

Wybrane elementy praktyki projektowania oprogramowania

Wykład 10/15

node.js: Express (3)

Wiktor Zychla 2023/2024

1	Spis treści	
2	Autentykacja, autoryzacja	2
3	Middleware autentykacji, strona logowania	4
4	Implementacja modelu RBAC.....	8
5	Bezpieczna infrastruktura uwierzytelniania	10
5.1	Połączenie szyfrowane	10
5.2	Przechowywanie haseł	10
6	Uwierzytelnianie federacyjne.....	12
6.1	Protokół WS-Federation	12
6.2	Protokół OAuth2	14
6.3	Przykład	15

2 Autentykacja, autoryzacja

Autentykacja = proces rozpoznania tożsamości użytkownika

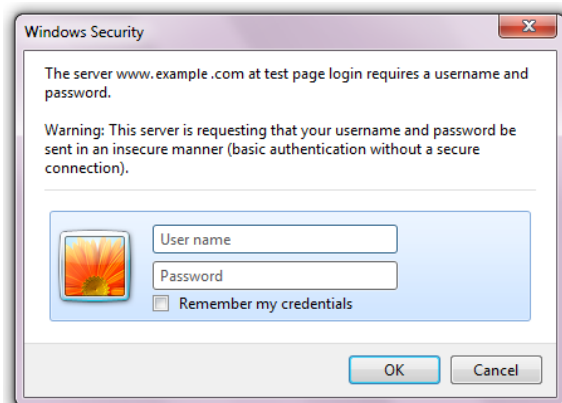
Autoryzacja = proces decyzyjny w którym użytkownikowi przyznaje się dostęp do zasobów lub zabrania się dostępu do zasobów

W praktyce, upraszczając, można powiedzieć że autentykacja jest *jakoś* związana z logowaniem, natomiast autoryzacja pozwala sterować dostępem do zasobów (np. „brak dostępu dla niezalogowanych” lub „dostęp tylko dla użytkowników w roli administratorzy” itp.)

Standardowym sposobem rozpoznania czy użytkownik jest zalogowany jest utrzymywanie przez aplikację **ciastka** zawierającego jakąś formę informacji o użytkowniku:

- Tylko nazwa użytkownika – w takim podejściu pozostałe informacje (np. role) są przez aplikację wyliczane (pobierane z bazy) przy każdym żądaniu
 - Wada – koszt dodatkowego wyliczania uprawnień przy każdym żądaniu
 - Zaleta – uprawnienia mogą się zmieniać użytkownikowi w trakcie pracy
- Nazwa użytkownika i dodatkowe informacje, np. **role**
 - Wada – brak możliwości zmiany uprawnień w trakcie pracy, użytkownik musi się wylogować i zalogować ponownie żeby aplikacja zauważyła dodatkowe uprawnienia
 - Wada – ograniczony rozmiar ciastka, jeśli użytkownik ma dużo ról może być z tym problem
 - Zaleta – brak dodatkowego kosztu sprawdzania uprawnień (alternatywą do przechowywania ról w ciastku logowania jest sprawdzanie ich przy każdym żądaniu)

Uwaga! Istnieje inny sposób podtrzymywania ciągłości sesji zalogowanego użytkownika niż ciastko. Ten sposób oparty jest o tzw. [401 Challenge](#) czyli mechanizm uwierzytelnienia wykorzystujący fragment specyfikacji protokołu http. Tym sposobem nie będziemy się zajmować ponieważ jest mniej wygodny dla użytkownika – formularz logowania jest wbudowany w przeglądarkę i programista nie ma możliwości jego stylowania:



Model bezpieczeństwa w którym użytkownik ma przypisaną **jedną lub wiele ról** (tzw. [Role-Based Access Control](#), RBAC) jest bodaj najczęściej implementowanym modelem autoryzacji.

W tym modelu każdej operacji którą można wykonać w systemie przypisany jest zbiór **rol** uprawnionych do wykonania tej operacji, a użytkowników przypisuje się do ról.

Przykład. W systemie są dwie role, **User** i **Admin**.

- Operacja listowania towarów w katalogu nie ma przypisanej roli (tę operację może wykonać każdy, nawet nieznany/niezałogowany użytkownik)
- Operacja dodania towarów do koszyka i złożenia zamówienia ma przypisaną rolę **User** – tylko załogowany użytkownik w roli **User** wykona tę operację
- Operacja dodawania towarów do listy towarów ma przypisaną rolę **Admin**

W systemie występują użytkownicy niezałogowani, użytkownicy w roli User, użytkownicy w roli Admin oraz użytkownicy będący równocześnie w rolach User i Admin.

