European Funds
Knowledge Education Development
Wrocław University of Science and Technology
European Union
European Social Fund

ZPR PWr – Zintegrowany Program Rozwoju Politechniki Wrocławskiej

Aplikacje webowe na platformę .NET

W12 – Identity, konfigurowanie aplikacji, potok przetwarzania

Syllabus

- Autentykacja, autoryzacja
- Uwierzytelnianie w ASP .NET Core
- ASP .Net Core **Identity**
- Autoryzacja oparta na rolach
- Polityki autoryzacji

- Konfigurowanie aplikacji ASP
  - Konfigurowanie projektu
  - Klasa `Program`
  - Klasa `Startup`
- Potok przetwarzania
- Oprogramowanie pośredniczące

Autentykacja, Autoryzacja

- Autentykacja/Uwierzytelnienie: To jest weryfikacja czy użytkownik jest tym, za którego się podaje.
  - Czyli problemy tutaj to "Kto to jest?" i "Jak sprawdzić, że to ta osoba?„
  - Rozwiązywana najczęściej przez parę użytkownik-hasło oraz sesja (ciasteczko dla sesji)
- Autoryzacja: To jest weryfikacja czy użytkownik ma prawo dostępu do konkretnych usług / zasobów.
  - Pytania to "Czy userX może przeczytać Y?", "Czy userX może zmienić Z?„
  - Najczęściej przez nadanie roli użytkownikowi (admin, magazynier, kierownik itp.) i sprawdzenie dla operacji, czy dana rola ma do tej operacji dostęp

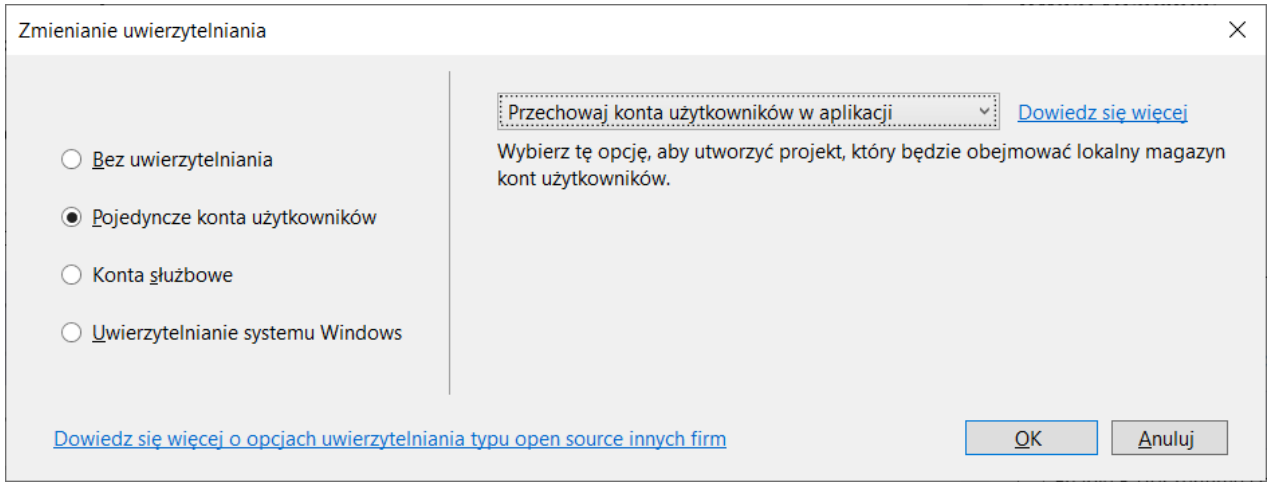
ASP .Net Core Identity

Uwierzytelnianie w ASP .NET Core

- Za pomocą kont zapamiętanych w bazie danych aplikacji
- Za pomocą kont zapamiętanych w chmurze
- Za pomocą kont służbowych:
  - W usłudze Active Directory
  - W usłudze MS Azure Active Directory
  - W usłudze Office 365
- Za pomocą uwierzytelniania Windows.
- Za pomocą uwierzytelniania open source
- Za pomocą kont Facebook, Google i in.
- Za pomocą każdego dowolnego z powyższych.

Można też napisać wszystko od początku, jednak zastosowanie powyższych sposobów pozwala na użycie:

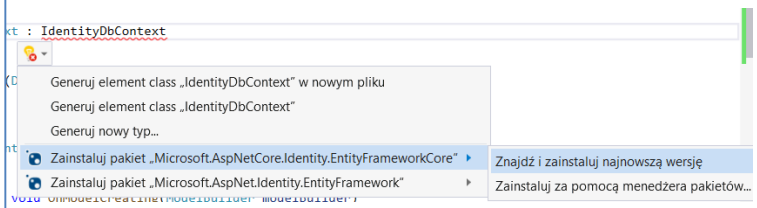
- Gotowych klas do dostępu do konta, roli itp.
- Gotowych klas do zarządzania kontami, rolami itd.
- Adnotacji ułatwiających autoryzację
- Gotowych stron w Razorze do zakładania konta, logowania, modyfikacji danych, odzyskiwania hasła itp.



ASP .Net Core Identity

- ASP .Net Core Identity:
  - Jest interfejsem API obsługującym funkcję logowania w interfejsie użytkownika (UI).
  - Zarządza użytkownikami, hasłami, danymi profilu, rolami, poświadczeniami, tokenami, potwierdzeniami e-mail i nie tylko.
- Użytkownicy mogą utworzyć konto z danymi logowania przechowywanymi w Identity lub mogą korzystać z zewnętrznego dostawcy logowania. Obsługiwani zewnętrzni dostawcy logowania obejmują Facebook, Google, konto Microsoft i Twitter.
- Założenie: klasa `Student` oraz projekt stworzony od początku z możliwością uwierzytelniania (oparty o bazę z EntityFramework)
  - Z wszystkimi krokami do działania z EntityFramework
  - Z kontrolerem i widokami za pomocą generatora kodów dla klasy `Student` w kontekście bazy danych EntityFramework
  - Główna różnica – kontekst bazy danych dziedziczy po `IdentityDbContext` zamiast po `DbContext`

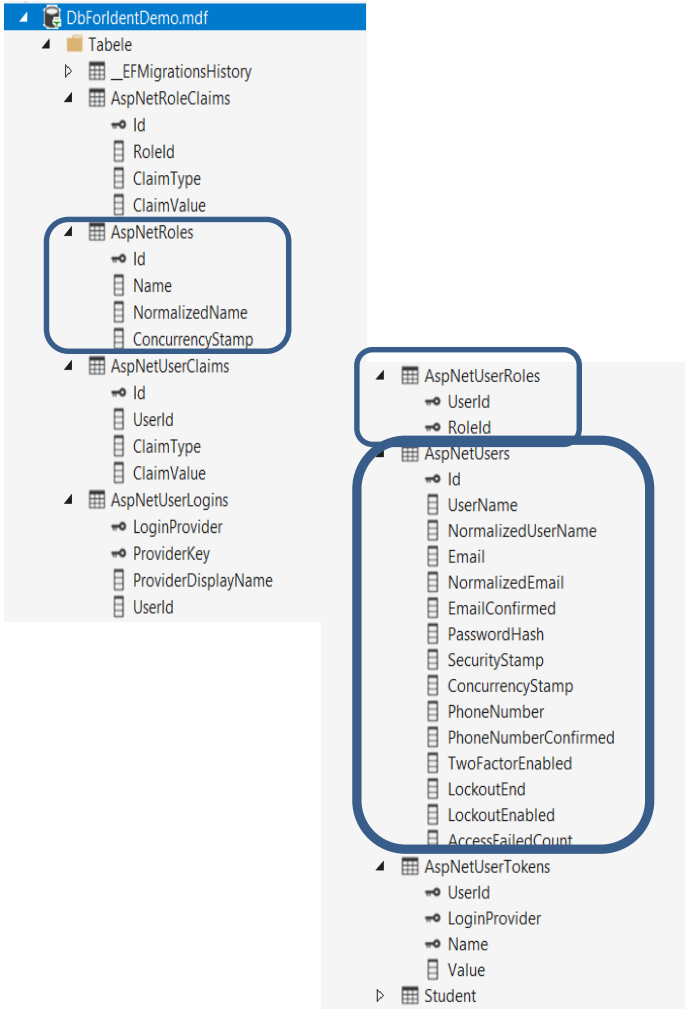
```
public class MyDbContext : IdentityDbContext
{
    Odwołania: 0
    public MyDbContext(DbContextOptions<MyDbContext> options) : base(options)
    {
    }
    Odwołania: 7
    public DbSet<Student> Student { get; set; }

    Odwołania: 0
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder); // create tables for Identity
        modelBuilder.Seed();
    }
}
```



