# Uwierzytelnianie i autoryzacja

Bezpieczne systemy internetowe

---

1. Różnica między uwierzytelnianiem a autoryzacją jest oczywista, jeżeli posługujemy się definicjami i prostymi przykładami. Niestety w przypadku systemów webowych różne standardy, mechanizmy i modele, zarówno uwierzytelniania jak i autoryzacji, przenikają się ze sobą na tyle mocno, że często jeden termin można interpretować na wiele różnych sposobów. Utrudnia to wybór odpowiednich mechanizmów i wymusza dobre ich zrozumienie (zazwyczaj nie da rady stosować ich "intuicyjnie"). Przykładowo **JWT**, czyli format uwierzytelnionego tokenu **JSON**[1], może być stosowany zarówno do autoryzacji (np. jako zamiennik natywnego tokenu schematu **OAuth2**[2]) jak i składowa modelu uwierzytelniania (istnieje wiele odmiennych modeli uwierzytelniania wykorzystujących **JWT**, które autorzy w skrócie nazywają po prostu … **JWT**). Nierzadko zdarza się, że w celu osiągnięcia pewnych określonych efektów łączy się kilka schematów, np **OAuth2** i jeden z modeli **JWT**[3]
2. W zależności od od tego, czy serwer, wobec którego się uwierzytelniamy, przechowuje informacje o udanym uwierzytelnieniu, rozróżniamy metody stanowe i bezstanowe[4]. Metody stanowe są zazwyczaj bardziej użyteczne w przypadku uwierzytelniania na stronie, metody bezstanowe (czyli najczęściej klucze API i tokeny sprowadzone do funkcji autoryzującej) znajdują zastosowanie w przypadku zabezpieczania punktów końcowych serwisów[5].
3. Z każdą metodą uwierzytelniania i autoryzacji związane są konkretne zagrożenia. Decydując się na wybraną metodę powinniśmy zapoznać się z jej ograniczeniami (np. w przypadku do uwierzytelniania tokenów **JWT**[6]). Dobrym źródłem informacji o zaleceniach dotyczących poprawnej implementacji wybranych metod oraz ich ograniczeń są dokumenty **OWASP** (przykładowo dotyczącego poprawnego zarządzania hasłami[7]). Absolutną podstawą jest zestawianie połączeń jedynie po protokole HTTPS oraz, w przypadku uwierzytelniania stanowego, wykorzystującego sesje, poprawne zarządzanie plikami ciasteczek z uwzględnieniem ich podatności na ataki **XSS** i **CSRF**[8].
4. W ćwiczeniu zabezpieczymy jedynie serwis REST, wykorzystując dwa rodzaje uwierzytelniania bezstanowego, **Basic Authentication** w formie podstawowej (bez używania funkcji skrótu) oraz tokeny **JWT** (JSON Web Token, jedna z metod w klasie **Bearer**). Obydwu metod będziemy używali niezależnie (zabezpieczymy różne punkty końcowe API udostępnianego przez serwer REST).
5. Po stronie serwera REST do obsługi uwierzytelniania użyjemy frameworku **Passport**. Pozwala on na używanie różnych metod uwierzytelniania (nazywanych **strategiami**). Na potrzeby naszego zadania instalujemy pakiety **passport** (npm install passport --save), **passport-http** (npm install passport-http --save) oraz **passport-jwt** (npm install passport-jwt --save).
6. Po stronie serwera HTTPS instalujemy pakiet **jsonwebtoken** (npm install jsonwebtoken --save), który pozwala na generowanie i manipulowanie tokenami **JWT**.
7. W serwerze REST tworzymy tablicę z kilkoma użytkownikami (obiekt użytkownika ma zawierać unikalny identyfikator, nazwę i hasło). **UWAGA! W rzeczywistej aplikacji**

