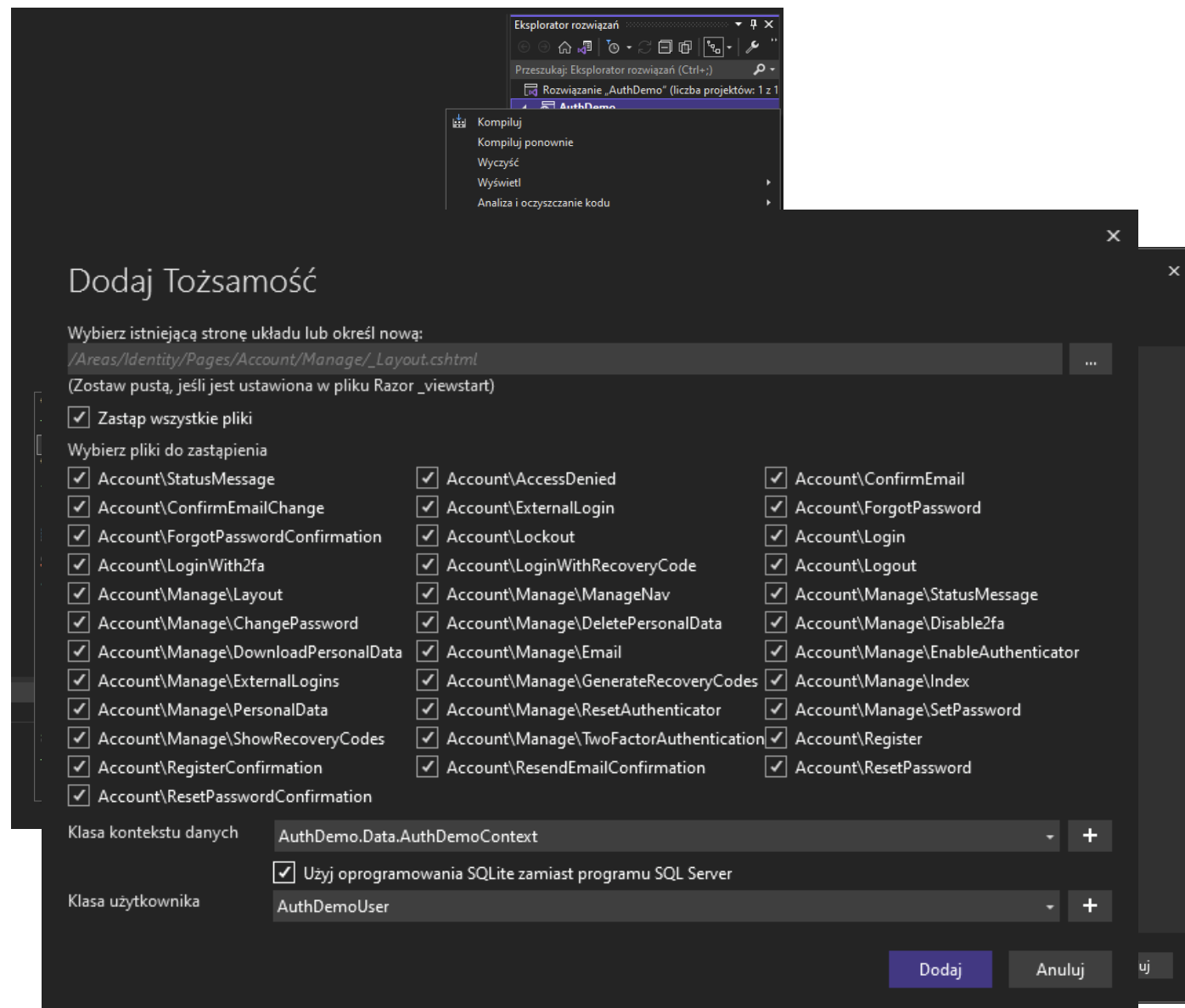
JAK ROZPOCZĄĆ KORZYSTANIE Z ASP.NET IDENTITY W PROJEKCIE MVC?