`WebAppStudentWithIdentity`

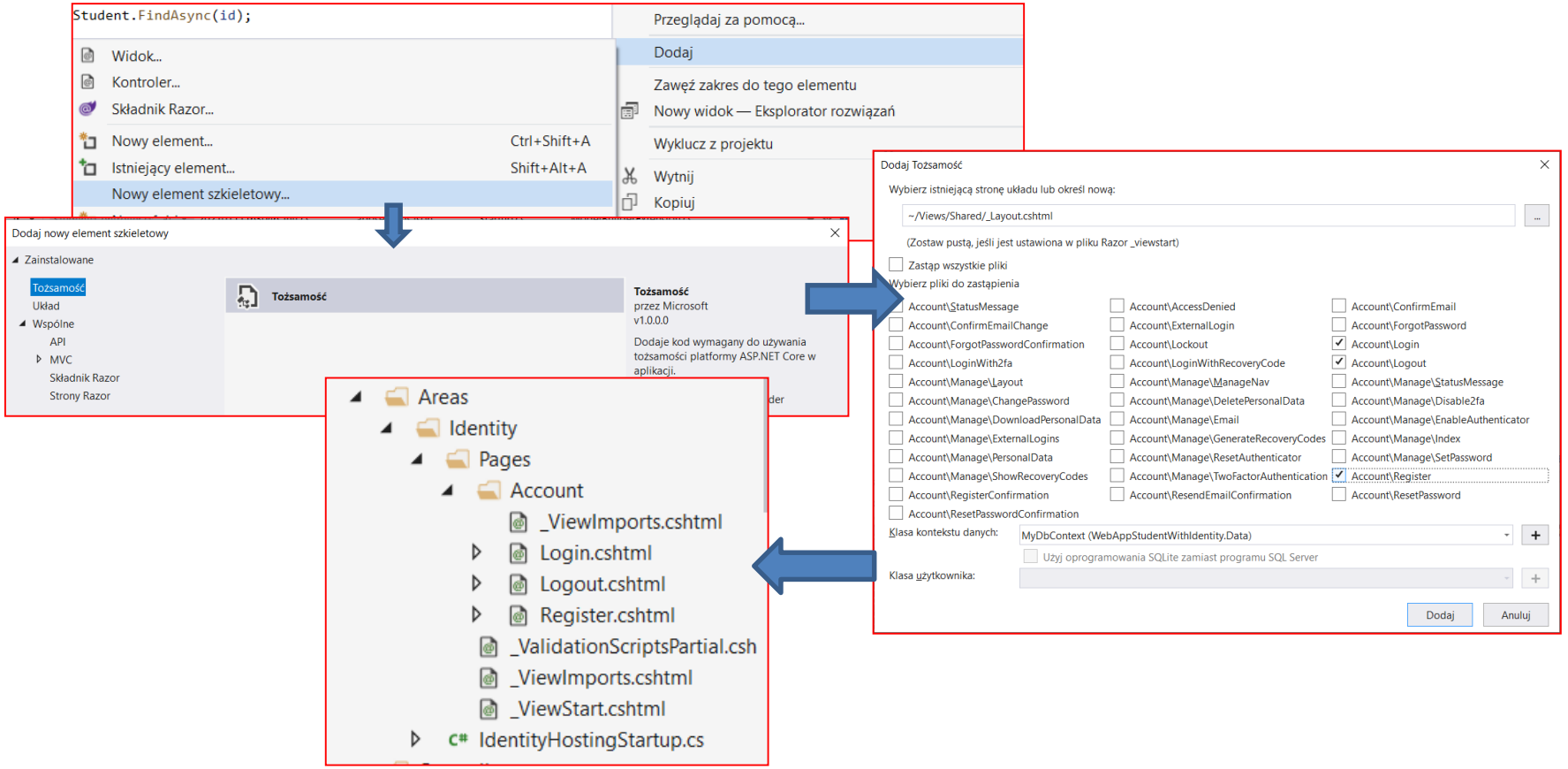
Nowe tabele w bazie

- Po stworzeniu bazy powstaną odpowiednie tabele
- Tabela `AspNetUsers` zawiera dane o użytkownikach (w tym identyfikator i hasło).
- Dwie tabele służą do autoryzacji oparte na rolach (ang. **Role-Based Security**)
  - `AspNetRoles` – role jakie może posiadać użytkownik
  - `AspNetUserRoles` – którzy użytkownicy jakie mają przydzielone role
- Tabele ze słowem `Claim` służą do autoryzacji bazującej na roszczeniach/oświadczeniach (ang. **Claims-Based Security**). Aby ich użyć trzeba stworzyć polityki (policy) autoryzacji umieszczone w kodzie C#
  - Bardziej elastyczne
  - Bardziej czasochłonne
- Pozostałe tabele są do pamiętania danych do autoryzacji z innych źródeł (`AspNetUserLogins`) lub autoryzacją oparta na tokenach (`AspNetUserTokens`)



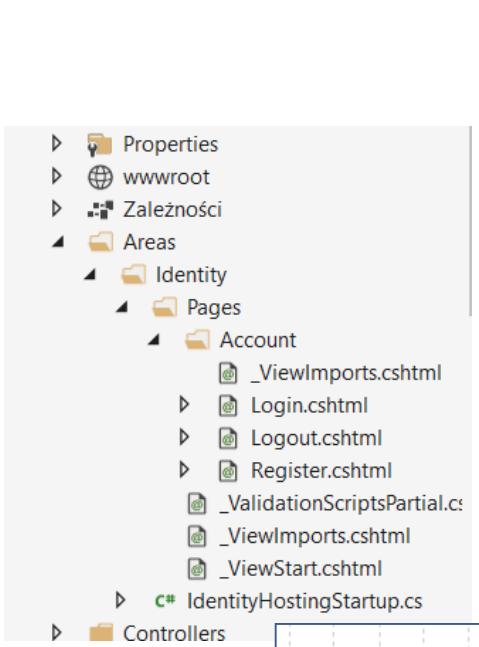
Strony Razora dla `Identity`

- Strony do operacji związanych z zarządzaniem kontami, rolami itd. są to **strony** Razora. Nie są one tworzone samoistnie, gdyż można je też napisać od zera.
- Aby skorzystać z gotowych wzorców stron należy:
  - PPM na projekcie -> Dodaj -> Dodaj nowy element szkieletowy
  - Tożsamość ->Tożsamość ->Dodaj
  - Jeśli chcemy dodać do istniejącego layout-u to wpisujemy np. `~/Views/Shared/_Layout.cshtml`
  - Jako kontekst można stworzyć nowy lub podać używany np. `MyUsersDbContext`
- Pojawi się nowy folder `Area/Identity` z wybranymi stronami
- Nie powoduje to zmiany układu strony (layout), chociaż w folderze `Shared` przygotowany jest widok częściowy do użycia.



Zmiany w plikach

- Dodać w serwisach, że identyfikacja będzie na podstawie wytworzonej bazy EntityFramework i będziemy używać ról (menadżera ról).
- Wstawić do strumienia oprogramowania pośredniczącego obsługę autoryzacji.
- Dodać obsługę routingu dla stron Razora:
  - Sposób realizacji tworzonych generatorem stron logowania.
- Dodanie do layout-u widoku częściowego do logowania




```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews();
    services.AddDbContextPool<MyDbContext>(options =>
            options.UseSqlServer(Configuration.GetConnectionString("MyDb")));
    services.AddDefaultIdentity<IdentityUser>(options => options.SignIn.RequireConfirmedAccount = false)
            .AddRoles<IdentityRole>()
            .AddEntityFrameworkStores<MyDbContext>();
}
```

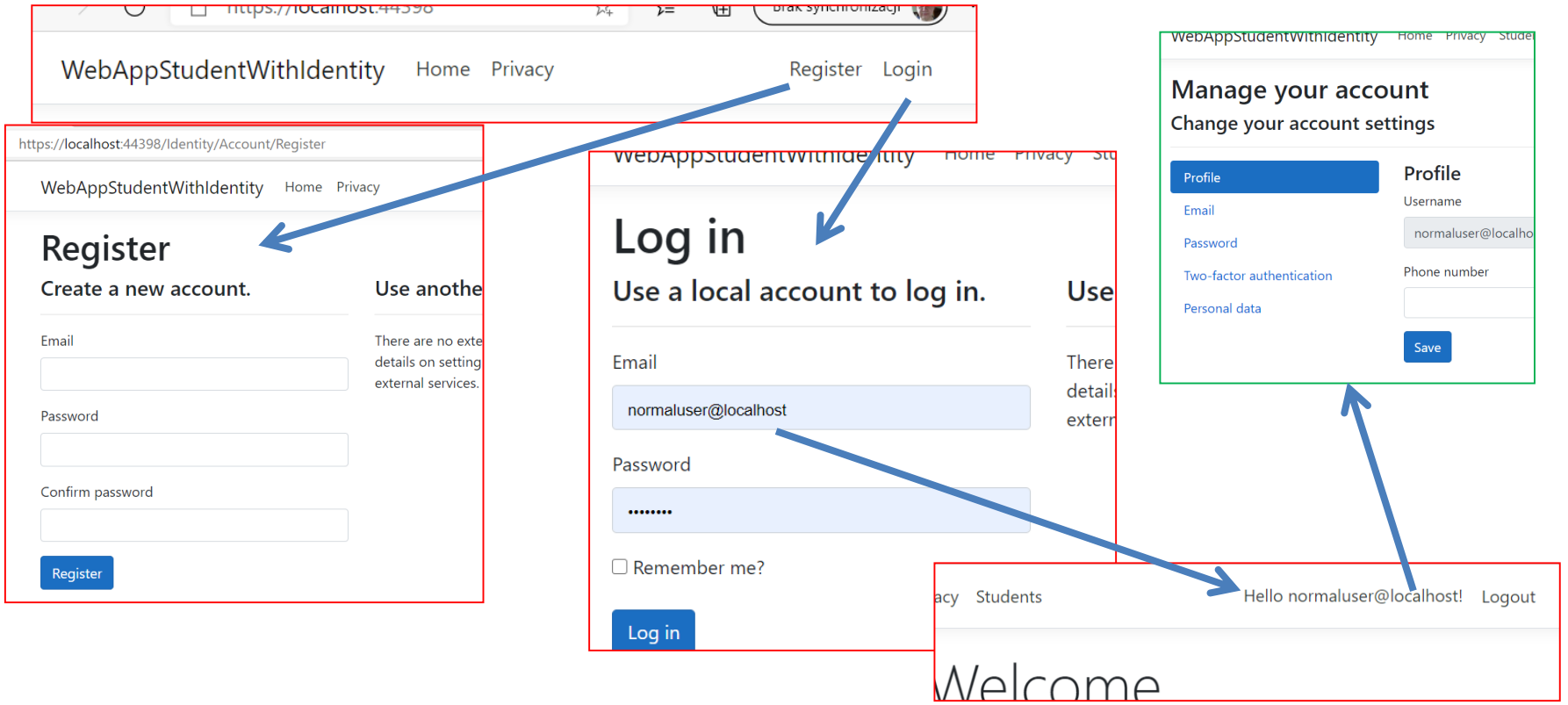
```
app.UseAuthentication();

app.UseAuthorization();

MyIdentityDataInitializer.SeedData(userManager, roleManager);

app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "default",
        pattern: "{controller=Home}/{action=Index}/{id?}");
    endpoints.MapRazorPages();
});
```

later

```
                <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="Privacy">
            </li>
        </ul>
    </div>
    <div class="navbar-collapse collapse d-sm-inline-flex flex-sm-row-reverse">
        <partial name="_LoginPartial" />
    </div>
</nav>
```

Działanie

- Użytkownicy mogą się zarejestrować i logować
  - Ale domyślnie jest potrzeba potwierdzenia zakładania konta przez podany email (wyłączona opcją na poprzednim slajdzie)
  - Są wymagania na poprawne, trudne hasło (też można zmienić)
  - Użytkownik nie ma roli
- Po zalogowaniu działa panel użytkownika (PPM na nazwę użytkownika):
  - Istnieją domyślne strony Razora (mimo, że ich nie dodaliśmy do obszaru)



Autoryzacja oparta na rolach

Autoryzacja oparta na rolach

- Ustalić nazwy ról jakie mogą występować w aplikacji.
- Ustalić związane z nimi uprawnienia.
- Oznaczyć kontrolery lub poszczególne akcje adnotacjami pozwalającymi/zabraniającymi korzystania z danej akcji przez określone role/użytkowników.
- Użyć adnotacji dla akcji dopuszczalnych dla nie/zalogowanych użytkowników
- Role można sprawdzić w ramach kody C#/Razor i na ich podstawie zaprezentować inny widok (widok częściowy) lub jego fragment.
- Warto na początku „zasiać” bazę danych rolami oraz kontem administracyjnym.
- Domyślny szkielet aplikacji z autoryzacją pozwala każdemu założyć konto (w niektórych zastosowaniach nie jest to pożądana funkcjonalność)
- Należy w wielu miejscach kodu dodać zespół `Microsoft.AspNetCore.Authorization`;
- Zmienić kod w `Startup.ConfigureServices()`, aby używać ról.

