MEAN Stack - uwierzytelnienie

Tworzenie serwisów Web 2.0

dr inż. Robert Perliński
rperlinski@icis.pcz.pl

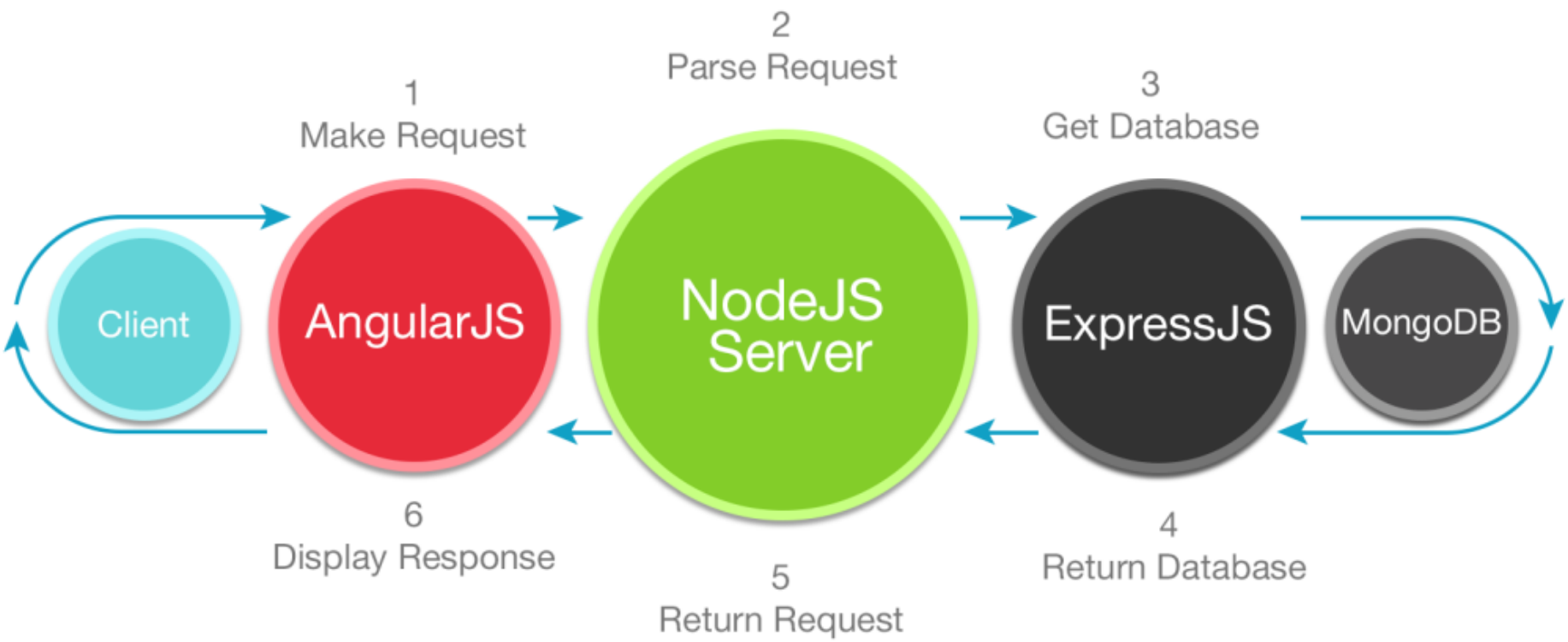
Politechnika Częstochowska
Instytut Informatyki Teoretycznej i Stosowanej

22 marca 2019

Plan prezentacji

1. MEAN Stack, ExpressJS - logowanie
2. Obsługa sesji w express
3. Uwierzytelnienie i autoryzacja
4. Uwierzytelnienie z passport
   - Instrukcja krok po kroku
5. Tutoriale
6. Źródła

MEAN Stack


2
Parse Request
1
Make Request
3
Get Database
Client
AngularJS
NodeJS
Server
ExpressJS
MongoDB
6
Display Response
5
Return Request
4
Return Database

