

Projektowanie aplikacji ASP.NET

Wykład 08/15

ASP.NET Autentykacja, autoryzacja

Wiktor Zychla 2024/2025

Spis treści

1	Autoryzacja.....	2
1.1	Filtr autoryzacji.....	3
1.2	Polityka autoryzacji	3
2	Model danych dla autentykacji / autoryzacji, bezpieczeństwo.....	5
2.1	Połączenie szyfrowane	5
2.2	Użytkownicy, role	5
2.3	Hasła	5
2.4	Przywracanie dostępu do konta	6
3	Uwierzytelnianie dwuskładnikowe.....	7
4	Uwierzytelnianie federacyjne, protokoły Single Sign-on.....	8
4.1	Protokół WS-Federation.....	8
4.2	Protokół OAuth2.....	9
4.3	Przykład	10
4.3.1	Implementacja dla .NET Framework	12
4.3.2	Implementacja dla .NET.Core	15
5	Fido2.....	17

1 Autoryzacja

W trakcie wykładu pokażemy użycie ról użytkowników do zabezpieczania dostępu do zasobów. Podstawą pracy z rolami jest model wykorzystujący tożsamość typu **ClaimsPrincipal** – na poprzednim wykładzie pokazaliśmy jak pracować z takimi tożsamościami zarówno w przypadku

- **.NET.Framework** – wymaga użycia modułu **SessionAuthenticationModule**
- **.NET.Core** – natywne wsparcie

Najprostszy sposób użycia ról to po prostu rozbudowanie tożsamości o claimy typu rola – w takim układzie aplikacja prawidłowo reaguje na atrybut autoryzacyjny ze wskazaniem roli. W przeciwieństwie do claimu z nazwą użytkownika, claim z rolą może być wielokrotny.

Wydanie oświadczeń:

```
List<Claim> claims = new List<Claim>
{
    new Claim(ClaimTypes.Name, model.UserName),
    new Claim(ClaimTypes.Role, "..."),
    new Claim(ClaimTypes.Role, "...")
};

// create identity
ClaimsIdentity identity = new ClaimsIdentity(claims, CookieAuthentication
Defaults.AuthenticationScheme);
ClaimsPrincipal principal = new ClaimsPrincipal(identity);
```

i potem na akcji lub na kontrolerze:

```
[Authorize( Roles = "admin")]
public IActionResult Admin()
{
    return Content( $"Zalogowany admin jako {this.User.Identity.Name}" );
}
```

Middleware autoryzacji w .NET Core w przypadku nieprawidłowej autoryzacji przekierowuje żądanie na ścieżkę wyspecyfikowaną w konfiguracji (**AccessDeniedPath**). W .NET Framework następuje przekierowanie do strony logowania.

```
builder.Services
    .AddAuthentication( CookieAuthenticationDefaults.AuthenticationScheme )
    .AddCookie( CookieAuthenticationDefaults.AuthenticationScheme, options
=>
    {
        options.LoginPath = "/Account/Logon";
        // options.AccessDeniedPath = ""; ścieżka przekierowania dla braku
        // uprawnień
        options.SlidingExpiration = true;
    } );
```

W przypadku bardziej złożonych wymagań co do autoryzacji można posłużyć się **filtrem autoryzacji** (.NET Framework i .NET.Core) lub **polityką autoryzacji** (.NET Core).

1.1 Filtr autoryzacji

```
[AttributeUsage( AttributeTargets.Class | AttributeTargets.Method, AllowMultiple = true, Inherited = true )]
public class CustomAuthorizeAttribute : AuthorizeAttribute, IAuthorizationFilter
{
    private readonly string _userName;

    public CustomAuthorizeAttribute( string userName )
    {
        _userName = userName;
    }

    public void OnAuthorization( AuthorizationFilterContext context )
    {
        var user = context.HttpContext.User;

        if ( !user.Identity.IsAuthenticated )
        {
            // it isn't needed to set unauthorized result
            // as the base class already requires the user to be authenticated

            // this also makes redirect to a login page work properly
            // context.Result = new UnauthorizedResult();
            return;
        }

        var isAuthorized = context.HttpContext.User.Identity.Name == this._userName;
        if ( !isAuthorized )
        {
            context.Result = new StatusCodeResult( (int)System.Net.HttpStatusCode.Forbidden );
            return;
        }
    }
}
```

Użycie

```
[CustomAuthorize("foo")]
public IActionResult CustomAuthorize()
{
    return Content( $"CustomAuthorize admin jako {this.User.Identity.Name}" );
}
```

1.2 Polityka autoryzacji

Mechanizm [polityk autoryzacji](#) w .NET.Core jest ogólniejszy niż mechanizm filtrów.