Alternatywą dla modelu RBAC mógłby być m.in. model poziomów dostępu (np. model [Bell-LaPadula](#)). Bywa, że model RBAC jest implementowany niepoprawnie, na przykład rażącym niedopatrzeniem jest taka implementacja, w której użytkownikowi może być przypisana tylko jedna rola (sic!).

Wadą modelu RBAC jest podatność na tzw. eksplozję zbioru uprawnień, wystarczy wyobrazić sobie sytuację w której model jest implementowany w sytuacji, w której użytkownicy należą dodatkowo do jakichś jednostek organizacyjnych (jednej lub wielu) (na przykład **szkołach**) a role w poszczególnych jednostkach nie są równoważne (role „uczeń szkoły podstawowej 1” i „uczeń liceum 2” nie są równoważne).

W takim przypadku moc zbioru ról jest równa iloczynowi mocy zbioru „ról bez jednostki” i mocy zbioru jednostek organizacyjnych.

Przykład. Jeśli zbiór ról bez jednostki zawiera następujące role: uczeń, nauczyciel, rodzic, sekretarka, dyrektor, bibliotekarz, pedagog (łącznie 7) a zbiór jednostek ma moc 50 (szkoła podstawowa 1, szkoła podstawowa 2, liceum 3 itd.) to w systemie jest łącznie $7 * 50$ ról.

3 Middleware autentykacji, strona logowania

Klasyczne podejście do autentykacji wymaga tylko automatyzacji procesu decyzyjnego – czy żądanie należy obsłużyć czy też wymusić **przekierowanie** na stronę logowania. Architektura Express wychodzi tu naprzeciw – w definicji ścieżki może się pojawić bowiem nie jedno, ale **wiele** middleware, które są wykonywane po kolei.

Dzięki temu proces decyzyjny można wynieść do osobnego middleware, które następnie umieszcza się w definicji ścieżek wymagających logowania jako pierwsze:

```
/**
 *
 * @param {http.IncomingMessage} req
 * @param {http.ServerResponse} res
 * @param {*} next
 */
function authorize(req, res, next) {
  if ( req.signedCookies.user ) {
    req.user = req.signedCookies.user;
    next();
  } else {
    res.redirect('/login?returnUrl='+req.url);
  }
}
```

```
var http      = require('http');
var express   = require('express');
var cookieParser = require('cookie-parser');

var app = express();

app.use(express.urlencoded({ extended: true }));
app.use(cookieParser('sgs90890s8g90as8rg90as8g9r8a0srg8'));

app.set('view engine', 'ejs');
app.set('views', './views');

// wymaga logowania dlatego strażnik – middleware „authorize”
app.get( '/', authorize, (req, res) => {
  res.render('app', { user : req.user } );
});

app.get( '/logout', authorize, (req, res) => {
  res.cookie('user', '', { maxAge: -1 } );
  res.redirect('/')
});

// strona logowania
```

```

app.get( '/login', (req, res) => {
    res.render('login');
});

app.post( '/login', (req, res) => {
    var username = req.body.txtUser;
    var pwd      = req.body.txtPwd;

    if ( username == pwd ) {
        // wydanie ciastka
        res.cookie('user', username, { signed: true });
        // przekierowanie
        var returnUrl = req.query.returnUrl;
        res.redirect(returnUrl);
    } else {
        res.render( 'login', { message : "Zła nazwa logowania lub hasło" } );
    }
});

http.createServer(app).listen(3000);
console.log( 'serwer działa, nawiguj do http://localhost:3000' );

```

Do tego widoki:

```

<!-- login.ejs -->
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Document</title>
    <style>
    html, body, form {
        height: 100%;
        overflow: hidden;
    }

    form {
        display          : flex;
        justify-content  : center;
        align-items      : center;
    }

    #login {
        padding          : 20px;
        border            : 1px solid black;
    }

```

```

#login div {
    overflow : auto;
}

#login input {
    float : right;
}

button {
    clear : both;
}

.message {
    color : red;
}
</style>
</head>
<body>
    <form method="POST">
        <div id='login'>
            <div>
                Logowanie
            </div>
            <div>
                <input type='text' name='txtUser' />
                <label>Nazwa użytkownika:</label>
            </div>
            <div>
                <input type='password' name='txtPwd' />
                <label>Hasło:</label>
            </div>
            <div>
                <button>Zaloguj</button>
            </div>
            <% if ( locals.message ) { %>
                <div class='message'>
                    <%= locals.message %>
                </div>
            <% } %>
        </div>
    </form>
</body>
</html>