ExpressJS i warstwy pośrednie

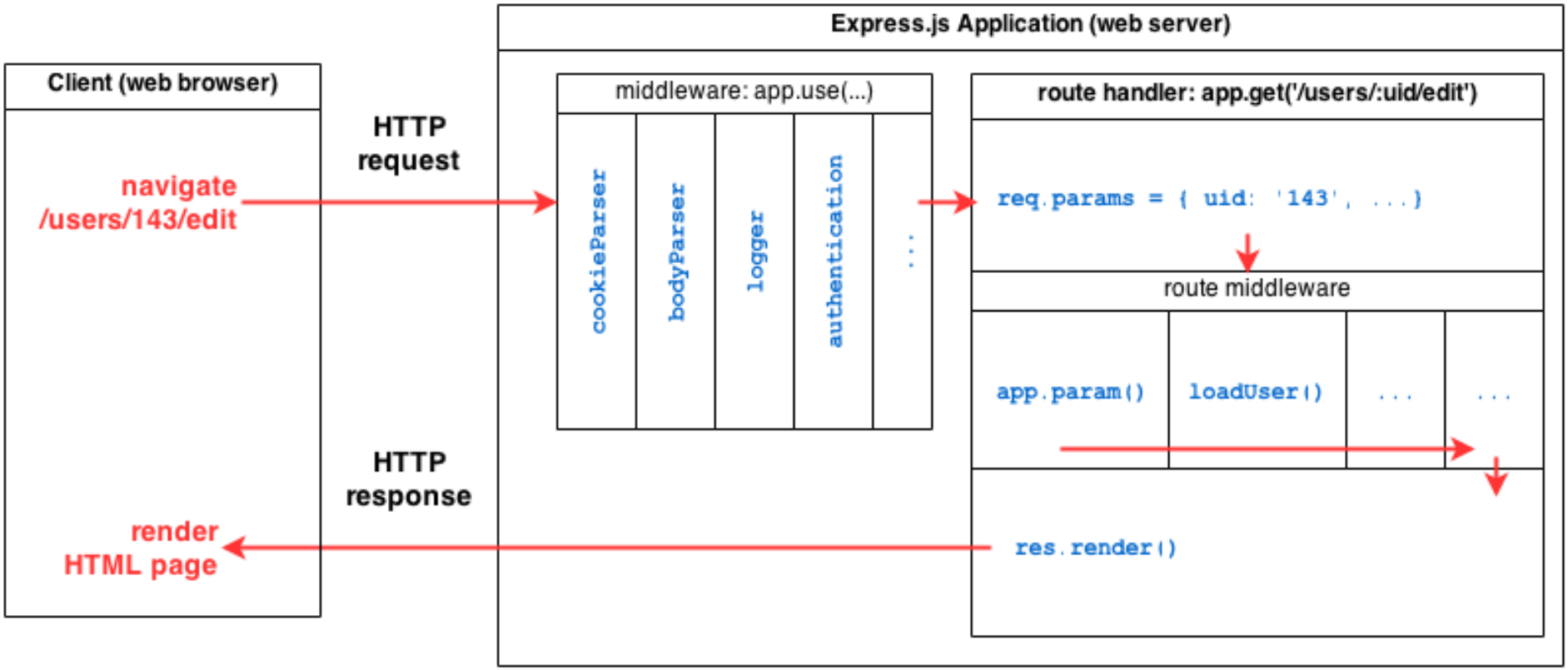
- ExpressJS to komplenty framework.
- Działa z szablonami (jade, ejs, handlebars, hogan.js)
- i kompilatorami CSS (less, stylus, compass).
- Dzięki wartstwom pośredniczącym (oprogramowanie pośredniczące) obsługuje również: ciasteczka, sesje, buforowanie, CSRF, kompresję i wiele innych.

Oprogramowanie pośredniczące:

- łańcuch/stos programów przetwarzających każde żądanie do serwera.
- Takich programów w stosie może być dowolna liczba,
- przetwarzanie odbywa się jeden po drugim.
- Niektóre z nich mogą zmieniać żądanie, tworzyć logi czy inne dane,
- przekazywać je do następnych (`next()`) programów w strumieniu.

Express-middlewares

Warstwy pośredniczące dodajemy do ExpressJS używając `app.use` dla dowolnej metody albo `app.VERB` (np. `app.get`, `app.delete`, `app.post`, `app.update`, ...)