Zaczynamy od zdefiniowania polityki

```
public class MinimumAgeRequirement : IAuthorizationRequirement
{
    public MinimumAgeRequirement( int minimumAge ) =>
        MinimumAge = minimumAge;

    public int MinimumAge { get; }
}
```

i zarejestrowania jej

```
builder.Services
    .AddAuthorization( options =>
    {
        options.AddPolicy("Over18",
            policy => policy.Requirements.Add( new MinimumAgeRequirement( 1
8 ) ) );
    } );
```

Polityki już można używać:

```
[Authorize( Policy = "Over18")]
public IActionResult Over18()
{
    return Content( $"Over18 jako {this.User.Identity.Name}" );
}
```

tylko że żeby zadziałała, musi mieć zarejestrowaną implementację

```
public class MinimumAgeHandler : AuthorizationHandler<MinimumAgeRequirement>
{
    protected override Task HandleRequirementAsync(
        AuthorizationHandlerContext context, MinimumAgeRequirement requirem
ent )
    {
        var dateOfBirthClaim = context.User.FindFirst( c => c.Type == Claim
Types.DateOfBirth );

        if ( dateOfBirthClaim is null )
        {
            return Task.CompletedTask;
        }

        var dateOfBirth = Convert.ToDateTime(dateOfBirthClaim.Value);
        int calculatedAge = DateTime.Today.Year - dateOfBirth.Year;
        if ( dateOfBirth > DateTime.Today.AddYears( -calculatedAge ) )
```

```

        {
            calculatedAge--;
        }

        if ( calculatedAge >= requirement.MinimumAge )
        {
            context.Succeed( requirement );
        }

        return Task.CompletedTask;
    }
}

```

oraz

```
builder.Services.AddSingleton<IAuthorizationHandler, MinimumAgeHandler>();
```

2 Model danych dla autentykacji / autoryzacji, bezpieczeństwo

2.1 Połączenie szyfrowane

Ponieważ POST formularza logowania niesie ze sobą login i hasło, krytyczne jest użycie połączenia szyfrowanego (SSL). Dawniej mylnie sądzono że po zalogowaniu aplikacja może przejść na kanał nieszyfrowany, jednak z takiego kanału można wykraść ciastko autentykacji (jak?) i doklejać je do preparowanych żądań do serwera. Dlatego obecnie zdecydowanie sugeruje się kanał szyfrowany do całej aplikacji.

2.2 Użytkownicy, role

Minimalny zestaw danych o użytkowniku jaki musi przechowywać aplikacja to **login**. Dobrym pomysłem jest przechowywanie **adresu e-mail**, w wielu przypadkach system może być zbudowany tak że loginem jest adres email.

Model ról o którym już mówiliśmy w poprzednim wykładzie zakłada że użytkownik może mieć przydzielone wiele ról. W efekcie w modelu danych mówimy o mapowaniu wiele-wiele między użytkownikami a rolami.

2.3 Hasła

Jeżeli sprawdzenie pary login/hasło odwołuje się do trwałego magazynu danych (np. baza danych) to pojawia się kwestia przechowywania haseł po stronie serwera:

- Pod żadnym pozorem nie wolno na serwerze przechowywać haseł w postaci jawnej
- Zamiast tego należy stosować jednokierunkowe funkcje skrótu o dużej entropii, np. [SHA2](#)

- Nawet dobra funkcja jednokierunkowa nie chroni przed atakiem tzw. [rainbow table](#) w którym koszt odwrócenia statystycznie dużej liczby haseł jest niewielki i chronione są wyłącznie nietypowe hasła
- Dlatego serwer dodatkowo chroni hasła użytkowników – przez hashowaniem dodając do nich tzw. [salt](#) czyli dodatkowy element entropii, utrudniający atak słownikowy. Salt jest przechowywany w bazie, obok zhaszowanego hasła i jest unikalny (losowy) dla każdego przechowywanego hasła
- Do tego, aby utrudnić odwracanie, stosuje się iterowanie funkcji skrótu

$$P = \text{SHA256}(\dots \text{SHA256}(\text{SHA256}(\text{password} + \text{salt}) + \text{salt}) \dots + \text{salt})$$

Liczbę iteracji dobiera się tak aby wyliczanie było jeszcze akceptowalne jeśli chodzi o czas (np. 50-500 ms) ale odwracanie – wtedy odpowiednio trudniejsze.

W praktyce stosuje się algorytmy [bcrypt](#) lub równoważne ([PBKDF2](#)). Wersja dla .NET wymaga zewnętrznego pakietu, np. [Bcrypt.Net](#).

Z uwagi na częsty wymóg polityki bezpieczeństwa, który wymaga unikalności hasła w odniesieniu do **poprzednich** haseł, w modelu danych wymaga się żeby relacja użytkownik-hasło była typu jeden-wiele (każdy użytkownik ma listę haseł, tylko ostatnie jest używane przy weryfikacji ale jakaś część poprzednich jest używana do określenia unikalności przy ustawianiu nowego hasła).

2.4 Przywracanie dostępu do konta

Typowy scenariusz w którym użytkownik traci dostęp do konta często rozwiązuje się w taki sposób, że na dedykowanej stronie użytkownik podaje adres e-mail związany z kontem. Aplikacja generuje parę identyfikator o dużej entropii (np. guid) – login

i zapisuje tę parę w pomocniczej tabeli w bazie danych, a następnie aplikacja używa protokołu SMTP do wysłania do użytkownika wiadomości email z linkiem postaci

<https://aplikacja.....com/PasswordReset?token={identyfikator}>

Użytkownik klikając w link nawiguje do strony, która w parametrze o nazwie **token** dostaje wartość identyfikatora. Są dwa przypadki:

- Identyfikator nie występuje w pomocniczej tabeli – użytkownikowi zwraca się informację o błędzie (niepoprawny token). Dodatkowo aplikacja może zweryfikować kiedy token był wydany i na przykład nie akceptować tokenów starszych niż 24h
- Identyfikator występuje w pomocniczej tabeli – w tej pomocniczej tabeli występuje też login. Aplikacja ma gwarancję że ktoś kto kliknął w link ma na pewno dostęp do adresu email związanego z kontem o podanym loginie. Takemu użytkownikowi można wyświetlić stronę umożliwiającą bezwarunkowe ustawienie nowego hasła do tego właśnie konta.