```

```

<!-- app.ejs -->
<!DOCTYPE html>
<html lang="en">
<head>

```

```
<meta charset="UTF-8">
<title>Document</title>
</head>
<body>
  Witaj, jesteś zalogowany jako <%= user %>. <a href='/logout'>Wyloguj
  się</a>
</body>
</html>
```

Przy okazji warto zwrócić uwagę na kilka typowych technik:

- Do przechowania danych użyte jest ciastko typu **signed**, bez tego użytkownik mógłby sam tworzyć udawane ciastka pozwalające mu dostać się do obcych sesji (!)
- Wylogowanie to po prostu usunięcie ciastka (formalnie: wydanie ciastka z przeszłą datą ważności)
- Warunkowe renderowanie całej sekcji (informacja o błędnym logowaniu) jest możliwe przy użyciu aliasu **locals**
- Centrowanie widoku możliwe jest na wiele sposobów – tu został użyty tzw. [CSS flex layout](#)

4 Implementacja modelu RBAC

Zaprezentowane w poprzednim rozdziale proste middleware autentykacji nie pozwala rozróżniać użytkowników należących do różnych grup zabezpieczeń. Prosta modyfikacja dodaje taką możliwość:

```
/**
 * Sprawdzenie czy użytkownik o podanej nazwie jest w roli o podanej nazwie.
 * Możliwe są różne implementacje np.:
 * * zapisanie ról w ciastku, razem z nazwą użytkownika i sprawdzanie w tym
ciastku
 * * sprawdzanie za każdym razem w bazie danych (lub innym magazynie
trwałym)
 */
function isUserInRole(user, role) {
  // trywialna implementacja na potrzeby demonstracyjne
  return user == role;
}

/**
 *
 * @param {http.IncomingMessage} req
 * @param {http.ServerResponse} res
 * @param {*} next
 */
function authorize(...roles) {
  return function(req, res, next) {
    if ( req.signedCookies.user ) {
      let user = req.signedCookies.user;
      if ( roles.length == 0 ||
        roles.some( role => isUserInRole( user, role ) )
      ) {
        req.user = user;
        return next();
      }
    }
    // fallback na brak autoryzacji
    res.redirect('/login?returnUrl='+req.url);
  }
}
```

W takiej wersji middleware autentykacji wymaga opcjonalnego określenia listy ról uprawnionych do dostępu do podanej funkcji:

```
app.get( '/', authorize(), (req, res) => {
  res.render('app', { user : req.user } );
});

app.get( '/logout', authorize(), (req, res) => {
```



```
    res.cookie('user', '', { maxAge: -1 } );  
    res.redirect('/')  
});  
  
// strona tylko dla administratora  
app.get( '/admin', authorize('admin'), (req, res) => {  
    res.setHeader('Content-type', 'text/html; charset=utf-8');  
    res.write('witaj administratorze');  
    res.end();  
})
```

5 Bezpieczna infrastruktura uwierzytelniania

Aby tak klasycznie zbudowana aplikacja nie padła łupem internetowych włamywaczy, musi być spełniony szereg warunków. Wymienimy wybrane z nich:

5.1 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.

5.2 Przechowywanie haseł

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

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

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](#)).

```
var bcrypt = require('bcrypt');

(async function() {

  // hasło użytkownika
  var password = 'Foo.Bar.123';
  var rounds   = 12;

  var hash = await bcrypt.hash(password, rounds );
```

```
// wynik hashowania zapisuje się w bazie danych
console.log( hash );

// weryfikacja
var attemptedPassword = 'Foo.Bar';
var result = await bcrypt.compare( attemptedPassword, hash );

console.log( result ); // false

attemptedPassword      = 'Foo.Bar.123';
result = await bcrypt.compare( attemptedPassword, hash );

console.log( result ); // true

})();
```

6 Uwierzytelnianie federacyjne

Przechowywanie haseł niezależnie w wielu aplikacjach naraża infrastrukturę na dodatkowe ryzyka. Dlatego współcześnie często rozważa się tzw. [uwierzytelnianie federacyjne](#) oraz protokoły [pojedynczego logowania](#) (SSO).

Uwierzytelnianie federacyjne = przepływ kontroli między aplikacją (**Service Provider**) a inną aplikacją (dostawcą usługi jednokrotnego logowania, **Identity Provider**), w której SP deleguje uwierzytelnianie użytkowników do IdP.

Dzięki temu SP w ogóle nie musi przechowywać haseł, implementować polityk wygasania, złożoności, obsługi dwuskładnikowego uwierzytelniania itd. itp.