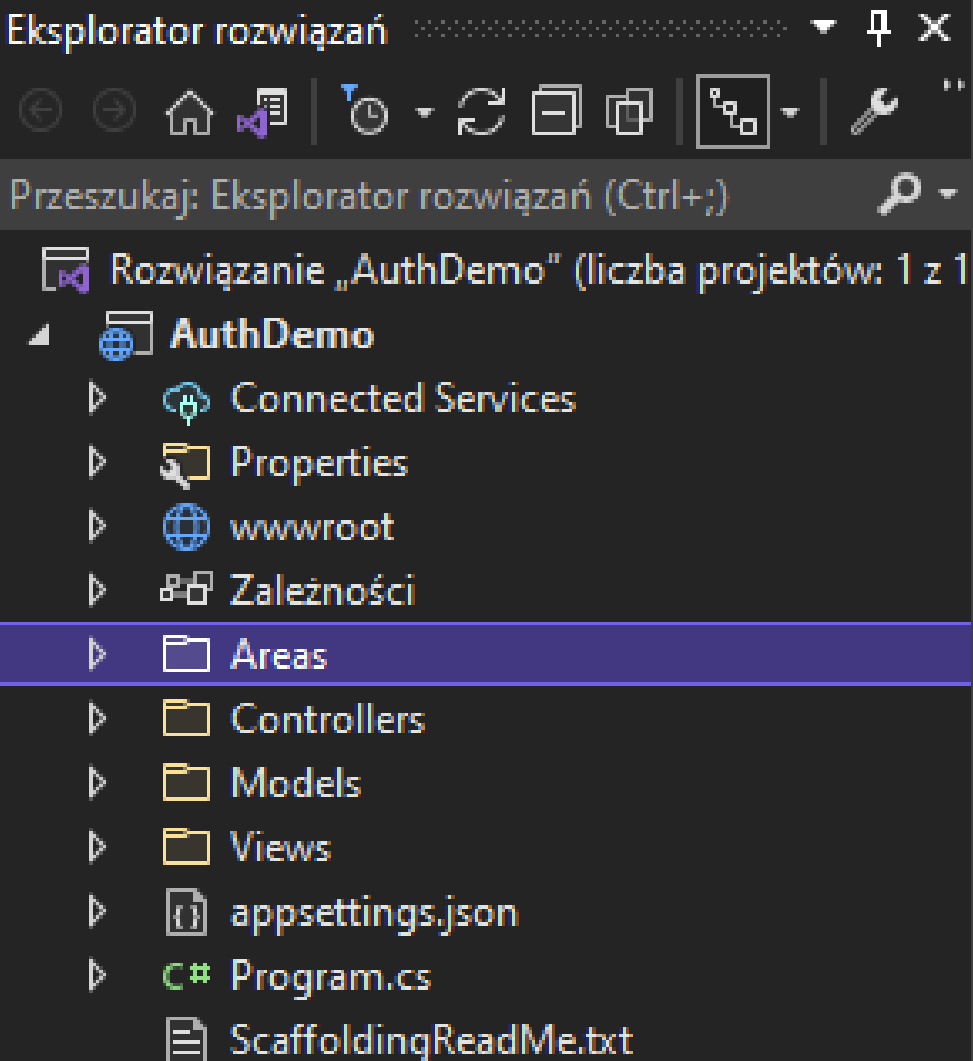


JAK ROZPOCZĄĆ KORZYSTANIE Z ASP.NET IDENTITY W PROJEKCIE MVC?



JAK ROZPOCZĄĆ KORZYSTANIE Z ASP.NET IDENTITY W PROJEKCIE MVC?





Do folderu projektu zostanie dodany folder Areas z wygenerowanymi elementami. Póki co jednak w ramach naszej aplikacji fizycznie nie zostały zagnieżdżone widoki uwierzytelniania/rejestracji użytkowników, nie podpięta została również baza danych. Żeby to naprawić musimy wykonać następujące czynności:

- Zmodyfikować plik Views/Shared/_Layout.cshtml dodając do navbaru na końcu elementu o klasie "container-fluid" tag `<partial name="_LoginPartial"/>`
- Do pliku Program.cs przed uruchomieniem aplikacji (`app.Run()`) musimy dorzucić instrukcję `app.MapRazorPages();` co pozwoli zmapować wygenerowane widoki zgodnie z naszymi oczekiwaniami. Do pliku należy jeszcze dodać brakujące użycie przestrzeni nazw w której znajduje się wygenerowana klasa użytkownika (w przykładzie jest to: `using Auth.Areas.Identity.Data;`)
- Dodać migrację inicjalizacyjną nowej bazy danych i zaaplikować zmiany.