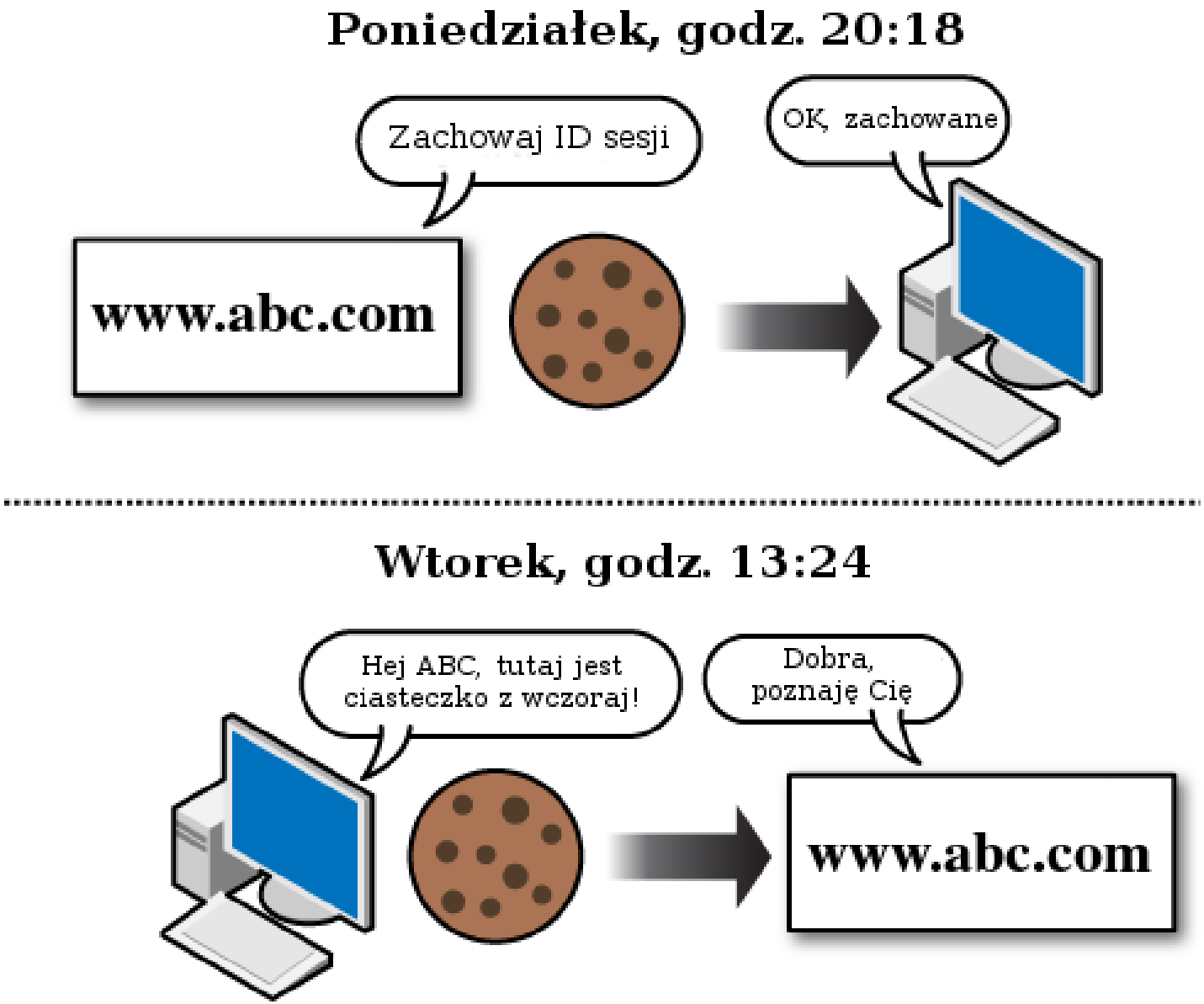
Sesja przeglądarki - bezstanowość HTTP

- Protokół HTTP jest protokołem bezstanowym.
- Nie posiada wbudowanego sposobu utrzymywania stanu pomiędzy dwiema transakcjami.
- Kiedy użytkownik żąda jednej strony, a później następnej, HTTP nie jest w stanie określić, czy oba żądania pochodzą od tego samego użytkownika.
- Ideą kontroli sesji jest zapewnienie możliwości śledzenia użytkownika podczas pojedynczej sesji w witrynie WWW.
- Umożliwia ona dokonanie wielu operacji:
  - łatwą obsługę logowania użytkowników,
  - wyświetlanie zawartości zależnie od poziomu uwierzytelnienia lub osobistych preferencji,
  - śledzenie zachowań użytkownika,
  - zaimplementowanie koszyków na zakupy itd.

Sesja przeglądarki

- Sesja zapamiętuje przez pewien czas na serwerze szczegóły dotyczące połączenia z klientem.
- W uproszczeniu to tablica zmiennych, przypisywanych do sesji przeglądarki.
- Otwarcie strony w przeglądarce powoduje utworzenie przez serwer sesji i załadowanie najnowszej wersji strony.
- Do przeglądarki wysyłane jest ciasteczko z unikalnym numerem sesji.
- Zmienne są przechowywane po stronie serwera.
- Identyfikator sesji jest jedyną informacją widoczną po stronie klienta.

Sesja przeglądarki



Sesje w express

Pakiet **express-session**

- Tworzy warstwę pośrednią według podanych opcji: `session(opcje)`
- Dane sesyjne same z siebie nie są zapisywane w ciasteczkach, zapisywane jest tylko ID sesji.
- Dane sesyjne przechowywane są po stronie serwera.
- Domyślnie dane są przechowywane w pamięci serwera (`MemoryStorage`) - jest to rozwiązanie tylko do tworzenia aplikacji, nie do wersji produkcyjnej.
- Właściwe metody przechowywania danych sesyjnych opierają się na osobnych modułach, głównie na bazach danych:
  - `connect-mongo` - Zapis sesji w oparciu o MongoDB
  - `connect-pg-simple` - Sesje w oparciu o PostgreSQL
  - `connect-sqlite3` - Sesja z użyciem SQLite3
  - ...
  - `session-file-storage` - Sesje zapisywane w systemie plików

Opcje sesji

Opcje tworzenia danych sesyjnych:

- **cookie** - ustawienia dla ciasteczka session ID, wartość domyślna:
  `{ path: '/', httpOnly: true, secure: false, maxAge: null }`
  - `path` - ścieżka dla domeny, której będzie dotyczyło ciasteczko, domyślna wartość to `'/'` czyli cała domena,
  - `secure` - domyślnie jest `false`, jeśli `true`, to wymaga to połączenia po HTTPS, jeśli serwer nie obsługuje HTTPS to klient nie wyśle później ciasteczka do serwera,
  - `maxAge` - liczba milisekund do utraty ważności ciasteczka, domyślnie jest `null` czyli sesja jest na okres życia przeglądarki,
  - `httpOnly` - wartość `true` nie pozwala na dostęp do ciasteczek z posiomu JavaScript przeglądarki,
  - `domain` - określa wartość domeny dla ciasteczka, domyślnie nieustawione - przeglądarki uznają, że ciasteczko odnosi się do bieżącej domeny,
  - `expires` - określa datę ważności ciasteczka, domyślnie nieustawione - większość przeglądarek uzna, że cisteczko jest tylko na okres życia sesji,

Opcje sesji

Opcje tworzenia danych sesyjnych:

- **genid** - określa funkcję wykorzystywaną do generowania ID sesji; funkcja powinna zwracać string, który będzie użyty jako session ID; domyślna funkcja używa biblioteki **uid-safe**

```
app.use(session({
  genid: function(req) {
    return genuuid() // use UUIDs for session IDs
  },
  secret: 'jakies haslo'
}))
```

- **name** - nazwa ciasteczka zawierającego ID sesji, które ustawiamy w odpowiedzi (res) albo odczytujemy w żądaniu (req); domyślna wartość: `'connect.sid'`, przy wielu aplikacjach na tym samym serwerze (ta sama nazwa hosta) trzeba używać różnych nazw

Opcje sesji

Opcje tworzenia danych sesyjnych:

- **proxy** - zaufaj serwerowi proxy przy tworzeniu bezpiecznych ciasteczek (przez nagłówek "X-Forwarded-Proto"); wartość domyślna: `undefined`, możliwe wartości:
  - `true` - nagłówek "X-Forwarded-Proto" będzie użyty,
  - `false` - wszystkie nagłówki są ignorowane, połączenie jest uważane za bezpieczne tylko jeśli jest bezpośrednie połączenie TSL/SSL
  - `undefined` - używa ustawienia "trust proxy" z express'a
- **resave** - wymusza zapisanie sesji, nawet jeśli nie była modyfikowana podczas żądania; wartość domyślna: `true`, zwykle ustawiana na `false`.
- **rolling** - wymusza ustawienie ciasteczka z ID sesji przy każdej odpowiedzi serwera; ważność ciasteczka jest odświerzana na `maxAge` (resetowanie jej wartości); domyślna wartość: `false`,

Opcje sesji

Opcje tworzenia danych sesyjnych:

- **saveUninitialized** - wymusza zapis niezainicjowanej sesji czyli takiej, która jest nowa i nie została jeszcze zmodyfikowana, domyślna wartość: `true`, wartość `false` dobrze sprawdza się przy sesji związanej z logowaniem,
- **secret** - **wymagana opcja**, sekret używany do oznaczenia ciasteczka z ID sesji; może być pojedyńczym obiektem string albo tablicą,
- **store** - określa sposób przechowywania sesji, domyślnie `MemoryStore`,
- **unset** - określa postępowanie przy wyczyszczeniu `req.session`:
  - `'destroy'` - sesja będzie usunięta po zakończeniu odpowiedzi na żądanie,
  - `'keep'` - domyślna wartość, sesja będzie zachowana ale zmiany wykonane w trakcie zapytania zostaną pominięte.

Trwała sesja

Ciasteczka:

- ciasteczka sesyjne - trwają dopóki nie wyłączy się przeglądarki,
- ciasteczka trwałe - zostają po zamknięciu przeglądarki, mają określony czas ważności, można je usunąć ręcznie,
- oprócz informacji o zalogowaniu użytkownika przechowują ustawienia, np. jezyk, styl strony, ...

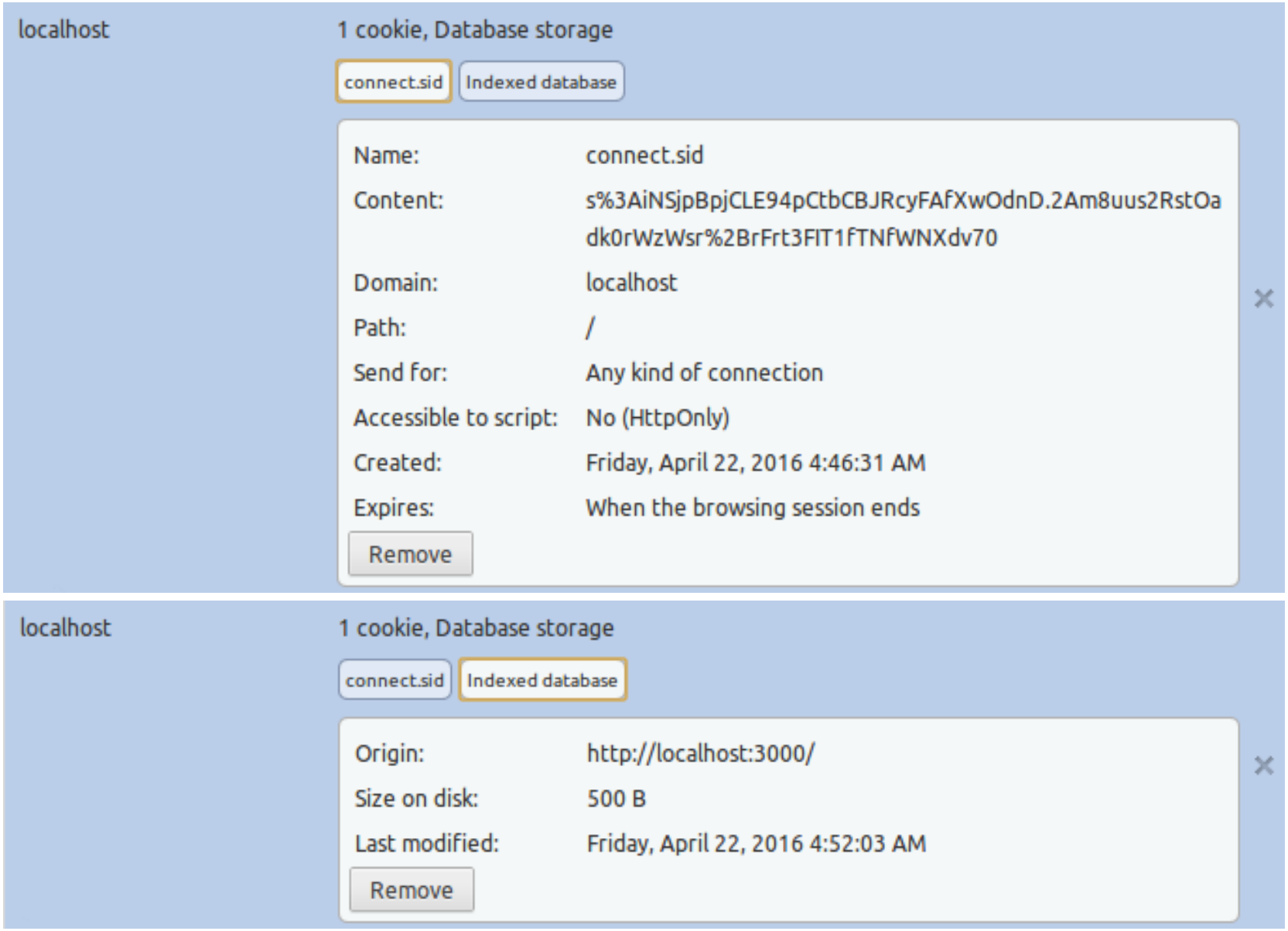
Trwała sesja (ang. *session presistence*):