- Analogicznie narzędzia (adnotacje/klasy i metody) dostępne są dla autoryzacji opartej o politykach (ang. policy)

Adnotacje dla autoryzacji

- Adnotacje mogą się odnosić do **kontrolera** lub do **akcji**.
- Adnotacja przy akcji **dokłada** regułę do adnotacji przy kontrolerze.
  - Wyjątek: `[AllowAnonymous]`
- Najczęstsze adnotacje:
  - `[AllowAnonymous]`
    - Dostęp możliwy dla wszystkich. Domyślny, jeśli nie ma żadnych innych adnotacji dla autoryzacji. Przy akcji odwołuje ograniczenia nałożone przez adnotacje przy kontrolerze/akcji.
  - `[Authorize]`
    - Dostęp dla zalogowanych użytkowników
  - `[Authorize(Roles = "Admin, PowerUser")]`
    - jeśli chcemy, aby użytkownik musiał mieć dowolną z ról, oddzielamy je przecinkiem
  - `[Authorize(Roles = "Admin")]`
    `[Authorize(Roles = "PowerUser")]`
    - Jeśli musi posiadać kilka ról, aby móc wykonać akcję.
- Podczas wykorzystywania polityk autoryzacji stosuje się
  - `[Authorize(Policy = "RequireRoleForTurnOnOff")]`
- Dla przykładów należy stworzyć role i użytkowników…

Autoryzacja – zasianie danych

- Zasianie danymi
  - Można uruchomić w kontekście w ramach metody `OnModelCreate()`
  - Lepiej w `Startup.Configure(…)` z wstrzykniętymi parametrami będącymi referencjami do manadżerów.
- W przypadku pracy na użytkownikach i rolach warto użyć odpowiednich manadżerów
  - Stworzą identyfikatory
  - Sprawdzą zgodność z regułami hasła
  - I in.

Startup.cs

```
public void ConfigureServices(IServiceCollection services)
{
...
  services.AddDefaultIdentity<IdentityUser>()
                .AddRoles<IdentityRole>()
                .AddEntityFrameworkStores<ApplicationDbContext>();
...
}
```

Startup.cs

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env,
            UserManager<IdentityUser> userManager, RoleManager<IdentityRole> roleManager)
{
...
  app.UseAuthentication();
  app.UseAuthorization();

  MyIdentityDataInitializer.SeedData(userManager, roleManager);
...
}
```

Inicjalizacja danych do autoryzacji 1/2

Data/MyIdentityDataInitializer

```
public class MyIdentityDataInitializer
{
  public static void SeedData(UserManager<IdentityUser> userManager,
                              RoleManager<IdentityRole> roleManager)
  {
    SeedRoles(roleManager);
    SeedUsers(userManager);
  }
        // name - poprawny adres email
        // password - min 8 znaków, mała i duża litera, cyfra i znak specjalny
  public static void SeedRoles(RoleManager<IdentityRole> roleManager)
  {
    if (!roleManager.RoleExistsAsync("Admin").Result)
    {
      IdentityRole role = new IdentityRole
      {
        Name = "Admin",
      };
      IdentityResult roleResult = roleManager.CreateAsync(role).Result;
    }
    if (!roleManager.RoleExistsAsync(„Dean").Result)
    {
      IdentityRole role = new IdentityRole
      {
        Name = "Dean",
      };
      IdentityResult roleResult = roleManager.CreateAsync(role).Result;
    }
  }
```

Inicjalizacja danych do autoryzacji 2/2

Data/MyIdentityDataInitializer

```
  public static void SeedOneUser(UserManager<IdentityUser> userManager,
    string name, string password, string role = null)
  {
    if (userManager.FindByNameAsync(name).Result == null)
    {
      IdentityUser user = new IdentityUser
      {
        UserName = name, // musi być taki sam jak email, inaczej nie zadziała
        Email = name
      };
      IdentityResult result = userManager.CreateAsync(user, password).Result;
      if (result.Succeeded && role != null)
      {
        userManager.AddToRoleAsync(user, role).Wait();
      }
    }
  }
  public static void SeedUsers(UserManager<IdentityUser> userManager)
  {
    SeedOneUser(userManager, "normaluser@localhost", "nUpass1!");
    SeedOneUser(userManager, "adminuser@localhost", "aUpass1!", "Admin");
    SeedOneUser(userManager, „deanuser@localhost", „dUpass1!", „Dean");
  }
}
```

Autoryzacja - przykład

- Adnotacje dla kontrolera i akcji

Controllers/HomeController.cs

```
public class HomeController : Controller
{
...
 [Authorize(Roles = "Admin")]
  public IActionResult ForAdmin()
  {  ViewData["Info"] = "For Admin„;  return View("Info");  }
  [AllowAnonymous]
  public IActionResult ForAll()
  {  ViewData["Info"] = "For All";  return View("Info");  }
  [Authorize]
  public IActionResult ForLogIn()
  {  ViewData["Info"] = "For Log In";  return View("Info");  }
  [Authorize(Roles = "Admin, Dean")]
  public IActionResult ForAdminOrDean()
  {  ViewData["Info"] = "For Admin or Dean";  return View("Info");  }
...
}
```

Controllers/AdminController.cs

```
[Authorize(Roles = "Admin")]
public class AdminController : Controller
{
...
 [AllowAnonymous]
  public IActionResult Index()
  {  ViewData["Info"] = "AdminController -> For All";  return View("Info");  }
  [Authorize(Roles ="Dean")]
  public IActionResult ForDean()
  {  ViewData["Info"] = "AdminController -> For (Admin and Dean)";  return View("Info");  }
  public IActionResult ForAdmin()
  {  ViewData["Info"] = "AdminController -> For Admin”;  return View("Info");  }
...
}
```

Autoryzacja – strony Razora

- Korzystanie z ról w kodzie Razor-a:
  - strona częściowa `_LoginPartial.cshtml`
  - zmiana menu w `_Layout.cshtml`

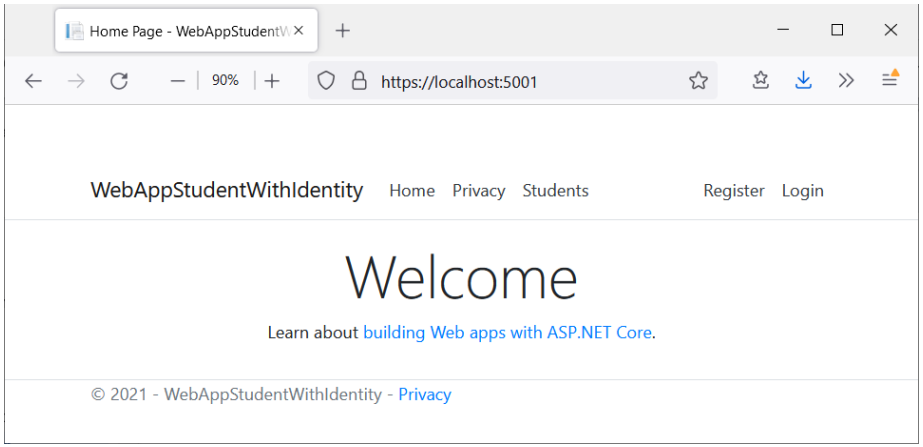
```
@using Microsoft.AspNetCore.Identity
@inject SignInManager<IdentityUser> SignInManager
@inject UserManager<IdentityUser> UserManager