---

[1] https://jwt.io/
[2] https://tools.ietf.org/html/rfc7523
[3] https://capgemini.github.io/architecture/combining-oauth-and-jwt-to-gain-performance-improvements/
[4] https://blog.imaginea.com/stateless-authentication-using-jwt-2/
[5] https://zapier.com/engineering/apikey-oauth-jwt/
[6] https://www.sjoerdlangkemper.nl/2016/09/28/attacking-jwt-authentication/
[7] https://www.owasp.org/index.php/Password_Storage_Cheat_Sheet
[8] https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies

**absolutnie nie wolno przechowywać haseł w formie jawnej!** Używamy do tego np. **bcrypt** lub posługujemy się mechanizmami udostępnionymi przez bazy danych.

8. Piszemy własną funkcję obsługującą **BasicStrategy**, czyli Basic HTTP Authentication[9]. Do funkcji przekazywana jest nazwa użytkownika i hasło zdekodowane z żądania przesłanego do serwera oraz funkcja w łańcuchu middlware, która ma zostać wywołana jako następna po funkcji uwierzytelniającej:

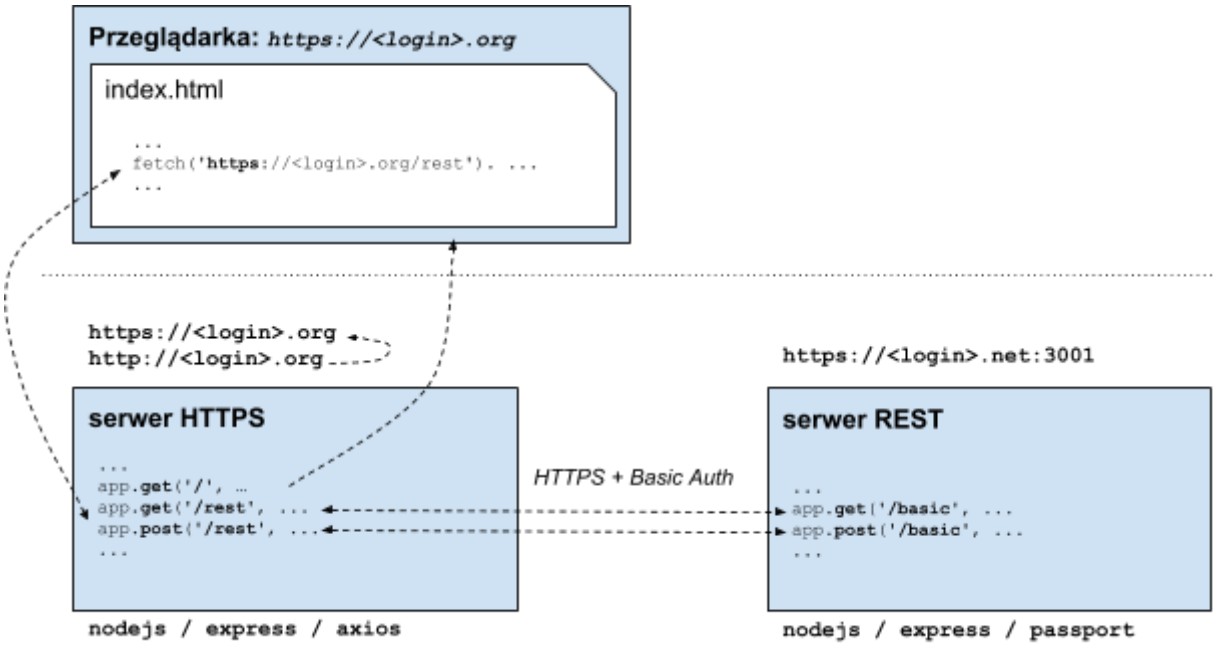
```
...
passport.use(new BasicStrategy(
   (username, password, next) => {

      ...

   }
));
...
```