3 Uwierzytelnianie dwuskładnikowe

Uwierzytelnianie dwuskładnikowe opiera się współcześnie o specyfikację [RFC 6328](#) i mechanizm zwany **Timed One Time Passwords (TOTP)**. Schemat wygląda następująco:

- Użytkownikowi przydzielany jest **klucz główny** (np. guid), klucz zapamiętywany jest w postaci jawnej
- Użytkownik korzysta z apki mobilnej zgodnej ze specyfikacją TOTP, na przykład Google Authenticator czy Microsoft Authenticator
- Aplikacja co 30 sekund generuje skrót (SHA1 lub SHA256) klucza głównego z datą (zaokrągloną do 30 sekund) i prezentuje użytkownikowi 6 cyfr będących fragmentem skrótu
- Użytkownik przepisuje 6 cyfr na formularzu logowania, dodatkowo, oprócz hasła
- Aplikacja weryfikuje hasło a dodatkowo serwer powtarza schemat apki mobilnej – używa klucza głównego i bieżącej daty, wylicza 6 cyfr ze skrótu i porównuje z tym co na formularzu logowania wpisał użytkownik

W ASP.NET możliwe jest użycie zewnętrznych bibliotek, m.in. [TwoFactorAuth.NET](#) czy [Otp.NET](#)

4 Uwierzytelnianie federacyjne, protokoły Single Sign-on

Uwierzytelnianie za pomocą zewnętrznego dostawcy możliwe jest wyłącznie przy zapewnieniu bezpieczeństwa, w szczególności braku możliwości oszukania przepływu kontroli między dwoma różnymi aplikacjami przez użytkownika.

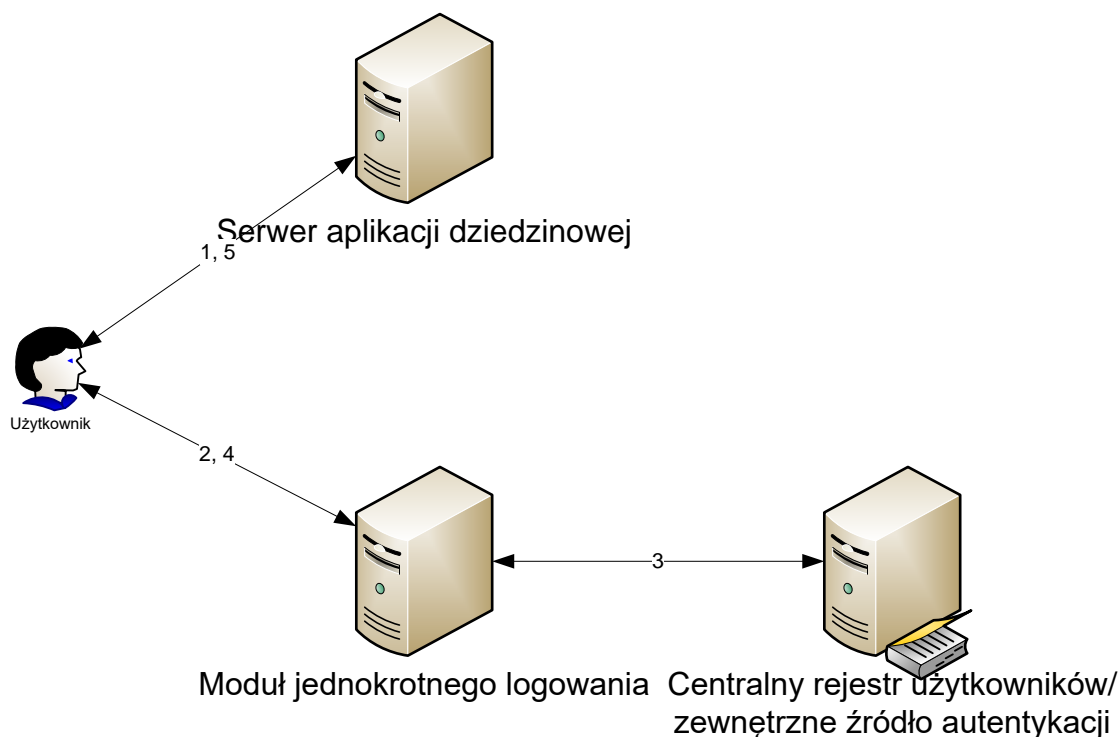
Z tego powodu współcześnie korzysta się z tzw. protokołów SSO, np.:

- protokół **passive** [WS-Federation](#), który definiuje przepływ komunikatów dla klienta pasywnego (przeglądarka internetowa) i umożliwia uzyskanie poświadczonej przez serwer informacji o tożsamości użytkownika i jego przynależności do ról (tu: grup zabezpieczeń). Protokół należy do rodziny WS-* i jest uznanym, przyjętym powszechnie w przemyśle rozwiązaniem, dla którego istnieją gotowe implementacje części klienckich i serwerowych dla różnych platform technologicznych – w przypadku systemu heterogenicznego jest to duża zaleta, otwierająca perspektywę łatwej rozbudowy systemu o kolejne moduły w przyszłości.
- Protokół [OAuth2/OpenID Connect](#), szeroko implementowany przez dostawców usług społecznościowych

Na potrzeby każdego wdrożenia systemu identyfikuje się podsystem nazywany dalej **modułem jednokrotnego logowania**, który w nomenklaturze technicznej jest dostawcą tożsamości (security token service, identity provider) protokołu pojedynczego logowania.