- nie kończy się po zamknięciu przeglądarki (a to jest domyślne zachowanie).
- towrzy się przez ustawienie `cookie.maxAge`, np.:

```
app.use(session({ secret: 'w4d93k', cookie: { maxAge: 24*60*60*1000 }}));
```

- zapamiętanie stanu pracy użytkownika z jakiejś aplikacji w ciasteczkach albo w bazie danych,
- przy ponownym logowaniu stan pracy jest przywracany.

Ciasteczko sesyjne



Używanie express-session

Instalacja: `npm install express-session`

Odwiedziny: 7

Utrata ważności po: 59.028 s

```
var session = require('express-session');
app.use(session({
  secret: 'tajny kod', cookie: { maxAge: 60000 },
  resave: false, saveUninitialized: true
}))

app.get('/licznik', function(req, res, next) {
  var sesja = req.session;     // Dostęp do sesji po req.session
  console.log(req.session.id);
  if (sesja.odwiedziny) {
    sesja.odwiedziny++
    res.setHeader('Content-Type', 'text/html')
    res.write('<p>Odwiedziny: ' + sesja.odwiedziny + '</p>')
    res.write('<p>Utrata ważności po: '
              + (sesja.cookie.maxAge / 1000) + ' s</p>')
    res.end()
  } else {
    sesja.odwiedziny = 1
    res.end('Przykład wykorzystania sesji. Odśwież stronę!')
  }
})
```

Dane sesji

Pole **req.session**:

- umożliwia dostęp do danych sesji,
- przy składowaniu jest serializowane do JSONa - można składować zagnieżdżone obiekty.

req.session.id - unikalne ID związane z sesją - nie można go modyfikować.

// **req.sessionID** - dostęp do ID sesji, pole tylko do odczytu.

req.session.cookie - unikalne ciasteczko związane z sesją. Można modyfikując je dostosować sesję do różnych użytkowników, np. zmienić `req.session.cookie.expires` albo `req.session.cookie.maxAge`:

```
var hour = 3600000
req.session.cookie.expires = new Date(Date.now() + hour)
req.session.cookie.maxAge = hour
```

Metody związane z sesją

Metody modyfikujące bieżącą sesję:

- Session.regenerate() - tworzy ponownie sesję dla danego żądania,
- Session.destroy() - usuwa bieżącą sesję,
- Session.reload() - ponownie wczytuje dane sesji,
- Session.save() - zapisuje dane sesji,
- Session.touch() - aktualizuje pole `maxAge`.

Metody do przechowywania sesji:

- store.all(callback) - pobiera wszystkie zapisane sesje jako tablicę,
- store.destroy(sid, callback) - usuwa wybraną sesję,
- store.clear(callback) - usuwa wszystkie zachowane sesje,
- store.length(callback) - zwraca liczbę wszystkich zachowanych sesji,
- store.get(sid, callback) - pobiera wybraną sesję,
- store.set(sid, session, callback) - nadpisuje wybraną sesję.

Uwierzytelnienie i autoryzacja

Uwierzytelnienie (ang. *authentication*) - proces polegający na potwierdzeniu zadeklarowanej tożsamości podmiotu biorącego udział w procesie komunikacji. Celem uwierzytelniania jest uzyskanie określonego poziomu pewności, że dany podmiot jest w rzeczywistości tym, za który się podaje.

Autoryzacja (ang. *authorization*) – proces nadawania podmiotowi uprawnień do danych.

Uzyskiwanie dostępu do zasobów

Uzyskiwanie dostępu do chronionych zasobów:

- **Identyfikacja** - podmiot deklaruje swoją tożsamość, np. w procesie logowania do serwera użytkownik podaje nazwę (login), serwer jest stroną ufającą.

- **Uwierzytelnienie** - strona ufająca stosuje odpowiednią technikę uwierzytelnienia w celu weryfikacji zadeklarowanej wcześniej torzsamośći, np. serwer prosi o wpisanie hasła (lub wskazanie pliku klucza) i weryfikuje zgodność z wpisaną wcześniej nazwą.