Technicznie, „delegowanie” oznacza, że zamiast wyświetlać formularz logowania, SP **przekierowuje** przeglądarkę na punkt końcowy (stronę) w ramach aplikacji IdP, IdP uwierzytelnia użytkownika i IdP **zwraca** do SP informację o tym kto się zalogował.

Owo „zwrócenie” informacji o tym kto jest zalogowany, z IdP do SP, jest bodaj najciekawszą częścią takiego przepływu. Należy zwrócić uwagę na następujące elementy:

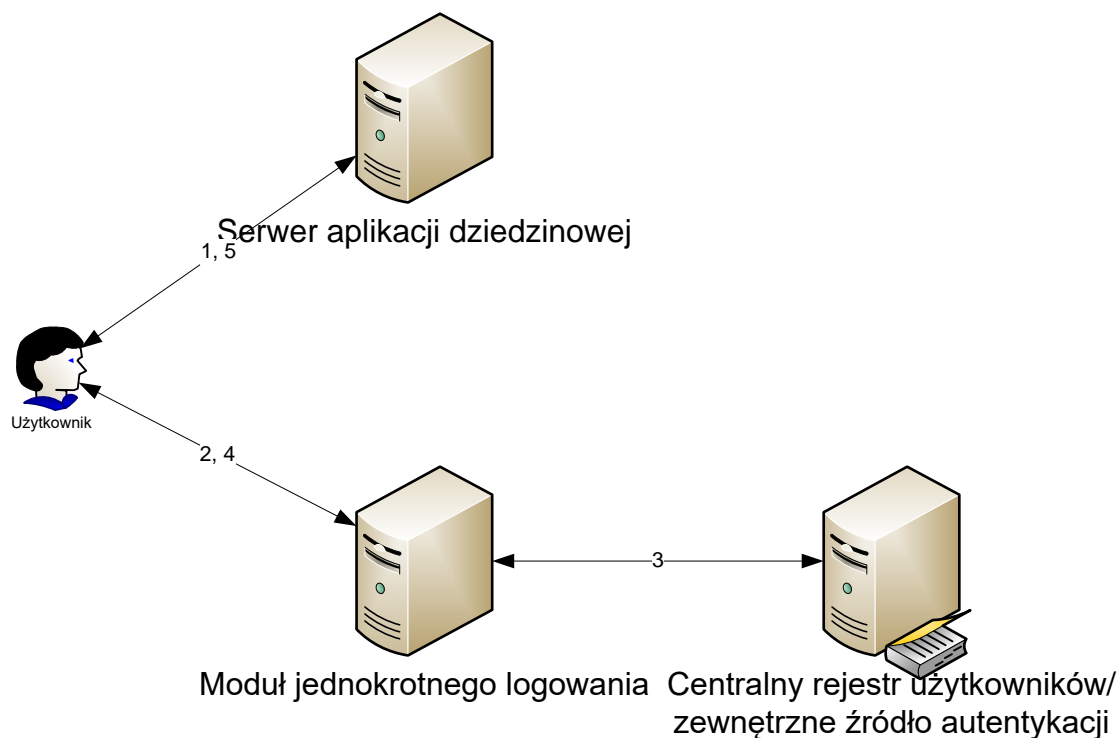
- Informacja musi przepłynąć z IdP do SP w taki sposób, żeby użytkownik nie mógł „oszukać” przepływu, czyli podszyć się pod innego użytkownika
- Informacja musi przepłynąć w nietrywialny sposób, w szczególności – IdP i SP są **osobnymi** aplikacjami internetowymi, dostępnymi pod różnymi adresami, i jako takie **nie współdzielą ciasteczek**.

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

- protokół **passive** [WS-Federation](#), który 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 – jest to duża zaleta, otwierająca perspektywę łatwej rozbudowy systemu o kolejne moduły w przyszłości.
- Protokół [OAuth2/OpenID Connect](#), szeroko implementowany przez dostawców usług społecznościowych

6.1 Protokół WS-Federation

Rysunek 1 przedstawia schemat poświadczania tożsamości między SP a IdP przy wykorzystaniu WS-Federation