4.1 Protokół WS-Federation

Rysunek 1 przedstawia schemat poświadczania tożsamości przy wykorzystaniu WS-Federation i modułu jednokrotnego logowania.



Rysunek 1 Poświadczanie tożsamości przy wykorzystaniu WS-Federation i dostawcy tożsamości

Poszczególne kroki protokołu przedstawiają się następująco:

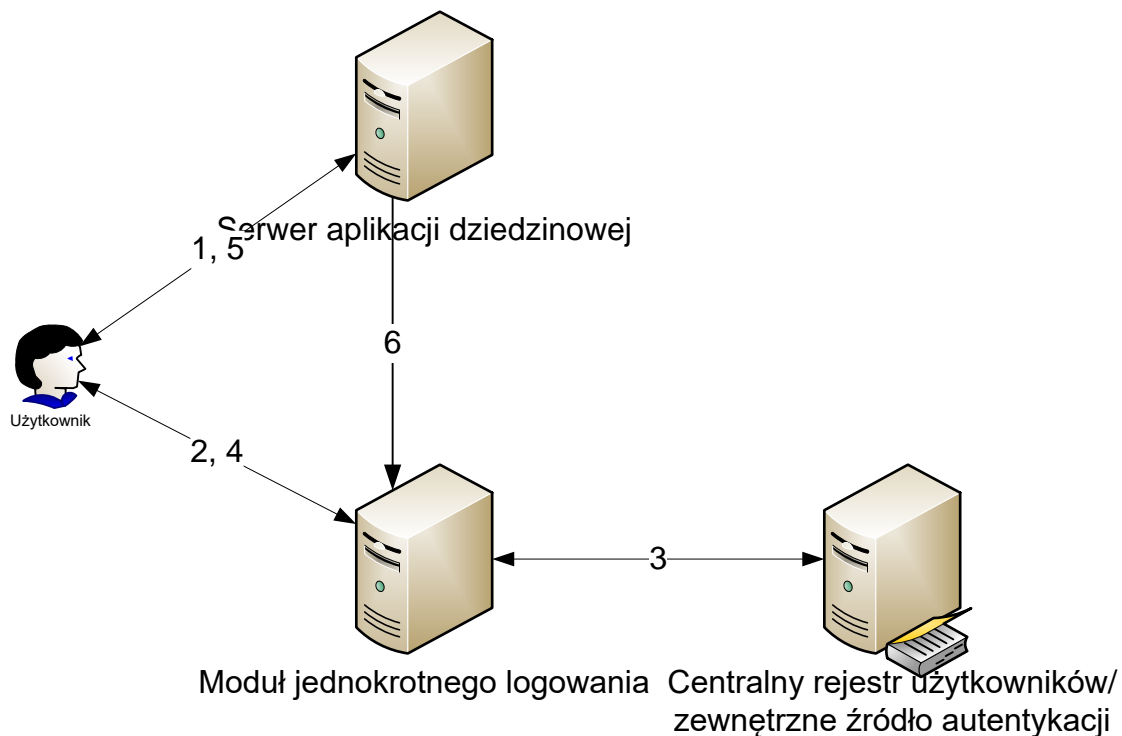
1. Użytkownik kieruje żądanie do wybranego serwera aplikacji obsługującego jeden z modułów systemu
2. Jeśli moduł do tej pory nie przeprowadził autentykacji tego użytkownika, za pośrednictwem przeglądarki kierowane jest żądanie wydania informacji o użytkowniku do serwera modułu jednokrotnego logowania
3. Serwer jednokrotnego logowania poświadcza tożsamość użytkownika, samodzielnie lub delegując autentykację dalej, do zaufanego dostawcy.
4. Serwer jednokrotnego logowania tworzy tzw. *token bezpieczeństwa* użytkownika zgodny ze standardem SAML, zawierający atrybuty opisujące użytkownika (**nazwa logowania, imię, nazwisko, unikalny identyfikator, adres e-mail i przynależność do grup zabezpieczeń**).
5. Serwer jednokrotnego logowania **podpisuje** token bezpieczeństwa, uniemożliwiając w ten sposób jego zafałszowanie i poświadczając jego wiarygodność i za pośrednictwem przeglądarki odsyła informację do właściwego serwera aplikacji. Token bezpieczeństwa (właściwie: token SAML) ma postać dokumentu XML.
6. Serwer aplikacji waliduje integralność przedstawionego tokenu bezpieczeństwa i przydziela użytkownikowi dostęp do właściwych zasobów w ramach zawartej w tokenie bezpieczeństwa informacji o przynależności użytkownika do grup zabezpieczeń

Szczegółowa dokumentacja techniczna protokołu autentykacji WS-Federation, zawartości i sposobu interpretacji tokenów SAML są publicznie dostępne i nie zostaną dołączone do niniejszego opracowania.