<ul class="navbar-nav">
@if (SignInManager.IsSignedIn(User))
{
    <li class="nav-item">
        <a  class="nav-link text-dark" asp-area="Identity" asp-page="/Account/Manage/Index" title="Manage">Hello @User.Identity.Name!</a>
    </li>
    <li class="nav-item">
        <form  class="form-inline" asp-area="Identity" asp-page="/Account/Logout" asp-route-returnUrl="@Url.Action("Index", "Home", new { area = "" })">
            <button  type="submit" class="nav-link btn btn-link text-dark">Logout</button>
        </form>
    </li>
}
else
{ ... }
```

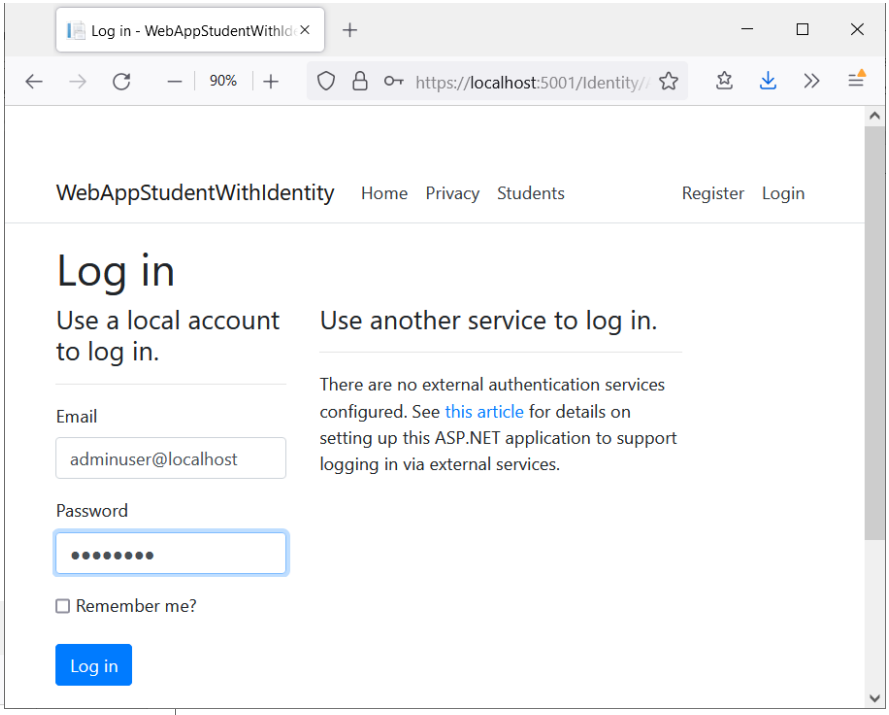
```
<li class="nav-item">
<a class="nav-link text-dark" asp-area="" asp-controller=„Students" asp-action=„Index">Students</a>
</li>
@if (User.IsInRole("Admin"))
{
  <li class="nav-item">
  <a class="nav-link text-dark" asp-area="" asp-controller="Admin" asp-action=„ForAdmin">For admin</a>
  </li>
}
```

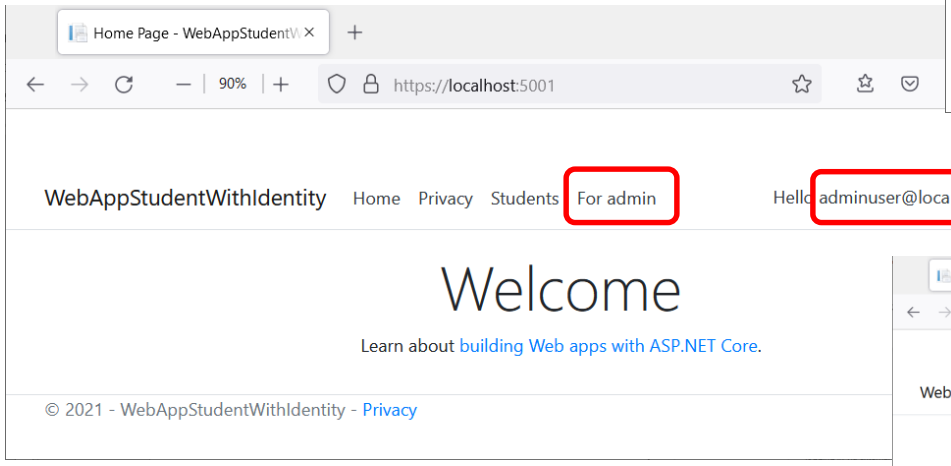
Autoryzacja – scenariusze użycia

- Zalogowanie na zwykłego użytkownika
- Zalogowanie z rolą dziekana („Dean”)
- Zalogowanie z rolą administrator
- Obserwacja zmiany menu
- Dostęp do stron bez zalogowania
  - Próba dostępu poprzez wpisanie nazwy akcji w pasku adresu – przekierowanie do logowania

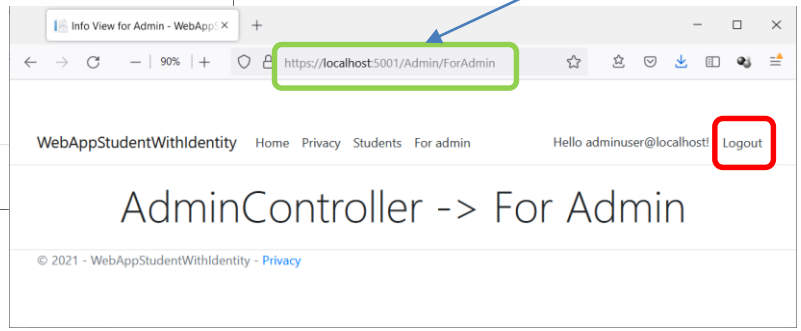
Logowanie z rolą „Admin”



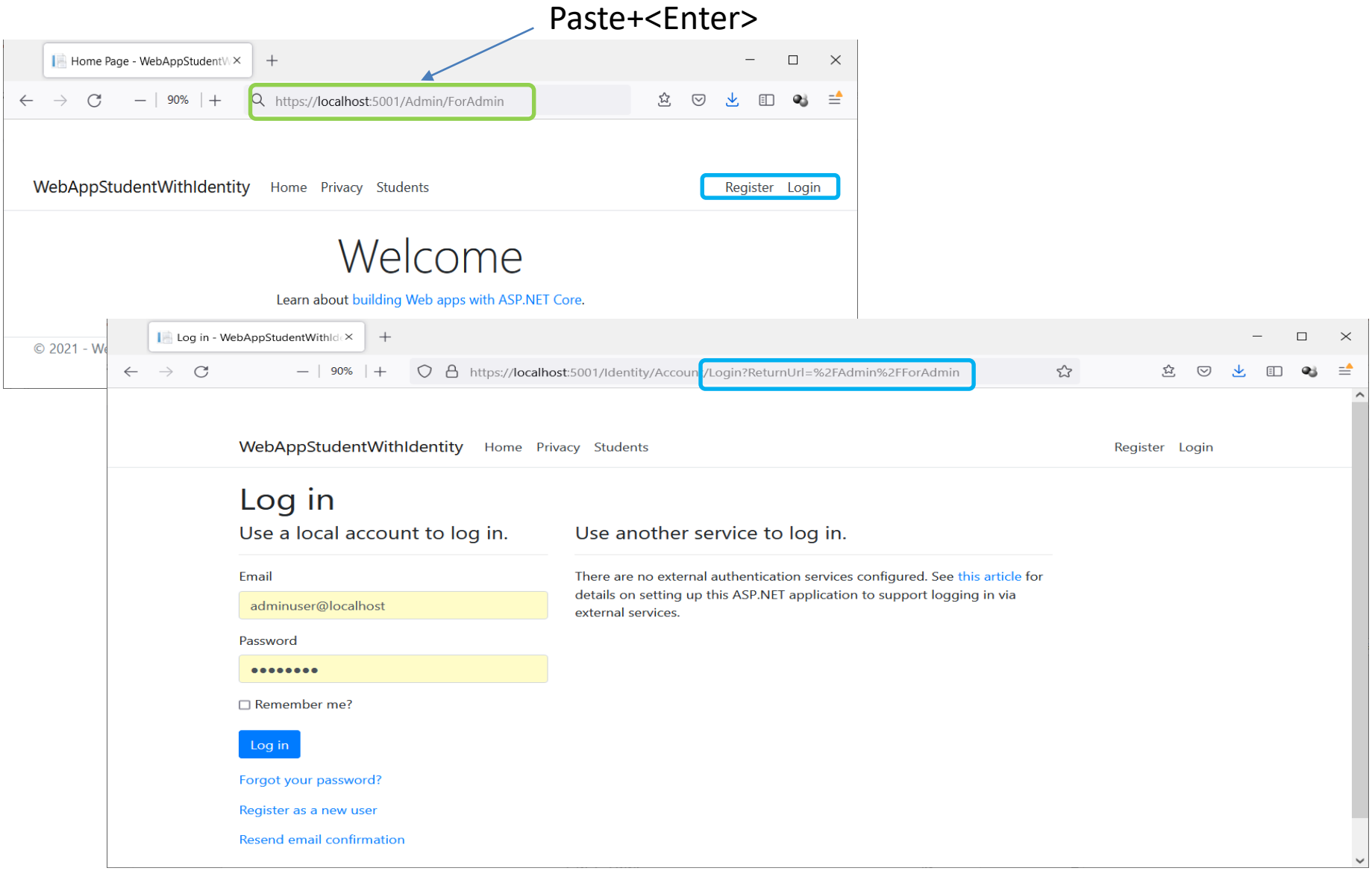




Copy



Przekierowanie po zalogowaniu



Przykład tworzenia polityk autoryzacji

- Tworzenie polityk autoryzacji jest bardziej elastyczne – można tworzyć reguły z użycie ról, roszczeń (`Claim`), tokenów itd.
- Pozwalają też dodać kolejny poziom elastyczności podczas programowania lub projektowania.
- Przykład (bez rzeczywistej implementacji) poniżej:
  - Zmiana definicji polityki bez potrzeby zmian adnotacji

Startup.cs

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddControllersWithViews();
  services.AddRazorPages();
  services.AddAuthorization(options => {
     options.AddPolicy("RequireRoleForTurnOnOff", policy =>
         policy.RequireRole("Administrator")); });
}
```

Change

XController.cs

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

Startup.cs

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddControllersWithViews();
  services.AddRazorPages();
  services.AddAuthorization(options => {
     options.AddPolicy("RequireRoleForTurnOnOff", policy =>
         policy.RequireRole("Administrator", "PowerUser", "BackupAdministrator")); });
}
```

Dodawanie Identity do istniejącego projektu

Dodawanie identyfikacji **do istniejącego** projektu:

- Zmiana dziedziczenia kontekstu
  - `public class GameDbContext : IdentityDbContext`
- Dodanie migracji w konsoli pakietów:
  - `add-migration AddIdentity`
- Uaktualnienie bazy danych
- Dodawanie stron Razora:
  - PPM na projekcie -> Dodaj -> Dodaj nowy element szkieletowy
  - Tożsamość ->Tożsamość ->Dodaj
  - Jeśli chcemy dodać do istniejącego layout-u to wpisujemy np. `~/Views/Shared/_Layout.cshtml`
- Stworzenie (skopiowanie z innego projektu?) `~/Views/Shared/_LoginPartial.cshtml`
- Zmodyfikowanie `~/Views/Shared/_Layout.cshtml`, aby korzystał z powyższej strony częściowej.
- Dodanie (gdzie potrzebujemy) korzystania z zespołów `Microsoft.AspNetCore.Identity`, `Microsoft.AspNetCore.Identity.EntityFrameworkCore` itp.
- Dodanie w serwisach użycie identyfikacji i autoryzacji:

```
 services.AddDefaultIdentity<IdentityUser>()
.AddRoles<IdentityRole>()
.AddEntityFrameworkStores<GameDbContext>();
```

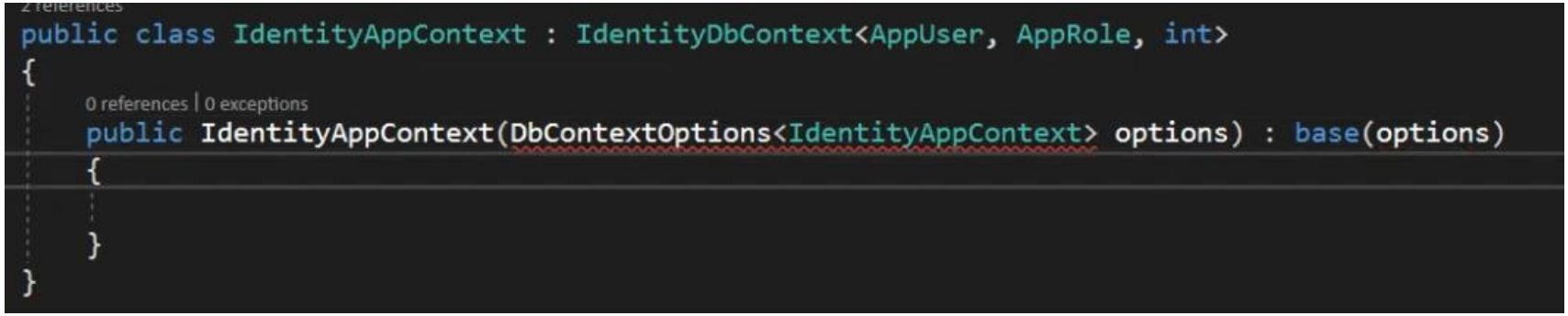
- Dodanie w konfiguracji potoku oprogramowania pośredniczącego:

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

- Dodanie mapowania dla stron Razora

Zawartość pakietu `Identity`

- Wybrane dodatkowe funkcjonalności:
  - Operacje CRUD na kontach użytkowników
  - Potwierdzanie poprawności konta
  - Odzyskiwanie hasła
  - Dwuetapowa autentykacja SMS-em
  - itd.
- Bardzo elastyczne rozwiązanie. Można:
  - Stworzyć własną klasę użytkownika dziedzicząc po IdentityUser (można też bez dziedziczenia)
  - Stworzyć własną klasę ról dziedzicząc po IdentityRole (można też bez dziedziczenia)
  - Podać typ klucza użytkownika
  - Stworzyć własne menadżery użytkowników i ról



```
2 references
public class IdentityAppContext : IdentityDbContext<AppUser, AppRole, int>
{
    0 references | 0 exceptions
    public IdentityAppContext(DbContextOptions<IdentityAppContext> options) : base(options)
    {

    }
}
```

Claim and Token

Krótko-idea:

- Inne sposoby podejścia do autoryzacji
- Rola to tylko nazwa (string)
  - Użytkownik posiada rolę lub nie, rodzaj wartości logicznej.
- Claims to jakby słownik z cechami dla użytkownika
  - Cechy mogą mieć wartość (np. stan konta w banku, poziom zaufania itp.)
- Token – jakby ciasteczka dla autoryzacji

Dodatek

- Jak skonfigurować opcje aby nie wprowadzać za dużo wymagań na hasło

```
public class IdentityHostingStartup : IHostingStartup
{
    public void Configure(IWebHostBuilder builder)
    {
        builder.ConfigureServices((context, services) => {
            services.AddDbContext<AuthDbContext>(options =>
                options.UseSqlServer(
                    context.Configuration.GetConnectionString("AuthDbContextConnection")));

            services.AddDefaultIdentity<ApplicationUser>(options =>
            {
                options.SignIn.RequireConfirmedAccount = false;
                options.Password.RequireLowercase = false;
                options.Password.RequireUppercase = false;
            })
                .AddEntityFrameworkStores<AuthDbContext>();
        });
    }
```

Konfigurowanie projektu, część 1

Konfigurowanie projektu

- Konfiguracja projektu znajduje się w pliku `<nazwaProjektu>.csproj` i jest ukryta w drzewie pokazywanym w eksploratorze rozwiązania
  - PPM na nazwie projektu i „Edytuj plik projektu”
- Zawartość i potrzeba uwzględnienia różnych sekcji zależy od wersji platformy

WebAppMinMVC21

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.App" />
    <PackageReference Include="Microsoft.AspNetCore.Razor.Design" Version="2.1.2" PrivateAssets="All" />
  </ItemGroup>

</Project>
```

WebAppMinMVC31

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>
  </PropertyGroup>

</Project>
```

Plik konfiguracyjny

- Używana jest notacja XML.
  - Podobna do znaczników HTML
- `<Project>` - główny element konfiguracji. Atrybut `Sdk` wskazuje na minimalny zbiór poleceń importujących
- `<PropertyGroup>` - grupuje właściwości konfiguracyjne, aby dodać strukturę do pliku
- `<TargetFramework>` - określa docelową wersję frameworku .Net dla procesu kompilacji. Może być zamieniony na `<TargetFrameworks>`, w którym po przecinku podawane jest kilka frameworków.

Mogą wystąpić również po dodaniu pakietów:

- `<ItemGroup>` – dodaje grupy powiązanych elementów konfiguracyjnych aby dodać je do struktury plików
- `<PackageReference>` - zdefiniowanie zależności do pakietu NuGet

```
<ItemGroup>
    <PackageReference Include="Microsoft.EntityFrameworkCore" Version="3.1.7" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Sqlite" Version="3.1.6" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer" Version="3.1.7" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="3.1.7">
      <PrivateAssets>all</PrivateAssets>
      <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
    </PackageReference>
    <PackageReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Design" Version="3.1.4" />
</ItemGroup>
```

Klasa `Program`

- Standardowa zawartość tej klasy (Core 5.0):

```
public class Program
{
  public static void Main(string[] args)
  {
    CreateWebHostBuilder(args).Build().Run();
  }

  public static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureWebHostDefaults(webBuilder =>
                {
                    webBuilder.UseStartup<Startup>();
                });
  }
}
```

- Kolejny slajd: zamiana `CreateDefaultBuilder` na „ręczne” wykonanie kolejnych kroków konfiguracyjnych.
  - Ale dla wersji Core 2.1
  - Ważna jest kolejność wywołań metod, gdyż kolejny krok może korzystać z poprzednich kroków.

`CreateDefaultBuilder` – kod zastępczy (Core 2.1)

```
public static IWebHost BuildWebHost(string[] args)
{
  return new WebHostBuilder()
             .UseKestrel()
             .UseContentRoot(Directory.GetCurrentDirectory())
             .ConfigureAppConfiguration((hostingContext, config) => {
                var env = hostingContext.HostingEnvironment;
                config.AddJsonFile("appsettings.json",
                     optional: true, reloadOnChange: true)
                  .AddJsonFile($"appsettings.{env.EnvironmentName}.json",
                            optional: true, reloadOnChange: true);
                  config.AddEnvironmentVariables();
                  if (args != null)
                  {
                      config.AddCommandLine(args);
                  }
              })
             .ConfigureLogging((hostingContext, logging) => {
                    logging.AddConfiguration(
                        hostingContext.Configuration.GetSection("Logging"));
                    logging.AddConsole();
                    logging.AddDebug();
              })
             .UseIISIntegration()
             .UseDefaultServiceProvider((context, options) => {
                 options.ValidateScopes =
                        context.HostingEnvironment.IsDevelopment();
              })
             .UseStartup(nameof(ConfiguringApps));
}
```

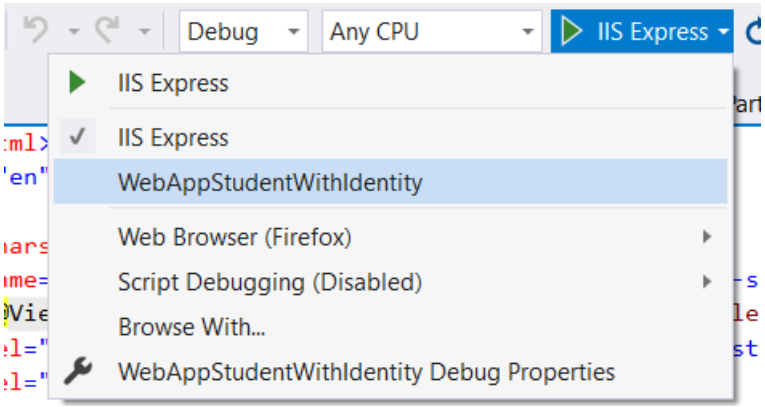
- Dla Core 5.0: https://docs.microsoft.com/pl-pl/aspnet/core/fundamentals/host/web-host?view=aspnetcore-5.0

`CreateDefaultBuilder` - szczegóły

- `UseKestrel()` – konfiguruje serwer Kestrel
- `UseContentRoot()` – ustawia katalog, z którego wczytywane są pliki konfiguracyjne oraz pliki statyczne (obrazki, skrypty JavaScript, arkusze CSS itp.)
- `ConfigureAppConfiguration()` - przygotowanie danych konfiguracyjnych (szczegóły dalej)
- `ConfigureLogging()` – dodanie rejestrowania informacji w pliku logów
- `UseIISIntegration()` – włączenie integracji z serwerami IIS i IIS express
- `UseDefaultServiceProvider()` – konfigurowanie domyślnego kontenera i mechanizmu wstrzykiwania zależności
- `UseStartup()` - wskazuje klasę do skonfigurowania platformy ASP .Net (szczegóły dalej)

Serwer Kestrel

- Niezależny od platformy sprzętowej serwer WWW przeznaczony do uruchamiania aplikacji ASP.NET
- Podczas uruchamiania można ten serwer wywołać bezpośrednio zmieniając w Visual Studio uruchamianie z „IIS Express” na nazwę projektu:



- W trakcie uruchamianie aplikacji uruchomi się serwer Kestrel w konsoli:

```
C:\Users\dariu\source\repos\WebAppStudentWithIdentity\WebAppStudentWithIdentity\bin\Debug\net5.0\WebAppStudentWithIden...
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: https://localhost:5001
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: http://localhost:5000
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\Users\dariu\source\repos\WebAppStudentWithIdentity\WebAppStudentWithIdentity
```

Serwer Kestrel - logging

- Pozwala to na obserwację informacji, błędów itd. , które można obserwować w oknie konsoli serwera
- Np. po wybraniu opcji „Private” w górnym menu:

HomeController.cs

```
0 references
public IActionResult Privacy()
{
    Console.WriteLine("starting Privacy()");
    return View();
}
```