UWAGA jeśli w ramach naszego projektu korzystaliśmy już wcześniej z bazy danych poprzez EF może zaistnieć konieczność dodania ręcznie kontekstu bazodanowego w pliku Program.cs:

```
var connectionString =
builder.Configuration.GetConnectionString("AuthDemoContextConnection");

builder.Services.AddDbContext<AuthDemoContext>(options =>
options.UseSqlite(connectionString));
```

Oczywiście po uwzględnieniu zmian w nazwie kontekstu :) Sama migracja może również wymagać podania w parametrze nazwy kontekstu zarówno przy Add-Migration jak i przy Update-Database.

REJESTRACJA NOWEGO UŻYTKOWNIKA

Po wspomnianych czynnościach w naszej aplikacji pokaże się sekcja logowania w ramach której będziemy mogli się zarejestrować.

[AuthDemo](#) [Home](#) [Privacy](#)

[Register](#) [Login](#)

Welcome

Learn about [building Web apps with ASP.NET Core](#).

Po przejściu rejestracji od razu zostaniemy poproszeni o potwierdzenie rejestracji:

[AuthDemo](#) [Home](#) [Privacy](#)

[Register](#) [Login](#)

Register confirmation

This app does not currently have a real email sender registered, see [these docs](#) for how to configure a real email sender. Normally this would be emailed: [Click here to confirm your account](#)

Docelowo funkcję tą należy oczywiście wyłączyć poprzez zmodyfikowanie pliku `Areas/Pages/Account/RegisterConfirmation.cshtml.cs` i ustawienie wartości false pod zmienną `DisplayConfirmAccountLink`

OGRANICZENIE DOSTĘPU DO KONTROLERA I/LUB AKCJI

Ograniczenia dostępu dokonuje się poprzez zastosowanie atrybutu `Authorize`:

```
1 using AuthDemo.Models;
2 using Microsoft.AspNetCore.Authorization;
3 using Microsoft.AspNetCore.Mvc;
4 using System.Diagnostics;
5
6 namespace AuthDemo.Controllers
7 {
8     [Authorize]
9     public class HomeController : Controller
10     {
11         private readonly ILogger<HomeController> _logger;
12
13         public HomeController(ILogger<HomeController> logger)
14         {
15             _logger = logger;
16         }
17
18         public IActionResult Index()
19         {
20             return View();
21         }
22     }
23 }
```

Po dodaniu atrybutu i próbie dostania się do dowolnej akcji tego kontrolera nastąpi przekierowanie do strony logowania. Dopiero zalogowany użytkownik będzie mógł oglądać stronę. Atrybut można również podczepić pod akcję metody ograniczając dostęp tylko do wskazanej akcji:

```
4 using System.Diagnostics;
5
6 namespace AuthDemo.Controllers
7 {
8     public class HomeController : Controller
9     {
10         private readonly ILogger<HomeController> _logger;
11
12         public HomeController(ILogger<HomeController> logger)
13         {
14             _logger = logger;
15         }
16
17         public IActionResult Index()
18         {
19             return View();
20         }
21
22         [Authorize]
23         public IActionResult Privacy()
24         {
25             return View();
26         }
27     }
28 }
```

ROLE UŻYTKOWNIKÓW

Celem grupowego określania przywilejów użytkowników możemy wykorzystywać tzw. role użytkowników. Żeby zarządzać użytkownikami w kontekście ról będziemy musieli dodać do dostępnych serwisów naszej aplikacji serwis odpowiadający za role, zarządzanie rolami oraz zarządzanie użytkownikami. Do tego celu instrukcję pliku Program.cs:

```
builder.Services.AddDefaultIdentity<AuthDemoUser>(
    options => options.SignIn.RequireConfirmedAccount = true)
    .AddEntityFrameworkStores<AuthDemoContext>();
```