Należy zwrócić uwagę, że jedną z pożądanych właściwości specyfikacji WS-Federation jest obsługa scenariusza Single Sign-out, czyli możliwość wylogowania się użytkownika z całego środowiska aplikacyjnego przez jeden wspólny odnośnik. Technicznie realizowane jest to następująco – podczas autentykacji użytkowników na potrzeby konkretnych aplikacji (krok 3) serwer jednokrotnego logowania w sesji użytkownika zapamiętuje odnośniki do tych aplikacji. W ten sposób w każdym momencie serwer jednokrotnego logowania wie do których aplikacji użytkownik jest zalogowany za jego pośrednictwem. Wylogowanie sprowadza się do wygenerowania spreparowanej strony z odnośnikami do poszczególnych aplikacji z dołączonym specjalnym parametrem, który dla aplikacji jest równoznaczny z poleceniem wylogowania się.

4.2 Protokół OAuth2

Rysunek 2 przedstawia schemat poświadczania tożsamości przy wykorzystaniu OAuth2 i modułu jednokrotnego logowania.



Rysunek 2 Poświadczanie tożsamości przy wykorzystaniu OAuth2 i dostawcy tożsamości

Poszczególne kroki protokołu przedstawiają się następująco:

1. Użytkownik kieruje żądanie do wybranego serwera aplikacji obsługującego jeden z modułów systemu
2. Jeśli moduł do tej pory nie przeprowadził autentykacji tego użytkownika, za pośrednictwem przeglądarki kierowane jest żądanie wydania informacji o użytkowniku do serwera modułu jednokrotnego logowania (przekierowanie typu 302 z określonym zestawem parametrów)
3. Serwer jednokrotnego logowania poświadcza tożsamość użytkownika, samodzielnie lub delegując autentykację dalej, do zaufanego dostawcy.
4. Serwer jednokrotnego logowania tworzy tzw. *jednokrotny kod bezpieczeństwa*
5. Serwer jednokrotnego logowania tworzy przekierowanie zwrotne (302) do aplikacji
6. Serwer aplikacji zamienia jednokrotny kod bezpieczeństwa na tzw. *token bezpieczeństwa*, którego następnie używa do uzyskania informacji o użytkowniku (**nazwa logowania, imię, nazwisko, unikalny identyfikator, adres e-mail i przynależność do grup zabezpieczeń**) w module jednokrotnego logowania

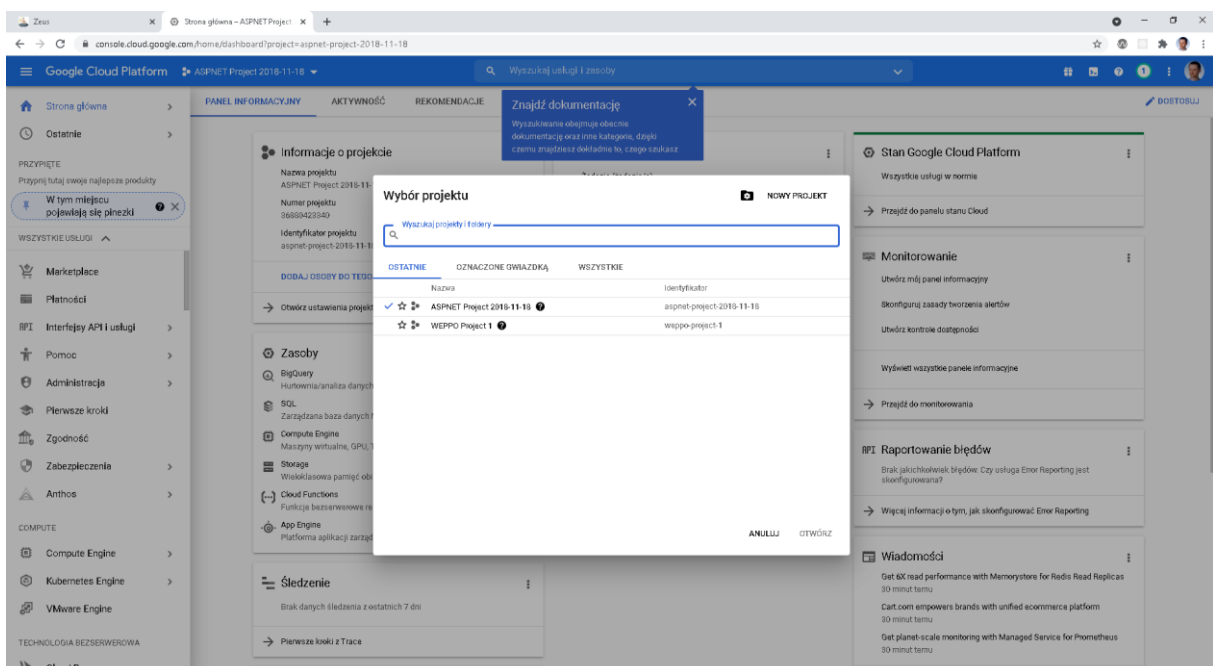
4.3 Przykład

W trakcie wykładu zobaczymy [przykład](#) użycia dostawcy (Google) do wykonania integracji logowania za pomocą protokołu OAuth2 i biblioteki **DotnetOpenAuth**. Uwaga na użyte w kodzie stałe reprezentujące adresy punktów końcowych usługi OAuth2 – Google aktualizuje te adresy regularnie, aktualny wypis znajduje się na adresie typu discovery, <https://accounts.google.com/.well-known/openid-configuration>.