- **Autoryzacja** - potwierdzenie czy podmiot jest uprawniony do uzyskania dostępu do żądanego zasobu. Na tym etapie podmiot jest już uwierzytelniony, sprawdzamy jednak czy ma prawo dostępu do konkretnego zasobu, np. sewer weryfikuje uprawnienia zalogowanego użytkownika do konkretnego pliku czy podstrony internetowej.

Uwierzytelnienie w express - passport

`http://passportjs.org/`

Cechy uwierzytelnienia z passport:

- proste i dyskretne (nie rzucające się w oczy),
- działa na bazie wartsty pośredniej w Node.js,
- bardzo elastyczne, zawiera budowę modułową,
- można je łatwo zastosować w każdej aplikacji używającej express'a,
- zawiera kompletny zbiór strategi pozwalający na autoryzację:
  - przy użyciu loginu i hasła
  - przy użyciu autoryzacji ze znanych portali: Facebook, Twitter, google, github, linkedin, ... (ponad 500 różnych strategii uwierzytelnienia do wykorzystania)

Uwierzytelnienie w express - passport

Passport:

- nie wymaga podłączania z góry określonych tras (routes)
- nie wymaga jakiegoś konkretnego schematu bazy danych
- jest bardzo elastyczny, kod jest czysty i zarządzalny, co pozwala na umieszczenie go w aplikacji zgodnie z wolą programisty.

API w passport jest proste:

- programista dostarcza żądanie, które należy uwierzytelnić
- passport dostarcza wstawki programowe (ang. *hooks*) kontrolujące czy uwierzytelnienie się powiodło.

```
$ npm install passport
```

Cechy passport

Cechy passport:

- ma służyć tylko autoryzacji, nie zajmuje się niczym więcej,
- obsługa pojedyńczej autoryzacji z OpenID czy OAuth,
- łatwa obsługa udanego i nieudanego logowania,
- wsparcie dla trwałych sesji,
- dynamiczny zasięg i uprawnienia,
- możliwość implementacji własnej strategii,
- lekki (mało kodu bazowego), passport wszystkie inne funkcjonalności przekazuje aplikacji ...

Uwierzytelnienie w passport

Różne współczesne metody uwierzytelnienia:

- tradycyjna: podajemy nazwę użytkownika i hasło
- związana z portalami społecznościowymi: pojedyncze logowanie dzięki OAuth; serwisy jak Facebook czy Twitter udostępniaja takie uwierzytelnienie,
- Serwisy udostępniające API często wymagają poświadczenia dostępu odpowiednim tokenem (żeton, dowód, znak).

Elastyczność

- Każda aplikacja ma inny system uwierzytelnienia.
- Mechanizmy uwierzytelnienia, znane jako strategie, są udostępniane jako osobne moduły.
- Aplikacja wybiera, które strategie autoryzacji chce udostępniać.

Uwierzytelnienie wbudowane w passport

- Złożony proces uwierzytelnienia ale prosty kod.
- Uwierzytelnienie z przekierowaniem:

```
app.post('/login',
        passport.authenticate('local', { successRedirect: '/',
                                         failureRedirect: '/login' })
);
```

- Uwierzytelnienie z funkcją:

```
app.post('/login',
  passport.authenticate('local'),
  function(req, res) {
    // Funkcja się wywoła jeśli użytkownik się uwierzytelnił
    // ‘req.user‘ zawiera dane uwierzytelnionego użytkownika
    res.send('### Użytkownik uwierzytelniony ###');
  });
```

Własna funkcja do uwierzytelnienia w passport

Jeśli wbudowane funkcje są niewystarczające, można dostarczyć własną funkcję obsługującą poprawne i błędne uwierzytelnienie:

```
app.post('/login', function(req, res, next) {
  passport.authenticate('local', function(err, user, info) {
    if (err) { return next(err); }
    if (!user) {
      return res.redirect('/login');
    }
    req.logIn(user, function(err) {
      if (err) { return next(err); }
      return res.send('Użytkownik uwierzytelniony - logowanie OK');
    });
  })(req, res, next);
});
```

- Mamy dostęp do żądania i odpowiedzi (`req` i `res`).
- Brak uwierzytelnienia - `user` jest ustawiony na `false`.
- Jest uwierzytelnienie - ręcznie musimy się zalogować `req.logIn()` i wysłać odpowiedź czy zrobić przekierowanie.

Wykorzystanie passport

Wykorzystanie passport krok po kroku:

1. Instalacja potrzebnych pakietów.
2. Przygotowanie bazy danych.
3. Wybór strategii, np. passport-local.
4. Ustawienie warstwy pośredniej.
5. Obsługa sesji (**opcjonalnie**).
6. Uwierzytelnienie + logowanie (z obsługą sesji lub bez).
7. Weryfikacja uwierzytelnienia użytkownika (**opcjonalnie**):
    1. pole isAuthenticated,
    2. pakiet connect-ensure-login.
8. Wylogowanie się (**opcjonalnie**).