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

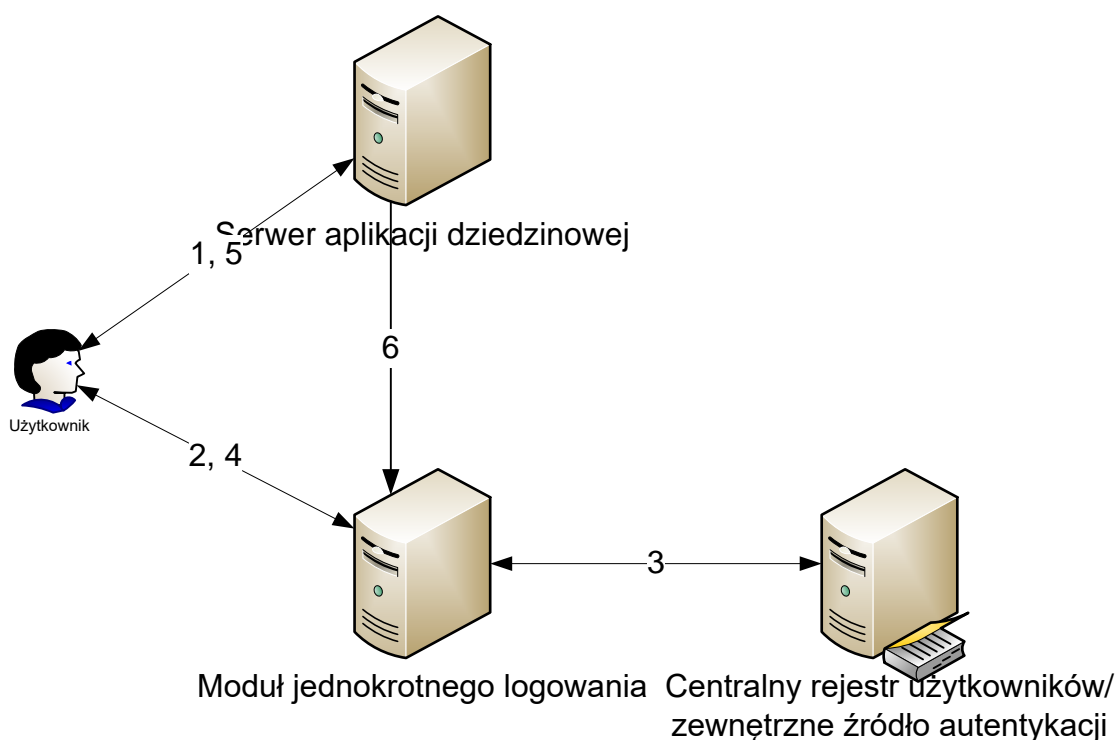
Szczegółowa dokumentacja techniczna protokołu autentykacji WS-Federation, zawartości i sposobu interpretacji tokenów SAML są publicznie dostępne i nie zostaną dołączone do niniejszego opracowania.

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.

6.2 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

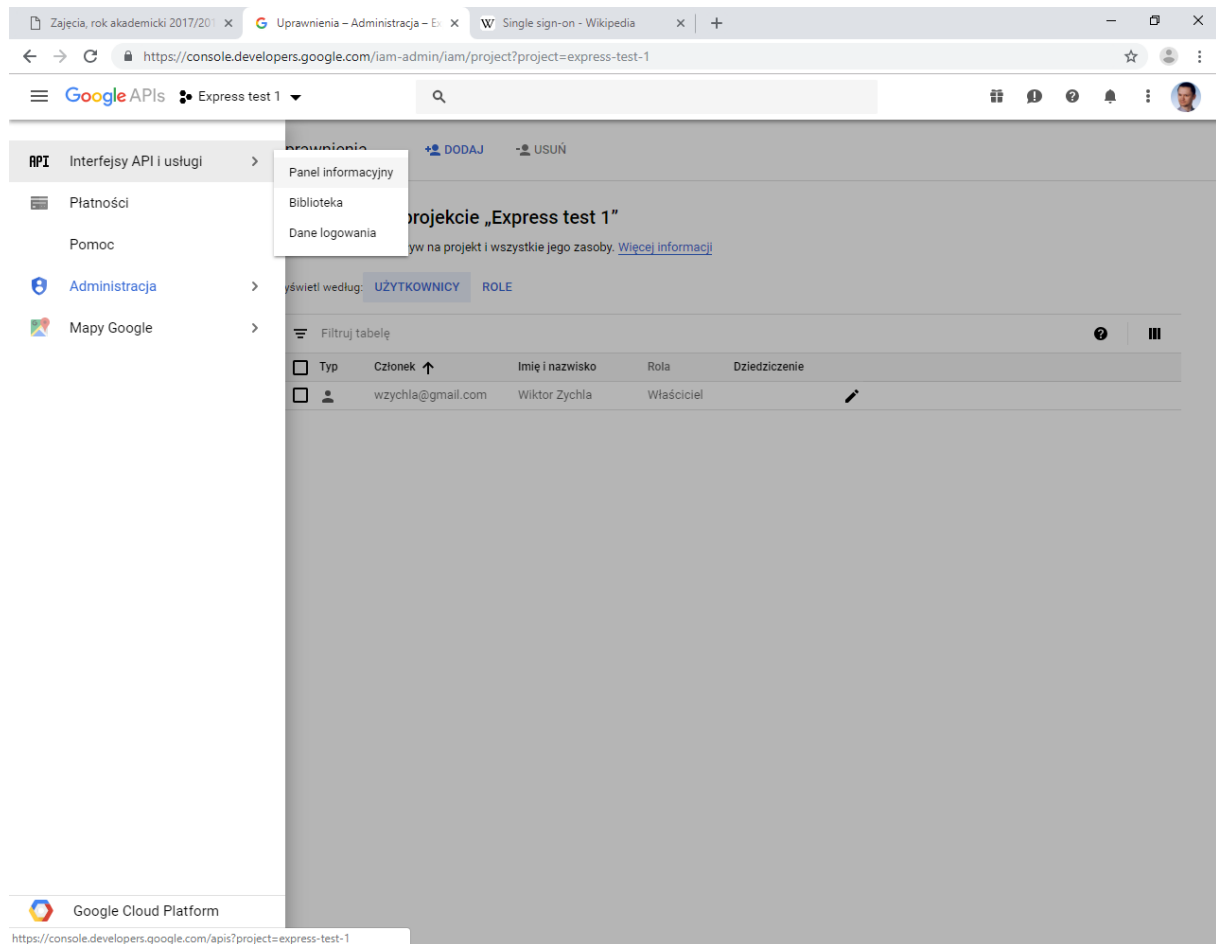
Poszczególne kroki protokołu przedstawiają się następująco:

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
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 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

6.3 Przykład

Zbudujemy aplikację uwierzytelniającą się w Google za pomocą protokołu OAuth2. Pierwszym krokiem jest rejestracja aplikacji w Google w [konsoli dla developerów](#).

W konsoli należy utworzyć nową aplikację i przywołać widok Interfejsy API i usługi:

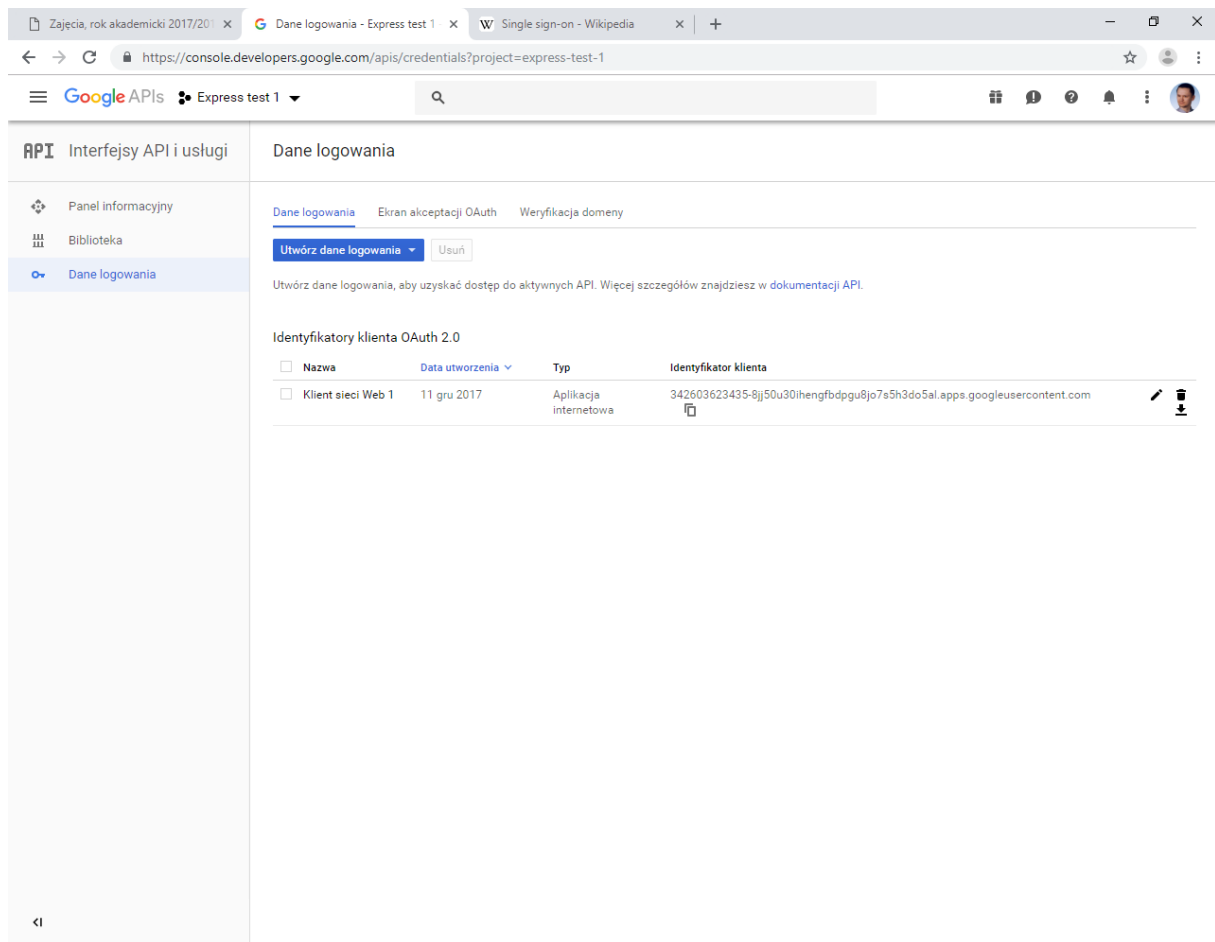


Tu należy pozyskać dane logowania, czyli

- **identyfikator klienta**
- **tajny klucz klienta**

oraz zarejestrować adres powrotny, w którym w aplikacji odbędzie się przetworzenie tokena federacyjnego i zamiana go na informacje o użytkowniku – w przykładzie adres zwrotny (callback) który należy podać w panelu konfiguracyjnym to <http://localhost:3000/callback>

Oba parametry danych logowania (identyfikator klienta i tajny klucz klienta) muszą być wstawione do kodu aplikacji. Identyfikator klienta jest publiczny (jest częścią adresu przekierowania), tajny klucz klienta nigdy nie powinien wyciekać na zewnątrz.



oraz na liście API **włączyć** API dla Google+ aby umożliwić aplikacji dostęp do API zwracającego informacje o profilu.

Na formularzu logowania pojawi się odnośnik umożliwiający logowanie za pomocą Google:

```
<!-- login.ejs -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Document</title>
  <style>
    html, body, form {
      height: 100%;
      overflow: hidden;
    }

    form {
      display          : flex;
      justify-content  : center;
      align-items      : center;
    }

    #login {