← → ↻ 🔒 accounts.google.com/.well-known/openid-configuration

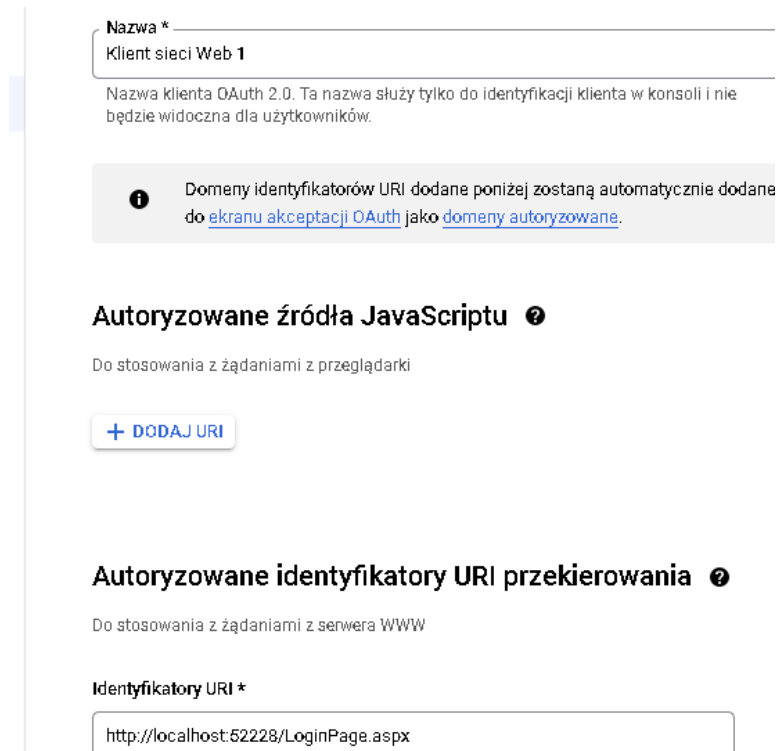
```
{
  "issuer": "https://accounts.google.com",
  "authorization_endpoint": "https://accounts.google.com/o/oauth2/v2/auth",
  "token_endpoint": "https://oauth2.googleapis.com/token",
  "userinfo_endpoint": "https://openidconnect.googleapis.com/v1/userinfo",
  "revocation_endpoint": "https://oauth2.googleapis.com/revoke",
  "jwks_uri": "https://www.googleapis.com/oauth2/v3/certs",
  "response_types_supported": [
    "code",
    "token",
    "id_token",
    "code token",
    "code id_token",
    "token id_token",
    "code token id_token",
    "none"
  ],
  "subject_types_supported": [
    "public"
  ],
  "id_token_signing_alg_values_supported": [
    "RS256"
  ],
  "scopes_supported": [
    "openid",
    "email",
    "profile"
  ],
  "token_endpoint_auth_methods_supported": [
```

Konfigurowanie należy zacząć od wizyty na stronie [konsoli deweloperskiej Google](#). Z listy na górnej belce należy wybrać istniejący projekt lub utworzyć nowy:



Z menu rozwijanego należy wybrać funkcję „Interfejsy API i usługi” a następnie „Włącz interfejsy API i usługi” i upewnić się że interfejsy usług społecznościowych są włączone („Google+ API”).

Następnie w „Interfejsy API i usługi” / „Dane logowania” należy utworzyć nowe dane logowania dla OAuth2. Ważne, aby w sekcji „Autoryzowane identyfikatory URI przekierowania” pojawił się adres zwrotny (http://nazwa.hosta/nazwa_zasobu) w aplikacji ASP.NET:



Nazwa *

Klient sieci Web 1

Nazwa klienta OAuth 2.0. Ta nazwa służy tylko do identyfikacji klienta w konsoli i nie będzie widoczna dla użytkowników.

i Domeny identyfikatorów URI dodane poniżej zostaną automatycznie dodane do [ekranu akceptacji OAuth](#) jako [domeny autoryzowane](#).

Autoryzowane źródła JavaScriptu ⓘ

Do stosowania z żądaniami z przeglądarki

[+ DODAJ URI](#)

Autoryzowane identyfikatory URI przekierowania ⓘ

Do stosowania z żądaniami z serwera WWW

Identyfikator URI *

<http://localhost:52228/LoginPage.aspx>

Następnie należy odczytać **identyfikator klienta** i **tajny klucz klienta**. W nomenklaturze OAuth2 są to odpowiednio **client_id** i **client_secret** – w przepływie logowania będą się pojawiać w momentach określonych protokołem:

- **client_id** (identyfikator klienta) pojawia się w przekierowaniu z aplikacji do dostawcy tożsamości (krok 2 na diagramie wyżej)
- **client_secret** (tajny klucz klienta) pojawia się w żądaniu z serwera aplikacji do dostawcy tożsamości (krok 6 na diagramie wyżej)