```
C:\Users\dariu\source\repos\WebAppStudentWithIdentity\WebAppStudentWithIdentity\bin\Debug\net5.0\WebAppStudentWithIden...
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: https://localhost:5001
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: http://localhost:5000
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\Users\dariu\source\repos\WebAppStudentWithIdentity\WebAppStudentWithIdentity
starting Privacy()
```

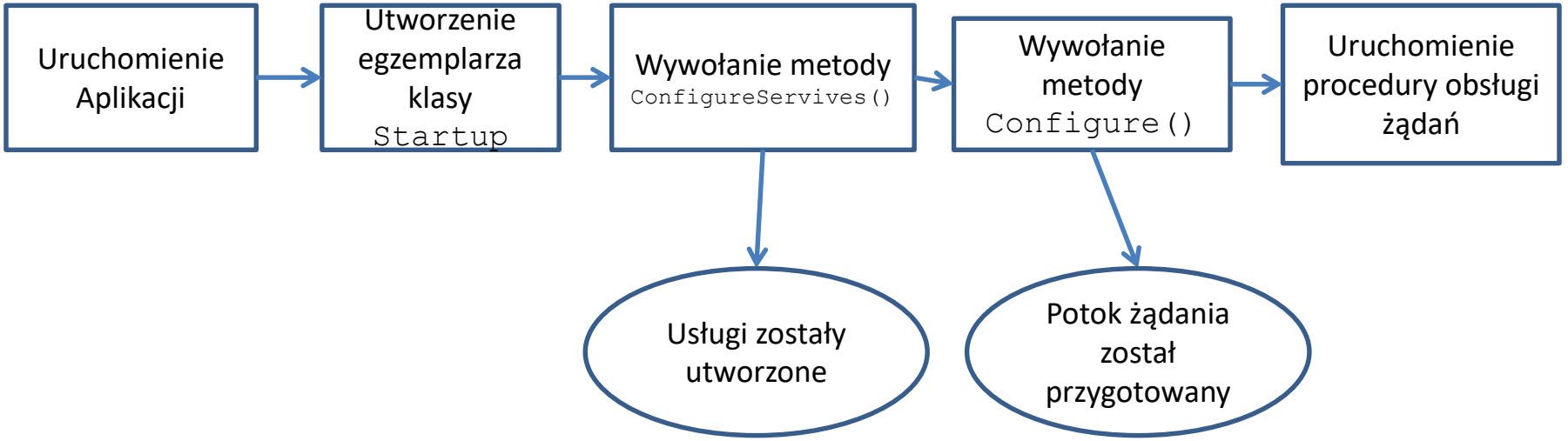
Klasa `Startup`

- Odpowiada za konfigurację ASP. Net Core
- Przykładowa minimalna zawartość dla pustego projektu ASP .Net (Core 2.1-5.0):
  - Zawsze wypisuje „Hello World!”, niezależnie od URL.

```
public class Startup
{

  public void Configure(IApplicationBuilder app)
  {
    app.Run(async (context) =>
    {
      await context.Response.WriteAsync("Hello World!");
    });
  }
}
```

Potok przetwarzania, oprogramowanie pośredniczące

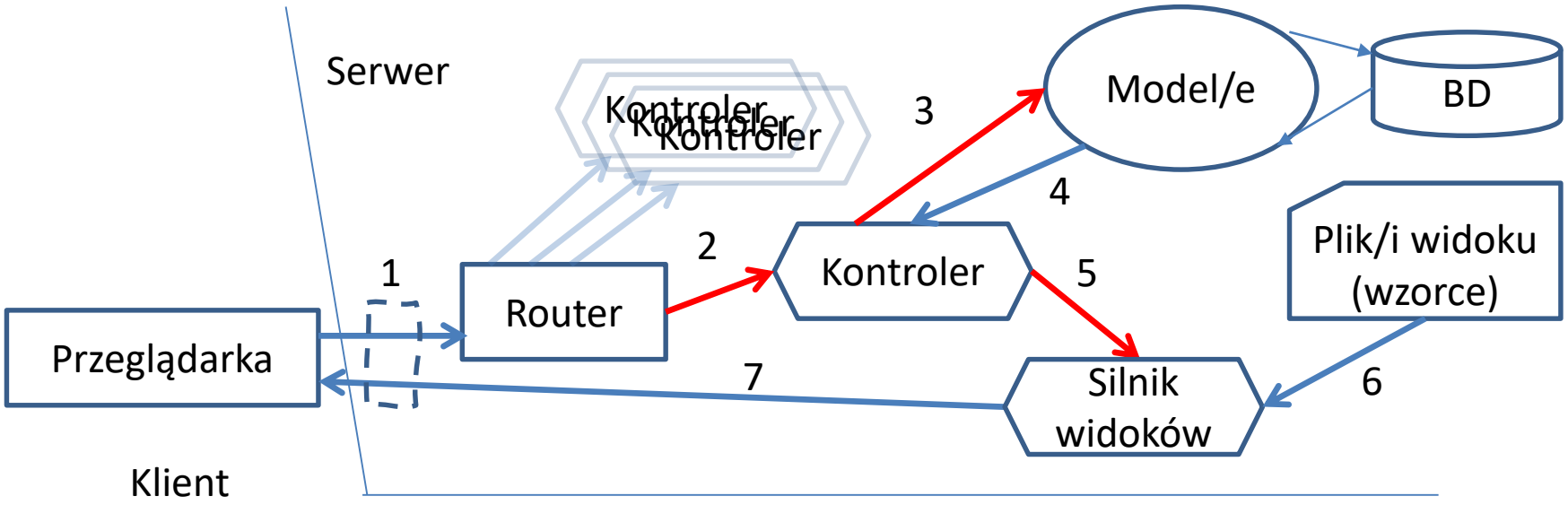
Sposób użycia klasy Startup



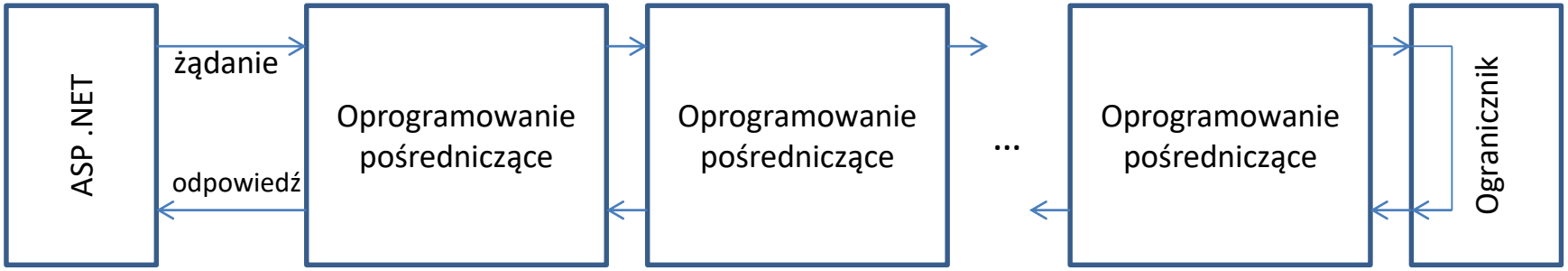
- Czyli podczas `configureServices()` tworzymy usługi
  - Również te powiązane ze wstrzykiwaniem w konstruktorach czy właściwościach
- Podczas `Configure()` tworzymy te powiązania między nimi, które nie da się wyrazić poprzez wstrzykiwanie
  - w tym potok przetwarzania.

MVC (i nie tylko) w kontekście aplikacji webowych (wykład W08)

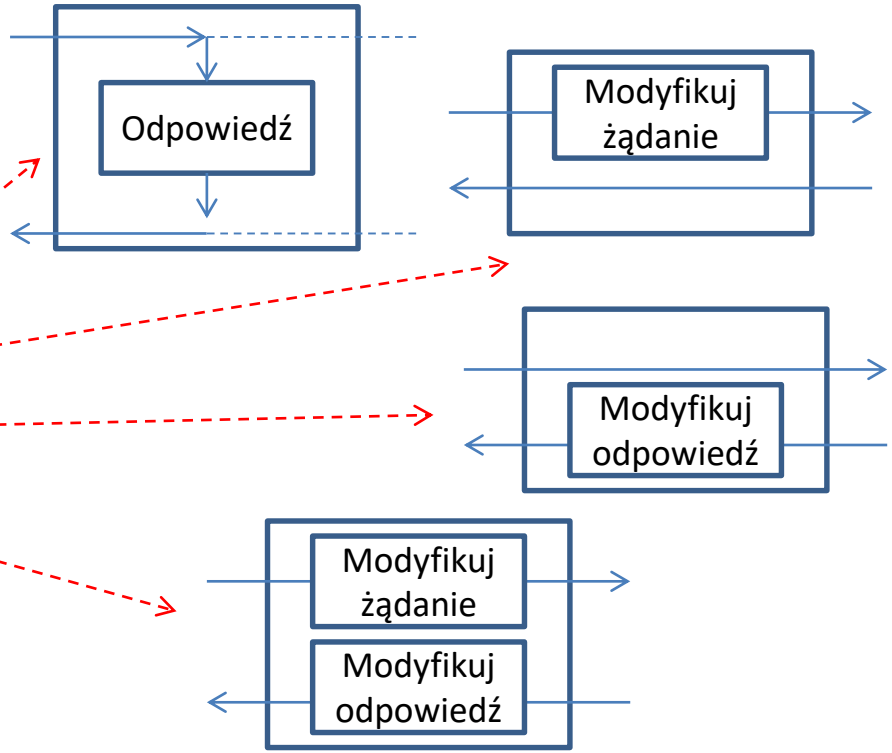
- Przebieg obsługi żądania HTTP w ASP:
  - Żądanie HTTP (1), przetworzone przez serwer, tworzy obiekt `HttpContext` (m. in. z właściwością `Request`) zawierający wszystkie informacje z żądania przetworzone na odpowiednie właściwości (ścieżka URL, parametry zapytania POST/GET/inne, ciasteczka, inne elementy nagłówka lub ciała żądania)
  - W większość przypadków nie obiekt ten nie będzie używany wprost (starsze podejście), ale informacje w nim zawarte będą używane wraz z mechanizmem odbicia do kolejnych kroków.
  - Na drodze (1) działa jeszcze tzw. **oprogramowanie pośredniczące**, które może zmodyfikować obiekt `HttpContext`.



Potok oprogramowania pośredniczącego (OP)



- Blok ASP .NET
  - odczytuje żądanie HTTP i „przepakowuje” je do `HttpContext.Request`
  - Odbiera wytworzoną odpowiedź i przekształca `HttpContext.Response` w odpowiedź HTTP
- Oprogramowania pośredniczące (OP, ang. middleware) ułożone są w ciąg.
- OP może wygenerować treść (odpowiedź) i nie przesłać dalej żądania
- OP może zmodyfikować żądanie i przesłać je dalej
- OP może zmodyfikować odpowiedź
- OP teoretycznie może równocześnie zmodyfikować żądanie, a po powrocie sterowania na strumieniu powrotnych zmodyfikować odpowiedź
- Wszystkie powyższe działania mogą być warunkowe, czyli tylko dla wybranych żądań lub odpowiedzi
- Ogranicznik zawraca potok w drugą stronę
  - Najczęściej poprawne żądanie do niego nie dotrze



Włączenie podstawowej usługi ASP

- Projekt `WebAppMiddleware`, ale w sposób narastający
- Dodanie MVC Core 5.0
- Uruchomienie podstawowej zasady routingu

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddControllersWithViews();
}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
   #region my middleware
   #endregion
   app.UseRouting();
   app.UseEndpoints(endpoints =>
   {
     endpoints.MapControllerRoute(
        name: "default",
        pattern: "{controller=Home}/{action=Index}/{id?}");
   });
}
```

