

Wybrane elementy praktyki projektowania oprogramowania
Wykład 10/15
node.js: Express (3)

Wiktor Zychła 2019/2020

1	Spis treści	
2	Autentykacja, autoryzacja	2
3	Middleware autentykacji, strona logowania	3
4	Bezpieczna infrastruktura uwierzytelniania	7
4.1	Połączenie szyfrowane	7
4.2	Przechowywanie haseł	7
5	Uwierzytelnianie federacyjne	8
5.1	Protokół WS-Federation	8
5.2	Protokół OAuth2	10
5.3	Przykład	11

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 wyliczania uprawnień

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:



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
- 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 Bezpieczna infrastruktura uwierzytelniania

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

4.1 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 i doklejać je do preparowanych żądań do serwera. Dlatego obecnie zdecydowanie sugeruje się kanał szyfrowany do całej aplikacji.

4.2 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 przez 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, wykluczający atak słownikowy
- 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 (np. 50-500 ms) ale odwracanie – wtedy odpowiednio trudniejsze.

W praktyce stosuje się algorytmy [bcrypt](#) lub równoważne ([PBKDF2](#)).

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

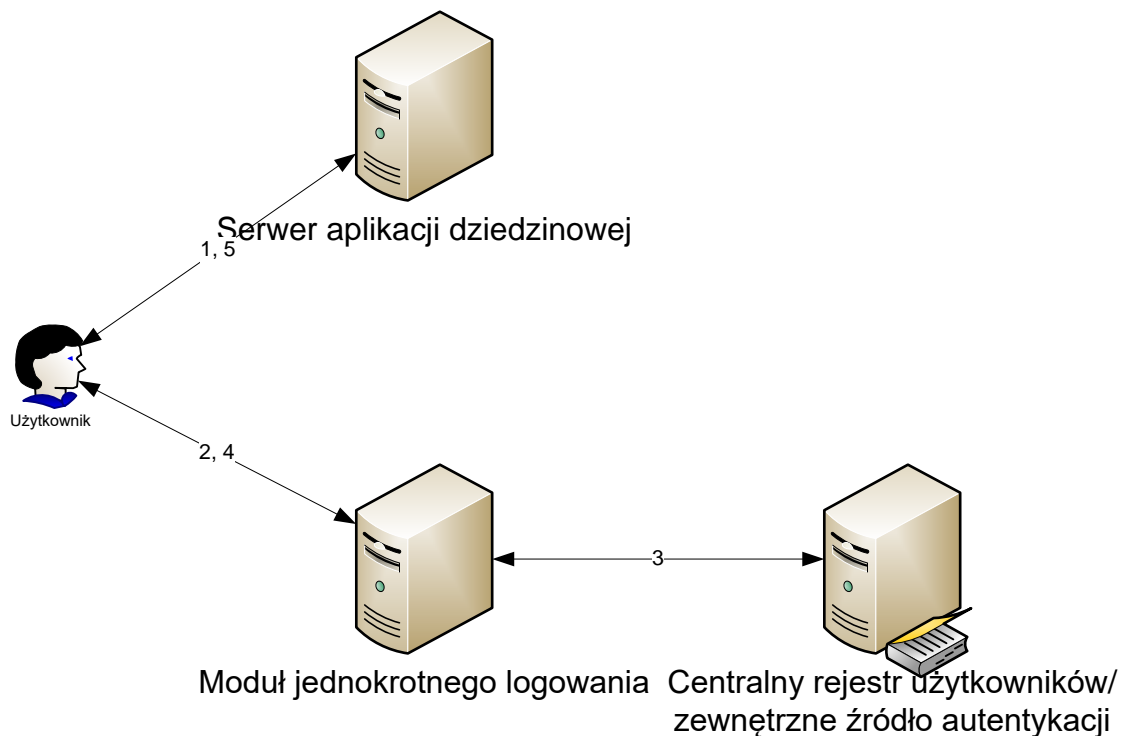
Z tego powodu współcześnie korzysta się z tzw. protokołów SSO, np.:

- 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.