4.3.1 Implementacja dla .NET Framework

.Net.Framework nie posiada wbudowanej obsługi protokołu OAuth2, należy więc użyć zewnętrznej implementacji. W przykładzie poniżej jest to **DotNetOpenAuth.Ultimate**

```
public class GoogleClient : WebServerClient
{
    private static readonly AuthorizationServerDescription GoogleDescription =
        new AuthorizationServerDescription
        {
            TokenEndpoint = new Uri("https://accounts.google.com/o/oauth2/token"),
            AuthorizationEndpoint = new Uri("https://accounts.google.com/o/oauth2/auth"),
            ProtocolVersion = ProtocolVersion.V20,
```

```

        };

        public const string ProfileEndpoint = "https://openidconnect.googleapis.com/
v1/userinfo";

        public const string OpenId      = "openid";
        public const string ProfileScope = "profile";
        public const string EmailScope  = "email";

        public GoogleClient()
            : base(GoogleDescription)
        {
        }
    }

    public class GoogleProfileAPI
    {
        public string email { get; set; }
        public string given_name { get; set; }
        public string family_name { get; set; }

        private static DataContractJsonSerializer jsonSerializer =
            new DataContractJsonSerializer(typeof(GoogleProfileAPI));

        public static GoogleProfileAPI Deserialize(Stream jsonStream)
        {
            try
            {
                if (jsonStream == null)
                {
                    throw new ArgumentNullException("jsonStream");
                }

                return (GoogleProfileAPI)jsonSerializer.ReadObject(jsonStream);
            }
            catch (Exception ex)
            {
                return new GoogleProfileAPI();
            }
        }
    }

    public class MyAuthorizationTracker : IClientAuthorizationTracker
    {

        public IAuthorizationState GetAuthorizationState(
            Uri callbackUrl,
            string clientState)
        {
            return new AuthorizationState
            {
                Callback = new Uri(callbackUrl.GetLeftPart(UriPartial.Path))
            };
        }
    }
}

```

Sam przepływ autentykacji wymaga modyfikacji kodu strony logowania, gdzie odbywa się delegowanie uwierzytelnienia

```
public partial class LoginPage : System.Web.UI.Page
{
    public readonly GoogleClient gClient = new GoogleClient
    {
        AuthorizationTracker = new MyAuthorizationTracker(),
        ClientIdentifier = "...",
        ClientCredentialApplicator = ClientCredentialApplicator.PostParameter(".".
        .")
    };

    protected void Page_Load(object sender, EventArgs e)
    {
        IAuthorizationState authorization = gClient.ProcessUserAuthorization();
        // Is this a response from the Identity Provider
        if (authorization == null)
        {
            // no

            // Google will redirect back here
            Uri uri = new Uri("http://localhost:52228/LoginPage.aspx");
            // Kick off authorization request with OAuth2 scopes
            gClient.RequestUserAuthorization(returnTo: uri,
                scope: new[] { GoogleClient.OpenId, GoogleClient.ProfileScope, G
                oogleClient.EmailScope });
        }
        else
        {
            // yes

            var request = WebRequest.Create(GoogleClient.ProfileEndpoint);
            // add an OAuth2 authorization header
            // if you get 403 here, turn ON Google+ API on your app settings pag
            e
            request.Headers.Add(
                HttpRequestHeader.Authorization,
                string.Format("Bearer {0}", Uri.EscapeDataString(authorization.
                AccessToken)));
            // Go to the profile API
            using (var response = request.GetResponse())
            {
                using (var responseStream = response.GetResponseStream())
                {
                    var profile = GoogleProfileAPI.Deserialize(responseStream);
                    if (profile != null &&
                        !string.IsNullOrEmpty(profile.email))
                        FormsAuthentication.RedirectFromLoginPage(profile.email,
                        false);
                }
            }
        }
    }
}
```

4.3.2 Implementacja dla .NET.Core

Platforma .NET Core posiada wbudowane middleware logowania, **Microsoft.AspNetCore.Authentication.Google**

Po dodaniu middleware należy skonfigurować parametry

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllersWithViews();
builder.Services
    .AddAuthentication( options =>
    {
        options.DefaultScheme = CookieAuthenticationDefaults.Authentication
Scheme;
        //options.DefaultChallengeScheme = GoogleDefaults.AuthenticationSch
eme;
    } )
    .AddCookie( CookieAuthenticationDefaults.AuthenticationScheme, options
=>
    {
        options.LoginPath = "/Account/Logon";
        options.SlidingExpiration = true;
        options.Cookie.SameSite = SameSiteMode.Unspecified;
        options.Cookie.SecurePolicy = CookieSecurePolicy.SameAsRequest;
    } )
    .AddGoogle( GoogleDefaults.AuthenticationScheme, googleOptions =>
    {
        googleOptions.ClientId = "36880423340-
e3t0ionh0p72dg2araq43h5spbieeph5.apps.googleusercontent.com";
        googleOptions.ClientSecret = "_client_secret_";
        googleOptions.CorrelationCookie.SameSite = SameSiteMode.None;
        googleOptions.CorrelationCookie.SecurePolicy = CookieSecurePolicy.S
ameAsRequest;
    } );

var app = builder.Build();

app.UseRouting();

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

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

app.Run();
```

a następnie dodać obsługę logowania w kontrolerze/widoku

```

@model OAuth2GoogleCore.Models.LogonModel
<html>
    <body>
        <form method="post">
            <button name="provider" value="Google">Zaloguj Google</button>
        </form>

    </body>
</html>

```

```

public class AccountController : Controller
{
    public AccountController()
    {
    }