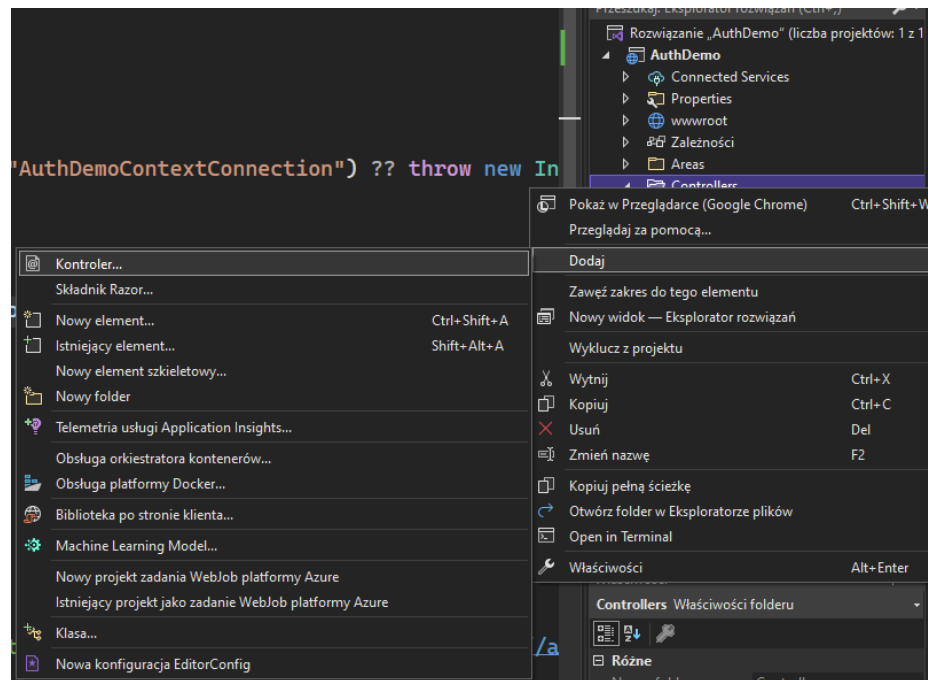
Podmieniamy na:

```
builder.Services.AddDefaultIdentity<AuthDemoUser>(
    options => options.SignIn.RequireConfirmedAccount = true)
    .AddRoles<IdentityRole>()
    .AddRoleManager<RoleManager<IdentityRole>>()
    .AddUserManager<UserManager<AuthDemoUser>>()
    .AddEntityFrameworkStores<AuthDemoContext>();
```

Po tej operacji w ramach dowolnego kontrolera będziemy mogli przy użyciu mechanizmu dependency injection wstrzykiwać obiekt managera ról oraz użytkowników.