```



```

        padding      : 20px;
        border        : 1px solid black;
    }

    #login div {
        overflow : auto;
    }

    #login input {
        float : right;
    }

    button {
        clear : both;
    }

    .message {
        color : red;
    }

    div > label {
        width: 120px;
    }
</style>
</head>
<body>
    <form method="POST">
    <div id='login'>
        <div>
            Logowanie
        </div>
        <div>
            <input type='text' name='txtUser' />
            <label>Nazwa użytkownika:</label>
        </div>
        <div>
            <input type='password' name='txtPwd' />
            <label>Hasło:</label>
        </div>
        <div>
            <button>Zaloguj</button>
        </div>
        <div>
            <a href='<%- google %>'>Zaloguj za pomocą Google</a>
        </div>
        <% if ( locals.message ) { %>
            <div class='message'>
                <%= locals.message %>
            </div>
        </div>
    </form>
</body>
</html>

```

```
        <% } %>
    </div>
</form>
</body>
</html>
```

a za obsługę protokołu OAuth2 będzie odpowiadał moduł **simple-oauth2**.

Dodatkowo w poniższym kodzie użyto kilku pakietów, o których do tej pory nie mówiliśmy:

- **node-fetch** – ułatwia tworzenie żądań http/https które wychodzą z serwera (do innego serwera)
- **jsonwebtoken** – pakiet do parsowania i walidacji tokenów JWT (dostawca OAuth2 zwraca informacje o użytkowniku w formacie JWT)
- **jwt-rsa** – pakiet do pozyskania klucza publicznego do walidacji JWT, opublikowanego w sieci w formacie JWKS (JSON Web Key Set)