    [HttpGet]
    public async Task<IActionResult> Logon(string returnUrl)
    {
        var model = new LogonModel()
        {
            ReturnUrl = returnUrl
        };

        return View( model );
    }

    [HttpPost]
    public IActionResult Logon( string provider, string returnUrl )
    {
        return this.Challenge( new AuthenticationProperties()
        {
            RedirectUri = Url.Action( "GoogleResponse" )
        }, GoogleDefaults.AuthenticationScheme );
    }

    public async Task<IActionResult>
        GoogleResponse( string returnUrl = null, string remoteError = null
    )
    {
        var result = await HttpContext.AuthenticateAsync(CookieAuthenticati
onDefaults.AuthenticationScheme);
        var claims = result.Principal.Identities
            .FirstOrDefault().Claims.Select(claim => new
            {
                claim.Issuer,
                claim.OriginalIssuer,
                claim.Type,
                claim.Value
            });
        return Redirect( "/" );
    }
}

```


5 Fido2

Protokół Fido2 jest nowym protokołem uwierzytelnienia promowanym przez [Fido Alliance](#). Obsługa tego protokołu pojawia się właśnie w przeglądarkach internetowych i zyskuje coraz większą popularność. Należy liczyć się z tym że w najbliższych latach wiele witryn zechce użyć tego standardu.

W trakcie lektury warto zajrzeć tu:

- <https://webauthn.guide/>
- <https://www.passkeys.io/>
- <https://github.com/herrjemand/awesome-webauthn>
- <https://medium.com/webauthnworks/introduction-to-webauthn-api-5fd1fb46c285>

W szczególności zwracam uwagę na scenariusz logowania tzw. **usernameless** gdzie użytkownik nie podaje ani loginu ani hasła (sic!).

<https://webauthnworks.github.io/FIDO2WebAuthnSeries/WebAuthnIntro/UsernamelessExample.html>

Pojęcia:

FIDO2 – nazwa protokołu uwierzytelniania

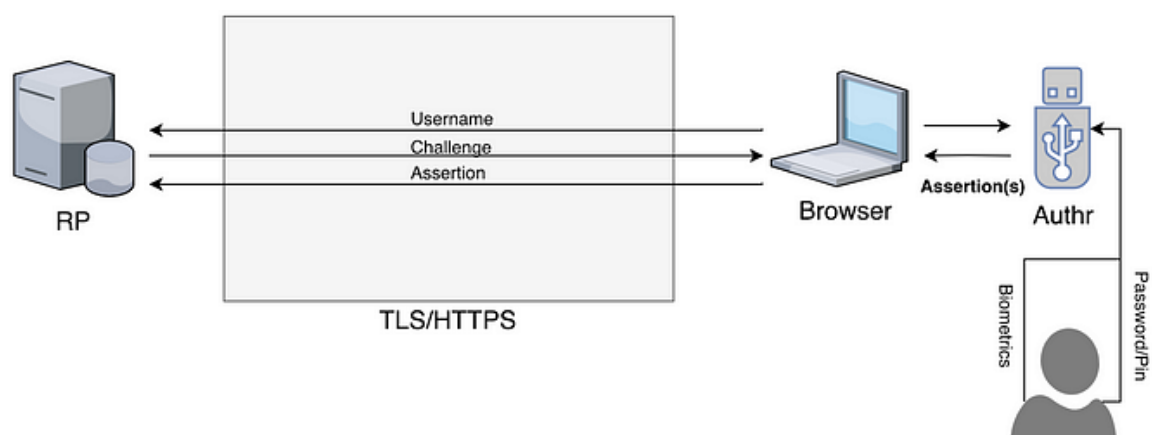
WebAuth – nazwa API w przeglądarce, wspierającego protokół FIDO2

Passkey – (klucz dostępu) – tożsamość użytkownika przechowana na urządzeniu lokalnym

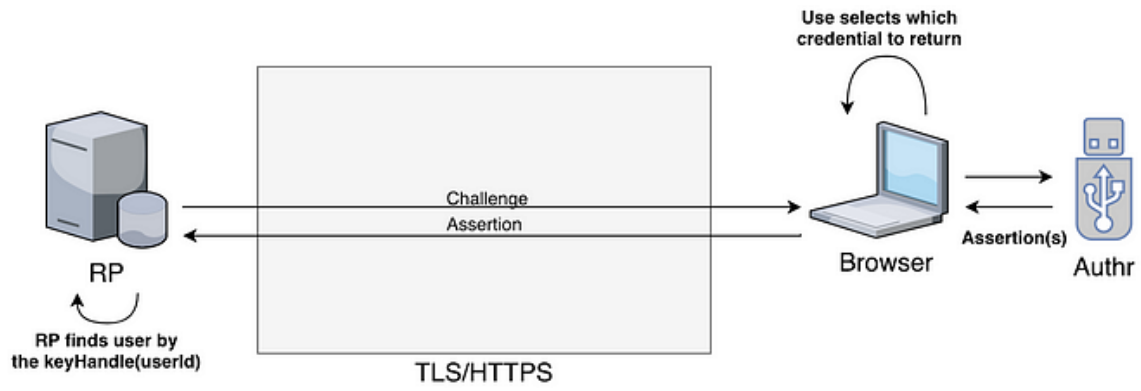
W trakcie wykładu opowiemy o tym standardzie i prześledzimy implementację przykładowego kodu dla .NET z <https://github.com/wzychla/Fido2.NetFramework> które jest portem bibliotek <https://github.com/passwordless-lib/fido2-net-lib>

Logowanie użytkownika w FIDO2 składa się z dwóch etapów.

Etap rejestracji klucza dostępu (tzw. **atestacja**) wykonywana raz gdy użytkownik jest zalogowany (jakoś inaczej niż za pomocą FIDO2)



Etap logowania (tzw. asercja) wykonywana przez anonimowego użytkownika który chce się zalogować



Zrzuty ekranu z artykułu **Introduction to WebAuthn API and Passkey**

<https://medium.com/webauthnworks/introduction-to-webauthn-api-5fd1fb46c285>