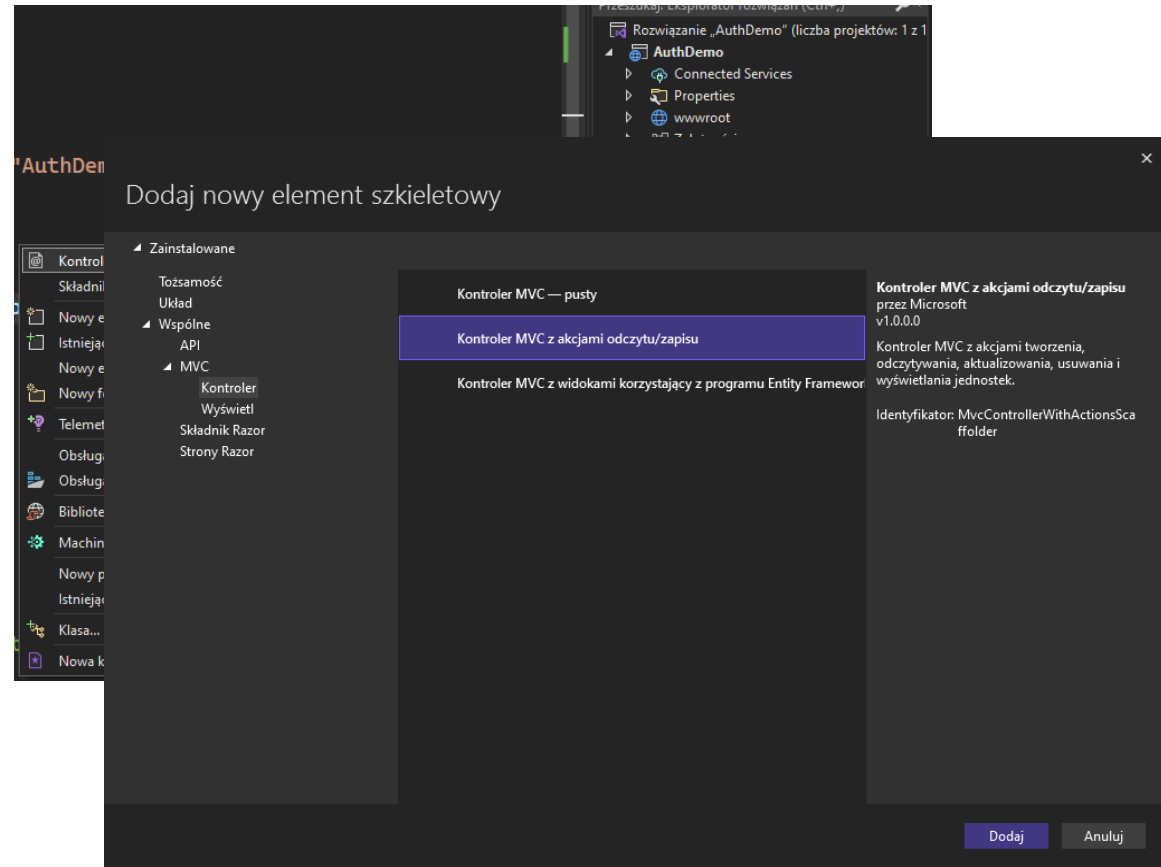
ZARZĄDZANIE ROLAMI

W celu zarządzania rolami można utworzyć kontroler
RoleController:



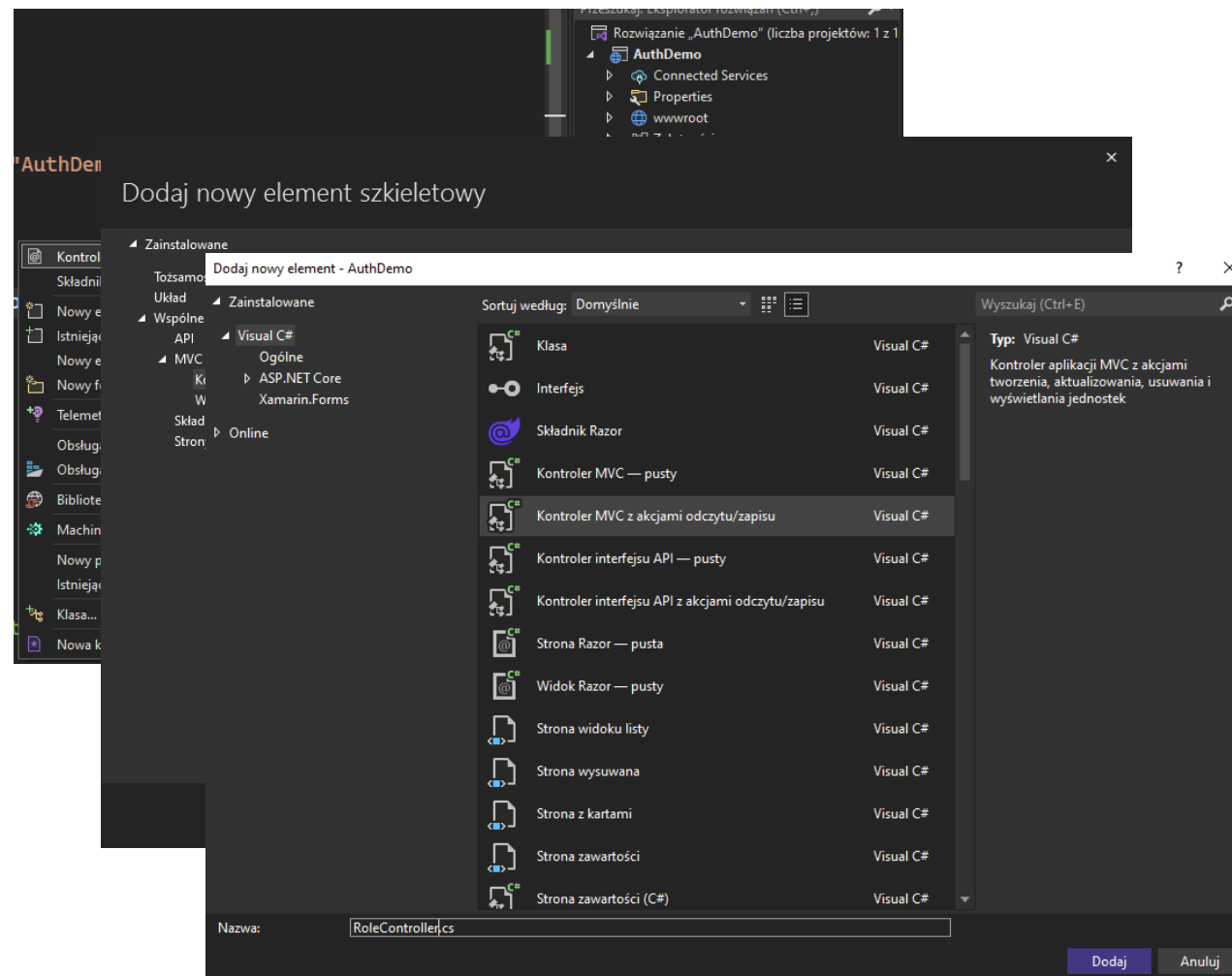
ZARZĄDZANIE ROLAMI

W celu zarządzania rolami można utworzyć kontroler RoleController:



ZARZĄDZANIE ROLAMI

W celu zarządzania rolami można utworzyć kontroler
RoleController:



ZARZĄDZANIE ROLAMI

W celu chwilowego składowania danych dotyczących ról będziemy potrzebowali klasy modelu np. RoleInfo z kodem:

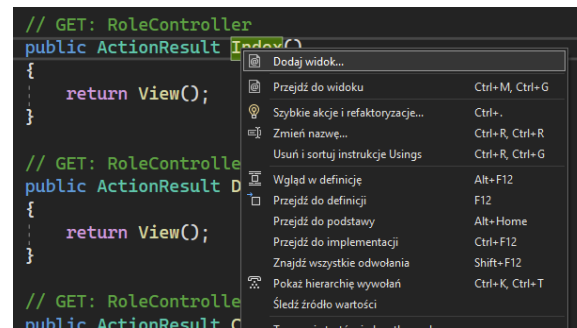
```
public class RoleInfo
{
    public string Id { get; set; }
    public string Name { get; set; }
}
```

Do kontrolera ról wygenerowanego w kroku z poprzedniego slajdu możemy teraz wstrzyknąć menagera użytkowników i ról:

```
protected RoleManager<IdentityRole> RoleManager { get; }
protected UserManager<AuthUser> UserManager { get; }

public RoleController(RoleManager<IdentityRole> roleManager,
    UserManager<AuthUser> userManager)
{
    RoleManager = roleManager;
    UserManager = userManager;
}
```

Teraz możemy dodać widok do akcji Index:



ZARZĄDZANIE ROLAMI

W celu chwilowego składowania danych dotyczących ról będziemy potrzebowali klasy modelu np. RoleInfo z kodem:

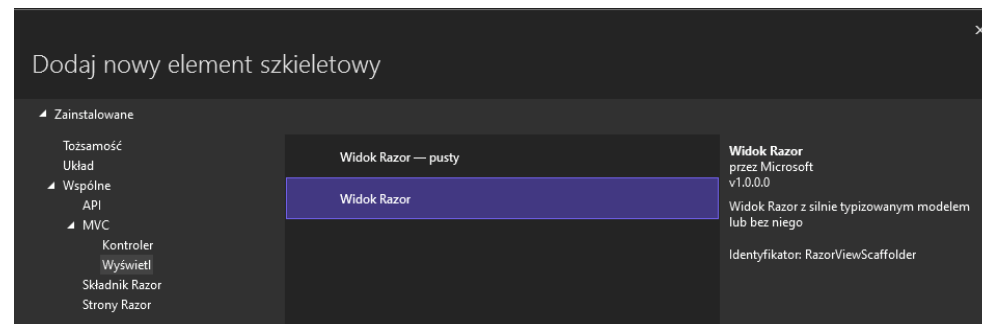
```
public class RoleInfo
{
    public string Id { get; set; }
    public string Name { get; set; }
}
```