- Stwórzmy serwis pokazujący czas działania aplikacji (zostanie dodany do kontenera serwisów)lo

```
public class UptimeService
{
  private Stopwatch timer;

  public UptimeService()
  {
    timer = Stopwatch.StartNew();
  }

  public long Uptime => timer.ElapsedMilliseconds;
}
```

Wstrzyknięcie serwisu w kontrolerze `HomeController`

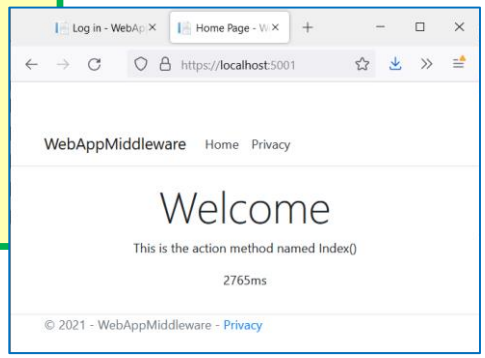
- W tym przypadku warto dodać serwis jako `AddSingleton`
  - Kontener działa dzięki `UseDefaultServiceProvider` z `Program.cs`
- Możemy stworzyć **tylko** akcję `Index` w kontrolerze `HomeController`
  - Brak akcji `Privacy()`
- Scenariusz użycia: uruchomienie

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddSingleton<UptimeService>();
  services.AddControllersWithViews();
}
```

```
public class HomeController : Controller
{
  private UptimeService uptime;
  public HomeController(UptimeService up)
  {
    uptime = up;
  }

   public IActionResult Index()
  {
    TempData["Message"] = "This is the action method named Index()";
    TempData["Uptime"] = ${uptime.Uptime}ms";
    return View();
  }
}
```

Index.cshtml | ContentMiddleware.cs | HomeControl

```
1  @{
2      ViewData["Title"] = "Home Page";
3  }
4
5  <div class="text-center">
6      <h1 class="display-4">Welcome</h1>
7      <p>@TempData["Message"]</p>
8      <p>@TempData["Uptime"]</p>
9  </div>
```



OP generujące odpowiedź

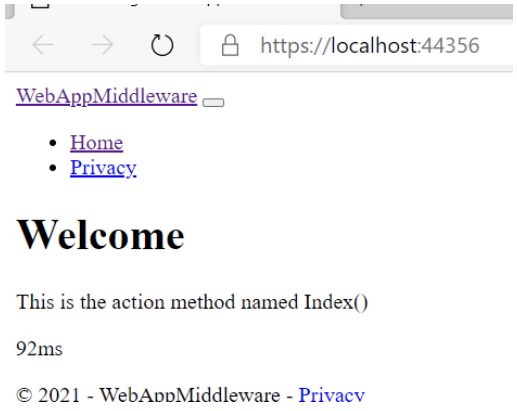
- Tworzenie OP, które przechwyci żądanie ze ścieżką „`/middleware`”, zanim uruchomi się domyślna reguła routingu
- Dodatkowo wstrzyknięcie stworzonego serwisu

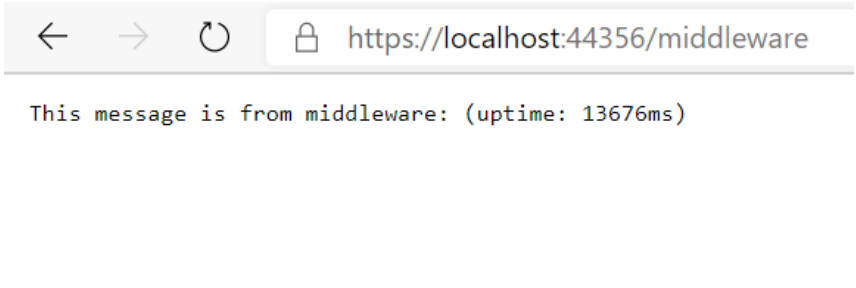
```
public class ContentMiddleware
{
  private RequestDelegate nextDelegate;
  private UptimeService uptime;

  public ContentMiddleware(RequestDelegate next, UptimeService up)
  {
    nextDelegate = next;
    uptime = up;
  }

  public async Task Invoke(HttpContext httpContext)
  {
    if (httpContext.Request.Path.ToString().ToLower() == "/middleware")
    {
      await httpContext.Response.WriteAsync(
        " This message is from middleware: " + $"(uptime: {uptime.Uptime}ms)", Encoding.UTF8);
          // do NOT call nextDelegate.Invoke
    }
    else
    {
      await nextDelegate.Invoke(httpContext);
    }
  }
}}
```

Użycie w konfiguracji

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
  …
  #region my middleware
  app.UseMiddleware<ContentMiddleware>();
  #endregion
  …
}
```

- Scenariusz użycia: adres URL z `/middleware` i bez
- Dlaczego nie w ramach zwykłych reguł routingu?
  - Pozwala w konfiguracji developerskiej dodać odpowiedzi na pewne żądania, których nie będzie w wersji użytkowej, a dodanie/usunięcie to tylko dodanie/usuniecie jednej linijki z procedury `Configure()`
  - Można wręcz napisać inne konfiguracje potoków oprogramowania pośredniczącego w zależności od środowiska

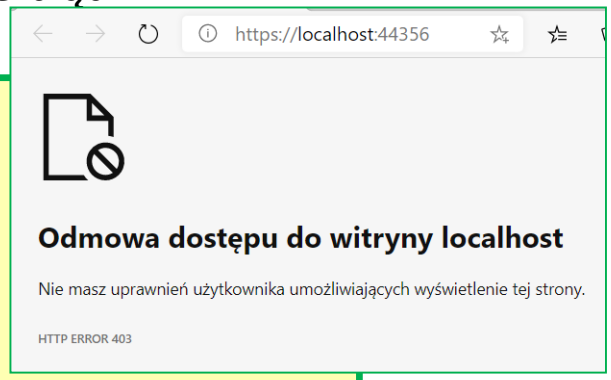


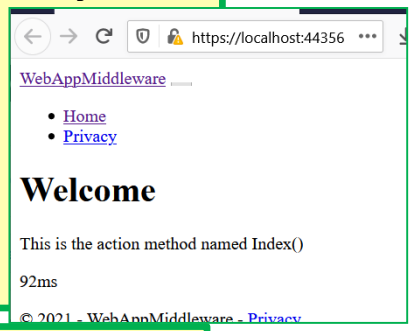


OP skracające potok żądań

- Załóżmy, że obecne rozwiązanie w pewnej przeglądarce (np. MS Edge) działa niepoprawnie, czy wręcz myląco.
- OP reagujące na takie żądanie i natychmiast zwracające błąd.
- Scenariusz użycia: w przeglądarce MS Edge i innej

```
public class ShortCircuitMiddleware
{
  private RequestDelegate nextDelegate;

  public ShortCircuitMiddleware(RequestDelegate next)
            => nextDelegate = next;

  public async Task Invoke(HttpContext httpContext)
  {
    if (httpContext.Request.Headers["User-Agent"].Any(v => v.ToLower().Contains("edg")))
    {
      httpContext.Response.StatusCode = 403;
    }
    else
    {
      await nextDelegate.Invoke(httpContext);
    }
  }
}
```






```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{ …
  #region my middleware
  app.UseMiddleware<ShortCircuitMiddleware>();
  app.UseMiddleware<ContentMiddleware>();
  #endregion
… }
```

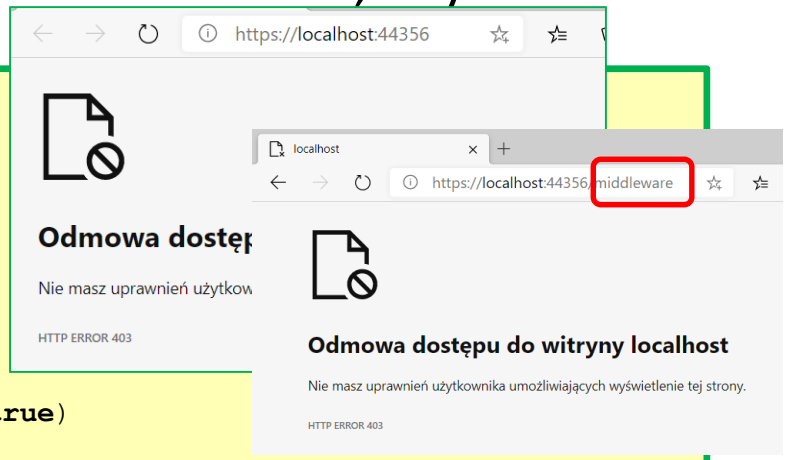
OP modyfikujące żądanie 1/2

- `HttpContext` posiada słownik pod właściwością `Items`.
- Rozbijamy poprzednie OP na dwa:
  - OP sprawdzające typ przeglądarki i jeśli jest to MS Edge, to ustawia wybrany klucz w słowniku `Items` (np. `Item["EdgeBrowser"]`) na **`true`** oraz wykonuje kolejne OP w ciągu
  - OP sprawdzające ten klucz i generujące ewentualnie kod błędu.

```
public class BrowserTypeMiddleware
{
  private RequestDelegate nextDelegate;

  public BrowserTypeMiddleware(RequestDelegate next)
            => nextDelegate = next;

  public async Task Invoke(HttpContext httpContext)
  {
    httpContext.Items["EdgeBrowser"]
          = httpContext.Request.Headers["User-Agent”].Any(v => v.ToLower().Contains("edg"));
    await nextDelegate.Invoke(httpContext);
  }
}
```

OP modyfikujące żądanie 2/2

- Zmodyfikujemy `ShortCircuitMiddleware` tak, aby korzystało z kolekcji `Items`




```
public class ShortCircuitMiddleware
{
  private RequestDelegate nextDelegate;

  public ShortCircuitMiddleware(RequestDelegate next)
           => nextDelegate = next;

  public async Task Invoke(HttpContext httpContext)
  {
    if (httpContext.Items["EdgeBrowser"] as bool? == true)
    {
      httpContext.Response.StatusCode = 403; // do NOT call nextDelegate.Invoke()
    }
    else
    {
      await nextDelegate.Invoke(httpContext);
    }
  }
}
```

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{ …
  #region my middleware
  app.UseMiddleware<BrowserTypeMiddleware>();
  app.UseMiddleware<ShortCircuitMiddleware>();
  app.UseMiddleware<ContentMiddleware>();
  #endregion
…}
```

OP modyfikujące odpowiedź

- OP, które powiadomi, że przeglądarka MS Edge nie jest obsługiwana w czytelny dla użytkownika sposób.