1. Instalacja pakietów

Instalacja pakietów:

```
npm install express-session --save
npm install passport --save
npm install passport-local --save

// również mongoose, body-parser i inne jeśli potrzebne
```

i ich wykorzystanie:

```
...
var session = require('express-session');
var passport = require('passport');
LocalStrategy = require('passport-local').Strategy;
...
```

2. Przygotowanie bazy danych

- Schemat bazy danych **musi** zawierać pola: `username` oraz `password`.
- Schemat **musi** dostarczać metodę do sprawdzenia poprawności hasła, np. `validPassword()`.

```
mongoose.connect('mongodb://localhost/bazalog', function(err) {
    if(err) { console.log('błąd połączenia', err);
    } else { console.log('połączenie udane');
    }
});

var userSchema = new mongoose.Schema({
    username: String,
    password: { type: String, default: "1234" },
    first: String,
    last: String
});

userSchema.methods.validPassword = function (pass, callback) {
    return sha1(pass)==this.password;
}

var User = mongoose.model('User', userSchema);
```

3. Wybór strategii

- Strategie wymagają **weryfikującego wywołania zwrotnego** (ang. *verify callback*).
- Funkcja wywołania zwrotnego ma znaleźć użytkownika z danymi podanymi do uwierzytelnienia.
- `done()` zwraca dane użytkownika do passport.

```
passport.use(new LocalStrategy(
  function(username, password, done) {
    User.findOne({ username: username }, function (err, user) {
      if (err) { return done(err); }
      if (!user) {
        return done(null, false, { message: 'Incorrect username.' });
      }
      if (!user.validPassword(password)) {
        return done(null, false, { message: 'Incorrect password.' });
      }
      return done(null, user);
    });
  }
));
```

4. Ustawienie warstwy pośredniej

- Ustawienie sesji express musi być przed sesją passport.
- Inicjalizacja passport musi być po przygotowaniu konkretnej strategii.

```
app.use(session({                  // sesja dla express'a, potrzebna
  resave: false,                   // jeśli wykorzystujemy sesje
  saveUninitialized: false,
  secret: 'tajny kod',
  cookie: { maxAge: null }
}));

app.use(passport.initialize());
app.use(passport.session());       // sesja dla passport, dodajemy TYLKO,
                                   // jeśli zdefiniowaliśmy serializację,
                                   // konieczna do trwałego zalogowania
```

5. Obsługa sesji (opcjonalnia ale zalecana)

- W typowej aplikacji dane uwierzytelniające są przesyłane tylko przy logowaniu.
- Po poprawnej autoryzacji utworzy się sesja z ID w ciasteczku przeglądarki.
- Każde kolejne żądanie nie będzie potrzebowało danych uwierzytelniających ale ciasteczka, które ma identyfikator sesji.
- passport dokona serializacji i deserializacji obiektu `user` (obiekt z mongoDB) do i z sesji.

```
passport.serializeUser(function(user, done) {
  // console.log(JSON.stringify(user));
  // console.log(user.id);
  done(null, user.id);
});

passport.deserializeUser(function(id, done) {
  User.findById(id, function(err, user) {
    done(err, user);
  });
});
```

5. Obsługa sesji (opcjonalnia ale zalecana)

- Tylko ID użytkownika jest użyte do serializacji do zapisu w sesji.
- Przy odbiorze żądań ID jest użyte do znalezienia użytkownika w bazie i odtworzeniu `req.user`.
- Serializację i deserializację dostarcza aplikacja, programista - duża elastyczność.

```
passport.serializeUser(function(user, done) {
  done(null, user.id);
});

passport.deserializeUser(function(id, done) {
  User.findById(id, function(err, user) {
    done(err, user);
  });
});

// {"username":"jb","_id":"571d56cb3bd41b73249ff3a0", "first": ...  }
// req.session:
// passport: { user: '5717ea4e909d743b23e58d2b' }
```

6. Uwierzytelnienie + login

- Bez wykorzystania sesji - (`session: false`)

```
app.post('/logowanie',
    passport.authenticate('local',
        { session: false,
          successRedirect: '/zalogowany',
          failureRedirect: '/logowanie.html' })
);
```

- Z wykorzystaniem sesji - (`session: true`), domyślnie
- Koniecznie trzeba zdefiniować serializację

```
app.post('/logowanie',
    passport.authenticate('local',
        { successRedirect: '/zalogowany',
          failureRedirect: '/logowanie.html' })
);
```

6. Formularz logowania

Formularz logowania:

- zawiera pola `username` oraz `password`
- adres `/logowanie`, metoda POST

```
<form action="/logowanie" method="post">
    <div>
        <label>Login:</label>
        <input type="text" name="username"/>
    </div>
    <div>
        <label>Hasło:</label>
        <input type="password" name="password"/>
    </div>
    <div>
        <input type="submit" value="Zaloguj"/>
    </div>
</form>
```

6. Ręczne logowanie, metoda login()

Logowanie się w passport:

- passport udostępnia metodę `login()` działającą na obiekcie zapytania (`req`),
- metoda ma również alias: `logIn()`,
- metodę można użyć do ustanowienia sesji użytkownika.

```
req.login(user, function(err) {
  if (err) { return next(err); }
  return res.redirect('/users/' + req.user.username);
});
```

- Po zakończeniu logowania `user` będzie przypisany do `req.user`.
- Warstwa pośrednia `passport.authenticate()` automatycznie wywołuje `req.login()`.
- Funkcja `login()` jest używana głównie w przypadku rejestracji użytkowników, których chcemy od razu zalogować.