Do kontrolera ról wygenerowanego w kroku z poprzedniego slajdu możemy teraz wstrzyknąć menagera użytkowników i ról:

```
protected RoleManager<IdentityRole> RoleManager { get; }
protected UserManager<AuthUser> UserManager { get; }

public RoleController(RoleManager<IdentityRole> roleManager,
    UserManager<AuthUser> userManager)
{
    RoleManager = roleManager;
    UserManager = userManager;
}
```

Teraz możemy dodać widok do akcji Index:



ZARZĄDZANIE ROLAMI

W celu chwilowego składowania danych dotyczących ról będziemy potrzebowali klasy modelu np. RoleInfo z kodem:

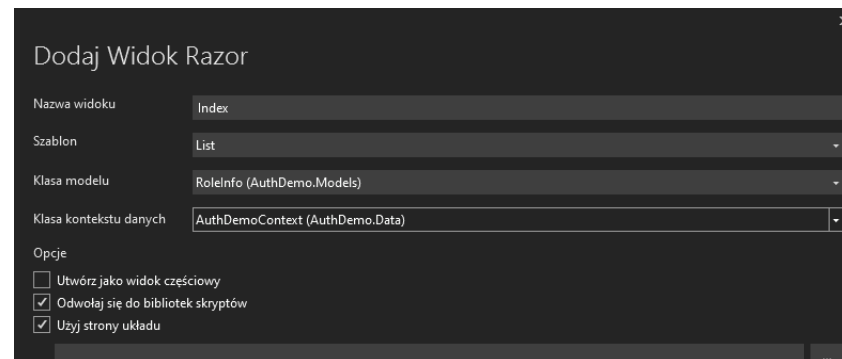
```
public class RoleInfo
{
    public string Id { get; set; }
    public string Name { get; set; }
}
```

Do kontrolera ról wygenerowanego w kroku z poprzedniego slajdu możemy teraz wstrzyknąć menagera użytkowników i ról:

```
protected RoleManager<IdentityRole> RoleManager { get; }
protected UserManager<AuthDemoUser> UserManager { get; }

public RoleController(RoleManager<IdentityRole> roleManager,
    UserManager<AuthDemoUser> userManager)
{
    RoleManager = roleManager;
    UserManager = userManager;
}
```

Teraz możemy dodać widok listowania do akcji Index:



Dodaj Widok Razor

Nazwa widoku: Index

Szablon: List

Klasa modelu: RoleInfo (AuthDemo.Models)

Klasa kontekstu danych: AuthDemoContext (AuthDemo.Data)

Opcje

- ☐ Utwórz jako widok częściowy
- ☒ Odwołaj się do bibliotek skryptów
- ☒ Użyj strony układu

ZARZĄDZANIE ROLAMI

Wypełniając widok przy użyciu managera ról np. tak:

```
public ActionResult Index()
{
    var roles = RoleManager.Roles
        .Select(r =>
            new RoleInfo { Name = r.Name, Id = r.Id }).ToList();
    return View(roles);
}
```

Podobnie tworzymy widok dodawania roli (wybierając szablon Create zamiast List). W widoku usuwamy grupę formularza odpowiadającą za wprowadzanie właściwości Id. A zamiast niej dodajmy grupę formularza odpowiadającą za użytkowników podpiętych pod rolę. Np.:

```
<h4>RoleInfo</h4>
<hr />
<div class="row">
  <div class="col-md-4">
    <form asp-action="Create">
      <div asp-validation-summary="ModelOnly" class="text-danger"></div>
      <div class="form-group">
        <label asp-for="Name" class="control-label"></label>
        <input asp-for="Name" class="form-control" />
        <span asp-validation-for="Name" class="text-danger"></span>
      </div>
      <div class="form-group">
        <label class="control-label">Users</label>
        <input id="Users" name="Users" class="form-control" />
      </div>
      <div class="form-group">
        <input type="submit" value="Create" class="btn btn-primary" />
      </div>
    </form>
  </div>
</div>
```