```
public class ErrorMiddleware
{
  private RequestDelegate nextDelegate;

  public ErrorMiddleware(RequestDelegate next)
  {
    nextDelegate = next;
  }

  public async Task Invoke(HttpContext httpContext)
  {
    await nextDelegate.Invoke(httpContext); // forward without changes

    // on return path modify response
    if (httpContext.Response.StatusCode == 403 &&
        (httpContext.Items["EdgeBrowser"] as bool? == true)) // one condition is enough
    {
      await httpContext.Response
              .WriteAsync("The Microsoft Edge web browser is unsupported.", Encoding.UTF8);
    }
    else if (httpContext.Response.StatusCode == 404)
    {
      await httpContext.Response
              .WriteAsync("No content.", Encoding.UTF8);
    }
  }
}
```

Cały potok OP

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{ …
  #region my middleware
  app.UseMiddleware<ErrorMiddleware>();
  app.UseMiddleware<BrowserTypeMiddleware>();
  app.UseMiddleware<ShortCircuitMiddleware>();
  app.UseMiddleware<ContentMiddleware>();
  #endregion
…}
```

- Analiza potoku do przodu i od tyłu.
- Scenariusz użycia:
  - Przez MS Edge
  - URL z „`/middleware`”
  - „`/`” lub „`/Home`” lub „`/Home/Index`”
  - „`/zlyAdres`”
  - Analogicznie dla np. przeglądarki Firefox

Potok - podsumowanie

- Ważna jest kolejność bloków w potoku
  - W przykładzie wstawienie oprogramowanie pośredniczącego `app.UseMiddleware<ContentMiddleware>()` jako pierwsze spowoduje, że adres …/`middleware` będzie działał tak samo dla każdej przeglądarki
  - Część bloków zależy od przekształceń wcześniejszych lub zakładają odrzucenie (filtr) niepoprawnych żądań (i już tego nie sprawdzają)
- Większość potoku działa na nagłówku żądania/odpowiedzi (czyli używa odpowiednie pola `HttpContext`), dopiero na końcu potoku analizowane jest ciało żądania. Czyli dopiero finalnie działa:
  - Mechanizm data binding
  - Wywołanie akcji kontrolera
- Istnieje wiele pakietów z oprogramowaniem pośredniczącym wewnątrz pakietu
- Wiele pakietów, aby poprawnie działać, wymaga wstawienia oprogramowania pośredniczącego (i to w odpowiednim miejscu) jak również, oczywiście, rejestracji serwisu/ów w kontenerze.
  - Np. `UseSession()`,
  - Przykład pokazywał, że nie działa Bootstrap, bo nie było `app.UseStaticFiles();`

Konfigurowanie projektu, część 2

Argumenty metody `Configure()`

- `IApplicationBuilder` – interfejs do konfigurowania potoku OP. Posiada 3 metody w interfejsie, ale ponad setkę metod rozszerzających w tym `UseMiddleware()`, `UseMvcWithDefaultRoute()` itp.
- `IHostingEnvironment` – interfejs do rozróżniania środowisk hostingu, np. programistycznego i produkcyjnego (do Core 2.2). Posiada właściwości:
  - `ApplicationName`
  - `EnvironmentName`
  - `ContentRootPath` – ścieżka do plików zawierających treść i konfigurację aplikacji
  - `WebRootPath` – katalog zawierający statyczną treść aplikacji (najczęściej „`/wwwroot`”)
  - `ContentRootFileProvider` – zwraca `IFileProvider` z dostępem do odczytu plików z `ContentRootPath`
  - `WebRootFileProvider` - zwraca `IFileProvider` z dostępem do odczytu plików z `WebRootPath`
- Posiada metody:
  - `IsDevelopment()`
  - `IsStaging()`
  - `IsProduction()`
  - `IsEnvironment(string env)`
- `IWebHostEnvironment` (od Core 3.0) – jak `IHostingEnvironment`, ale jedynie dostęp do plików konfiguracyjnych

Wykorzystanie `IWebHostEnvironment`

- Stworzony potok OP skonfigurujmy tylko dla środowiska developerskiego

```
public class Startup
{
  public void ConfigureServices(IServiceCollection services)
  {
    services.AddSingleton<UptimeService>();
    services.AddControllersWithViews();
  }

  public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
  {
    if(env.IsDevelopment()){
      app.UseMiddleware<ErrorMiddleware>();
      app.UseMiddleware<BrowserTypeMiddleware>();
      app.UseMiddleware<ShortCircuitMiddleware>();
      app.UseMiddleware<ContentMiddleware>();
    }
    app.UseRouting();
    app.UseEndpoints(endpoints =>
            {
                endpoints.MapControllerRoute(
                    name: "default",
                    pattern: "{controller=Home}/{action=Index}/{id?}");
            });
  }
}
```

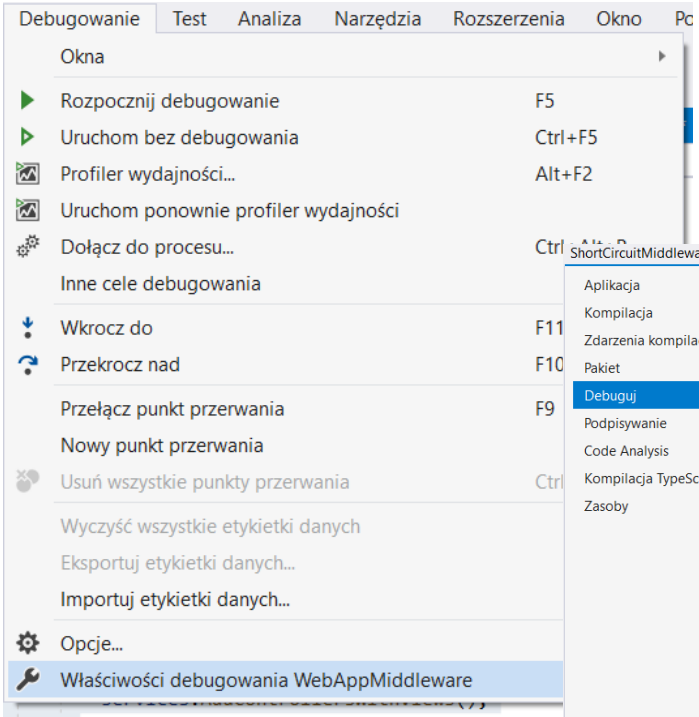
Konfigurowanie aplikacji ASP - inaczej

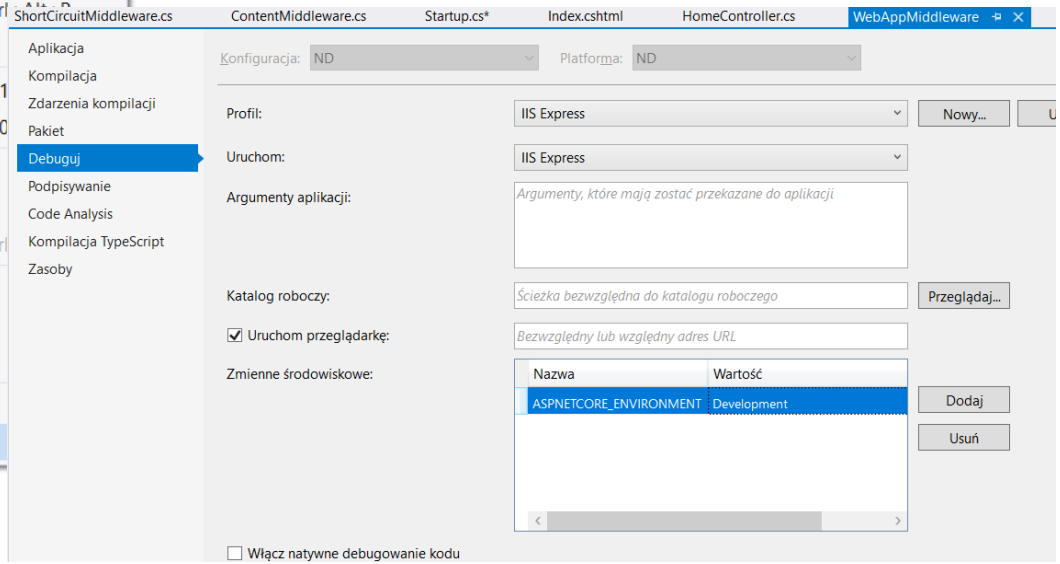
- ASP najpierw szuka metod konfiguracji serwisów i konfiguracji aplikacji z nazwami środowiska i jeżeli takie znajdzie, to zamiast metod ogólnych, uruchomi te z nazwą środowiska. Przykład poniżej.
- Podobnie można stworzyć pliki `.json` dla konfiguracji różnych środowisk dzięki `.AddJsonFile($"appsettings.{env.EnvironmentName}.json"` (strona 31 tego wykładu)
- Więcej: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/environments?view=aspnetcore-5.0

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddSingleton<UptimeService>(); // for HomeController
  services.AddControllersWithViews();
}
public void ConfigureDevelopmentServices(IServiceCollection services)
{
  services.AddSingleton<UptimeService>();
  services.AddControllersWithViews();
}
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
  app.UseExceptionHandler("/Home/Error");
  app.UseHsts();
  app.UseHttpsRedirection();
  app.UseStaticFiles();
  app.UseRouting();
          // …
}
public void ConfigureDevelopment(IApplicationBuilder app, IWebHostEnvironment env)
{
  app.UseDeveloperExceptionPage();
  app.UseMiddleware<ErrorMiddleware>();
  app.UseMiddleware<BrowserTypeMiddleware>();
  app.UseMiddleware<ShortCircuitMiddleware>();
  app.UseMiddleware<ContentMiddleware>();
  app.UseRouting();
  // …
}
```

Zmiana środowiska

- Zmianę środowiska można dokonać w środowisku systemu operacyjnego
- Można też w ramach projektu:





- Więcej informacji:
  - https://docs.microsoft.com/en-us/aspnet/core/fundamentals/environments?view=aspnetcore-5.0