Więcej informacji technicznych od konkretnego dostawcy (tu: Google) na [stronie dokumentacji](#).

```
/**
 * WEPP0 2022
 * Przykład autentykacji za pomocą OAuth2
 *
 * Aplikacja zadziała tylko po wcześniejszym zarejestrowaniu jej w Google i
 * odblokowaniu dostępu do Google+ API
 *
 * Więcej: http://www.wiktorzychla.com/2014/11/simple-oauth2-federated-
authentication.html
 *
 * Po zarejestrowaniu aplikacji należy przepisać jej id i secret z zakładki
 * Interfejsy API i usługi / Dane logowania do pól id/secret poniżej
 */
var http = require('http');
var https = require('https');
var fetch = require('node-fetch');
var jwt = require('jsonwebtoken');
var jwksClient = require('jwks-rsa');
var express = require('express');
var cookieParser = require('cookie-parser');
var { AuthorizationCode } = require('simple-oauth2');

// parametry id i secret należy przepisać ze strony konfiguracyjnej
// w Google
const oauth2 = new AuthorizationCode({
  client: {
    id: '947413312885-
37sp8f7m4a42ahmtesjso5938fqh09n4.apps.googleusercontent.com',
    secret: 'mw_...wN',
  },
```

```

    auth: {
      tokenHost: 'https://www.googleapis.com',
      tokenPath: '/oauth2/v4/token',
      authorizeHost: 'https://accounts.google.com',
      authorizePath: '/o/oauth2/v2/auth'
    },
  });

const authorizationUri = oauth2.authorizeURL({
  redirect_uri: 'http://localhost:3000/callback',
  scope: 'openid profile email'
});

var app = express();

app.use(express.urlencoded({ extended: true }));
app.use(cookieParser('sgs90890s8g90as8rg90as8g9r8a0srg8'));

app.set('view engine', 'ejs');
app.set('views', './views');

// wymaga logowania
app.get('/', authorize, (req, res) => {
  res.render('app', { user: req.user });
});

app.get('/logout', authorize, (req, res) => {
  res.cookie('user', '', { maxAge: -1 });
  res.redirect('/')
});

// strona logowania
app.get('/login', (req, res) => {
  res.render('login', { google: authorizationUri });
});

app.post('/login', (req, res) => {
  var username = req.body.txtUser;
  var pwd = req.body.txtPwd;

  if (username == pwd) {
    // wydanie ciastka
    res.cookie('user', username, { signed: true });
    // przekierowanie
    var returnUrl = req.query.returnUrl;
    res.redirect(returnUrl);
  } else {
    res.render('login', { message: "Zła nazwa logowania lub hasło",
google: authorizationUri });
  }
});

```

```

    }
  });

app.get('/callback', async (req, res) => {

  const code = req.query.code;
  const options = {
    code,
    redirect_uri: 'http://localhost:3000/callback'
  };

  // żądanie do punktu końcowego oauth2 zamieniające code na access_token
  // i id_token
  var result = await oauth2.getToken(options)

  // teraz są dwie możliwości
  // 1. użyć access_token żeby zapytać serwera kto kryje się pod wskazaną
  // tożsamością
  // 2. użyć id_token gdzie od razu zapisana jest wskazana tożsamość
  var access_token = result.token.access_token;
  var id_token = result.token.id_token;

  // wariant 1. - żądanie do usługi profile API Google+ po profil
  // użytkownika
  /*
  var response =
    await fetch('https://openidconnect.googleapis.com/v1/userinfo',
      { headers: {
        "Authorization": `Bearer
    ${encodeURIComponent(access_token)}`
      }});
  var profile = await response.json();
  if (profile.email) {
    // zalogowanie
    res.cookie('user', profile.email, { signed: true });
    res.redirect('/');
  }
  */

  // wariant 2. - tożsamość bez potrzeby dodatkowego żądania
  // Uwaga! Formalnie token JWT należy zweryfikować, posługując się
  // kluczami publicznymi
  // z https://www.googleapis.com/oauth2/v3/certs
  var client = jwksClient({
    jwksUri: 'https://www.googleapis.com/oauth2/v3/certs'
  });
  function getKey(header, callback){
    client.getSigningKey(header.kid, function(err, key) {
      var signingKey = key.publicKey || key.rsaPublicKey;

```

```

        callback(null, signingKey);
    });
}
var profile = jwt.verify(id_token, getKey, (err, profile) =>{
    if (profile.email) {
        // zalogowanie
        res.cookie('user', profile.email, { signed: true });
        res.redirect('/');
    }
});

// ... ale gdyby pominąć walidację, to kod się upraszcza do
//var profile = jwt.decode(id_token);
//if (profile.email) {
//    // zalogowanie
//    res.cookie('user', profile.email, { signed: true });
//    res.redirect('/');
//}
});

// middleware autentykacji
function authorize(req, res, next) {
    if (req.signedCookies.user) {
        req.user = req.signedCookies.user;
        next();
    } else {
        res.redirect('/login?returnUrl=' + req.url);
    }
}

http.createServer(app).listen(3000);
console.log('serwer działa, nawiguj do http://localhost:3000');

```