ZARZĄDZANIE ROLAMI

Uzupełnij akcję Create (wersję określoną atrybutem Post) np. w następujący sposób:

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Create(RoleInfo roleInfo, IFormCollection rest)
{
    try
    {
        RoleManager.CreateAsync(new IdentityRole {
            Name = roleInfo.Name }).Wait();
        string[] usersNames = rest["Users"].First().Split(' ');
        foreach (string userName in usersNames)
        {
            var userTask = UserManager.FindByNameAsync(userName);
            userTask.Wait();
            var user = userTask.Result;
            UserManager.AddToRoleAsync(user, roleInfo.Name).Wait();
        }
        return RedirectToAction(nameof(Index));
    }
    catch
    {
        return View();
    }
}
```

Następnie spróbuj dodać rolę Admin do puli użytkowników której dorzucisz utworzonego wcześniej użytkownika (w ramach pola Users podaj e-mail użytkownika).

OGRANICZENIE DOSTĘPU DO AKCJI PRZY UŻYCIU RÓL

W celu wykorzystania ról do ograniczenia dostępu atrybut `Authorize` pozwala na zdefiniowanie właściwości `Roles`, w ramach której przekazujemy oddzielone przecinkiem role pozwalające korzystać z danej akcji/kontrolera.

```
public class HomeController : Controller
{
    private readonly ILogger<HomeController> _logger;

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

    public IActionResult Index()
    {
        return View();
    }

    [Authorize(Roles = "Superadmin,Admin")]
    public IActionResult Privacy()
    {
        return View();
    }
}
```

ZASADY (POLITYKI)

Alternatywną formą ograniczania dostępu jest określenie zasad (polityk), którymi należy się kierować w celu określenia czy dany użytkownik ma uprawnienia do wykonywania określonych akcji. Zasady opiera się na poświadczeniach (ang. claims), zebranych dla danego użytkownika w ramach managera użytkowników. Zasady można zadeklarować w ramach pliku Program.cs: np. tak:

```
builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("CanDrive",
        policy => policy.RequireClaim("DriverLicense"));
    options.AddPolicy("Over18",
        policy => policy.RequireAssertion(context =>
        {
            var claim = context.User.FindFirst("Age");
            return int.Parse(claim?.Value??"0") >= 18;
        }));
});
```

W przykładzie zadeklarowaliśmy dwie zasady "CanDrive" oraz "Over18", które wymagają odpowiednio posiadania poświadczenia w postaci prawa jazdy w zbiorze uprawnień danego użytkownika oraz poświadczenia wieku, które dodatkowo wskazuje na wartość całkowitoliczbową równą przynajmniej 18.

OGRANICZANIE DOSTĘPU ZA POMOCĄ ZASAD

Podobnie jak w przypadku ról polityki mogą ograniczać dostęp do kontrolerów lub akcji poprzez zdefiniowanie w ramach atrybutu `Authorize` właściwości `Policy` określającej, którą spośród zasad należy zastosować w celu ograniczenia dostępu.

```
public class HomeController : Controller
{
    private readonly ILogger<HomeController> _logger;

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

    public IActionResult Index()
    {
        return View();
    }

    [Authorize(Policy = "CanDrive")]
    public IActionResult Privacy()
    {
        return View();
    }
}
```

NADAWANIE POŚWIADCZEŃ

Poświadczenia (claims) nadawane są podobnie jak pozostałe elementy dotyczące zarządzania użytkownikami i rolami z poziomu menagera użytkowników i ról. Przykładowy kod poświadczający nadanie prawa jazdy danemu użytkownikowi w ramach akcji kontrolera znajduje się poniżej:

```
public class DriversLicenseCertificationUnitController : Controller
{
    public UserManager<AuthUser> UserManager { get; }

    public DriversLicenseCertificationUnitController(UserManager<AuthUser> userManager)
    {
        UserManager = userManager;
    }

    [Authorize]
    public IActionResult Apply()
    {
        var claim = HttpContext.User.FindFirst(ClaimTypes.NameIdentifier);
        var userTask = UserManager.FindByIdAsync(claim.Value);
        userTask.Wait();
        var user = userTask.Result;
        UserManager.AddClaimAsync(user, new Claim("DriverLicense", "123-123")).Wait();

        return View();
    }
}
```