5.1 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 poświadczają 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

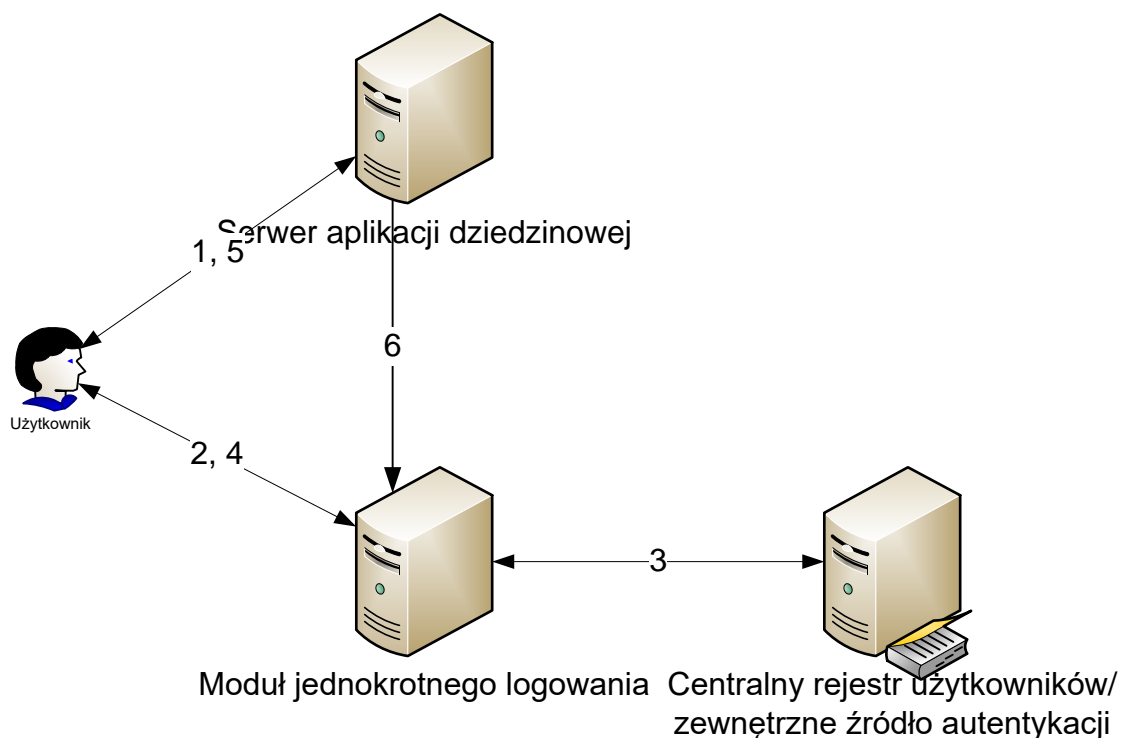
Szczegółowa dokumentacja techniczna protokołu autentykacji WS-Federation, zawartości i sposobu interpretacji tokenów SAML są publicznie dostępne i nie zostaną dołączone do niniejszego opracowania.

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.

5.2 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

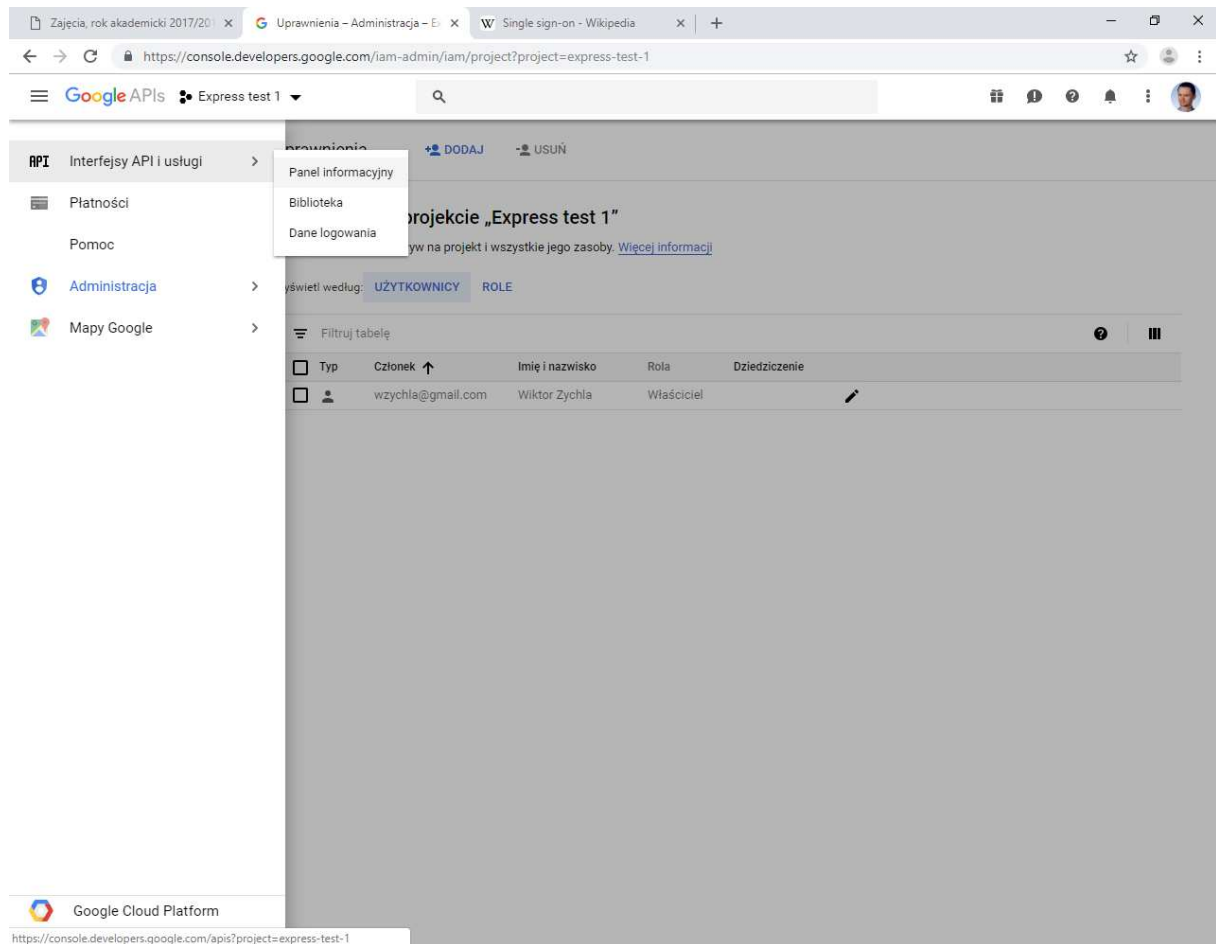
Poszczególne kroki protokołu przedstawiają się następująco:

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

5.3 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 aplikacji i tajny klucz oraz zarejestrować adres powrotny, w którym w aplikacji odbędzie się przetworzenie tokena federacyjnego i zamiana go na informacje o użytkowniku – w przykładzie adres zwrotny to <http://localhost:3000/callback>

The screenshot shows the Google Cloud Platform console interface. The browser address bar displays the URL: `https://console.developers.google.com/apis/credentials?project=express-test-1`. The page title is "Dane logowania" (Credentials). The left sidebar contains navigation options: "Panel informacyjny", "Biblioteka", and "Dane logowania". The main content area is titled "Dane logowania" and includes tabs for "Dane logowania", "Ekran akceptacji OAuth", and "Weryfikacja domeny". There is a button "Utwórz dane logowania" and a "Usuń" button. Below this, there is a table of OAuth 2.0 client IDs:

<input type="checkbox"/>	Nazwa	Data utworzenia	Typ	Identyfikator klienta	
<input type="checkbox"/>	Klient sieci Web 1	11 gru 2017	Aplikacja internetowa	342603623435-8jj50u30ihengfbdpgu8jo7s5h3do5al.apps.googleusercontent.com	

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>
    </if %>
  </div>
  </form>
</body>
</html>

```

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

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

```
/**
 * WEPPPO 2017
 * 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 express = require('express');
var cookieParser = require('cookie-parser');
var simpleOAuthModule = require('simple-oauth2');

const oauth2 = simpleOAuthModule.create({
  client: {
    id: '342603623435-8jj50u30ihengfbdpgu8jo7s5h3do5a1.apps.googleusercontent.com',
    secret: '...tu skopiować secret...',
  },
  auth: {
    tokenHost: 'https://www.googleapis.com',
    tokenPath: '/oauth2/v4/token',
    authorizeHost: 'https://accounts.google.com',
    authorizePath: '/o/oauth2/v2/auth'
  },
});

const authorizationUri = oauth2.authorizationCode.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', (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
  oauth2.authorizationCode.getToken(options, (error, result) => {
    if (error) {
      return res.end(`Błąd: ${error}`);
    }
  });
});

```

```

    }
    const token = oauth2.accessToken.create(result);

    // żądanie do usługi profile API Google+ po profil użytkownika
    var opts = {
      host: 'openidconnect.googleapis.com',
      path: '/v1/userinfo',
      headers: {
        "Authorization": `Bearer
${encodeURIComponent(token.token.access_token)}`
      }
    }
    https.get(opts, profileRes => {
      var profileJson = '';
      profileRes.on('data', data => {
        profileJson += data.toString()
      });
      profileRes.on('end', () => {
        var profile = JSON.parse(profileJson);
        if (profile.email) {
          // zalogowanie
          res.cookie('user', profile.email, { signed: true });
          res.redirect('/');
        } else {
          // obsługa błędu
          if (profile.error) {
            res.end(profile.error);
          } else {
            res.end(`Błąd żądania do usługi profile API`);
          }
        }
      }
    });
  });
});
});

// 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');

```