7.1. Pole isAuthenticated pakietu passport

- Korzystamy **z nieudokumentowanego** atrybutu, funkcji pakietu passport - `isAuthenticated`.
- Brak uwierzytelnienia - przeniesienie na stronę `/login`.
- Użytkownik zalogowany, uwierzytelnienie poprawne - wyświetlenie danych sesji i informacji użytkownikowi.

```
app.get('/authtest', function(req, res, next) {
  if (!req.isAuthenticated || !req.isAuthenticated()) {
    return res.redirect('/login');
  } else {
    console.log(req.session);
    res.send('### Użytkownik uwierzytelniony ###');
  }
});
```

7.2. Pakiet connect-ensure-login

- Korzystamy z metody ensureLoggedIn() pakietu `connect-ensure-login`.
- Brak uwierzytelnienia - przeniesienie na stronę `/login`.
- Użytkownik zalogowany, uwierzytelnienie poprawne - wyświetlenie danych sesji i informacji użytkownikowi.

```
app.get('/authtest',
  require('connect-ensure-login').ensureLoggedIn('/login'),
  function(req, res){
    console.log(req.session);
    res.send('### Użytkownik uwierzytelniony ###');
  });
```

7.2. Pakiet connect-ensure-login

Pakiet **connect-ensure-login**:

- wartswa pośrednia pozwalająca się upewnić, że użytkownik jest zalogowany,
- w przypadku żądania do zasobów bez uwierzytelnienia nastąpi przekierowanie na stronę logowania,
- adres URL żądania będzie zapisany w sesji; użytkownik będzie mógł być wygodnie przekierowany do strony, którą chciał otworzyć.

```
app.get('/settings',
  ensureLoggedIn('/login'),
  function(req, res) {
    res.render('settings', { user: req.user });
  });
```

- Aplikacja ma stronę ustawień (`settings`) dostępną tylko dla zalogowanych.
- Niezalogowany użytkownik zostanie przekierowany na stronę `/login`.
- Adres orginalnego żądania (`/settings`) zostanie zapisany w `req.session.returnTo`.

7.2. Pakiet connect-ensure-login

Pakiet **connect-ensure-login** krok po kroku:

```
app.get('/settings',
  ensureLoggedIn('/login'),
  function(req, res) {
    res.render('settings', { user: req.user });
  });
```

1. Żądanie adresu `/settings`
   - Warstwa pośrednia ustawia `session.returnTo` na `/settings`
   - Warstwa pośrednia przekierowuje do `/login`
2. Przeglądarka użytkownika przechodzi do GET `/login`
   - Aplikacja renderuje formularz logowania.
3. Użytkownik wpisuje dane i wysyła POST `/login`
   - Aplikacja weryfikuje dane logowania.
   - Passport odczytuje `session.returnTo` i przekierowuje na `/settings`.
4. Przeglądarka użytkownika przekierowuje na GET `/settings`
   - Po uwierzytelnieniu strona `/settings` wyświetla się w przeglądarce.

8.1. Wylogowanie się - metoda logout()

Wylogowywanie się w passport:

- passport udostępnia metodę `logout()` działającą na obiekcie zapytania (`req`),
- metoda ma również alias: `logOut()`,
- metodę można wywołać na dowolnej funkcji związanej z obsługą żądań, w której jest konieczność zakończenia sesji,
- wywołanie `logout()` usunie pole `req.user` i wyczyści sesję logowania (jeśli była).

```
app.get('/logout', function(req, res){
  req.logout();
  res.redirect('/');
});
```

8.2. Pakiet express-passport-logout

Pakiet **express-passport-logout**:

- warstwa pośrednia express, służy do wylogowania użytkownika z passport i przekierowania go do określonej lokalizacji
- wykonuje:
  - `req.logout()`
  - delete req.session
  - przekierowuje na `req.query.returnTo` jeśli adres istnieje, jeśli nie to zwraca `res.send('bye')`

```
module.exports = function() {
  return function (req, res) {
    var returnTo = req.query.returnTo;

    req.logout();
    delete req.session;
    return redir(res, returnTo);

    function redir(res, returnTo) {
      if (returnTo) {
        return res.redirect(returnTo);
      } else {
        return res.send('bye');
      }
    }
  };
};
```

8.2. Pakiet express-passport-logout

Użycie **express-passport-logout**:

```
var logout = require('express-passport-logout');

app.get('/logout', logout());
```

Tutoriale

- **Dokumentacja passport** - łatwa nie jest ale da się zrozumieć
- **passport tutorial dla Express v4** - jest podział na moduły z pakietem `passport-local-mongoose`
- **passport-local-express4** - działający przykład na github
- **The Ins and Outs of Token Based Authentication** - styczeń 2015.
- **JSON Web Tokens** - są też odniesienia do passport, kwiecień 2015.
- **Restful API user authentication** - tutorial z Web Tokens, listopad 2015.

Źródła

- https://pl.wikipedia.org/wiki/Uwierzytelnianie
- https://pl.wikipedia.org/wiki/Autoryzacja_(informatyka)
- https://pl.wikipedia.org/wiki/Sesja_(informatyka)
- http://passportjs.org/
- https://www.npmjs.com/package/passport-local
- https://devdactic.com/restful-api-user-authentication-1/
- https://scotch.io/bar-talk/the-ins-and-outs-of-token-based-authentication