W funkcji sprawdzamy, czy istnieje podany użytkownik i czy podano poprawne hasło. Najczęściej w tym miejscu pojawia się odwołanie do bazy danych z użytkownikami, my posłużymy się jednak wcześniej przygotowaną tablicą użytkowników. Następnie kopiujemy istniejący kod obsługi metod GET i POST (z istniejącego punktu końcowego **/**) i tworzymy dla tych metod nowy punkt końcowy **/basic**. W obydwu funkcjach dodajemy ochronę za pomocą uwierzytelniania 'basic', np, dla GET:

```
...
app.get('/basic', passport.authenticate('basic', {session: false}), ...
...
```

Po co w parametrach uwierzytelniania ustawiliśmy session na wartość false? Sprawdzić komunikację z zabezpieczonymi punktami końcowymi za pomocą programu **Insomnia**.

9. W serwerze HTTPS modyfikujemy funkcje obsługi żądań GET i POST tak, aby odwoływały się do zabezpieczonych punktów końcowych serwera REST (**https:<login>.net:3001/basic**). W wywołaniu klienta **axios** musimy dodać opcję **auth**, zawierająca nazwę i hasło użytkownika, które zostaną przekazane podczas uwierzytelniania [10]. W jaki sposób przekazywane są dane użytkownika przy basic auth? (użyć **Wireshark**). Czy metodę tę można stosować w protokole HTTP?




*Rys.1 Schemat komunikacji pomiędzy przeglądarką a serwerami (wykonanie ćwiczenia do kroku 9).*

[9] http://www.passportjs.org/docs/basic-digest/
[10] https://github.com/axios/axios

10. Uwierzytelnianie tokenem **JWT**, które zostanie zastosowane w dalszej części zadania, ma charakter jedynie poglądowy. Nie należy stosować architektury, w której token jest generowany przy każdym żądaniu (mija się to z podstawowymi założeniami leżącymi u podstaw koncepcji tokenów). Również miejsce generowania tokena w przykładzie jest co najmniej dyskusyjne. W normalnych warunkach token powinien zostać wygenerowany przez serwer uwierzytelniający (lub przez serwer usługowy, z którym mamy się zamiar komunikować) i podpisany symetrycznie (HMAC) bądź asymetrycznie (RSA, krzywe eliptyczne). Serwer powinien wygenerować token po wcześniejszym uwierzytelnieniu użytkownika **inną metodą!** Token może być wtedy używany podczas komunikacji klienta z serwerem usługowym, uwierzytelniając każde żądanie (serwer usługowy weryfikuje każdorazowo poprawność tokenu, wykorzystując do tego albo sekret symetryczny albo odpowiedni klucz publiczny, oraz zawartość tokenu - np. zawarty w nim identyfikator użytkownika)[11]. W naszym przykładzie zarówno serwer HTTPS (czyli klient) jak i serwer REST (czyli serwer usługowy) posłużą się tym samym sekretem (komunikacja będzie zabezpieczona domyślnym algorytmem HMAC).
11. W serwerze REST importujemy strategię **JWT** [12] [13]:

```
...
const passportJWT = require("passport-jwt");
const JwtStrategy = passportJWT.Strategy;
const ExtractJwt = passportJWT.ExtractJwt;
...
```

Następnie piszemy własną funkcję obsługującą **JwtStrategy**. W konstruktorze strategii podajemy m.in. sekret, który posłużył do symetrycznego podpisania tokenu (a teraz zostanie użyty do jego weryfikacji). Doomyślną metodą jest HMAC. Funkcja obsługi strategii otrzymuje jako argumenty zdekodowany i zweryfikowany ładunek tokenu oraz kolejną funkcję w łańcuch wywołań middleware:

```
...
passport.use( new JwtStrategy(
   {
       jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
       secretOrKey: 'nieprawdopodobnySekret'
   },
   (jwtPayload, next) => {
       ...
   }
));
...
```

Decyzję o autoryzacji podejmujemy na podstawie ładunku tokenu - w naszym scenariuszu będzie to jedynie identyfikator użytkownika. Ufamy, że podczas tworzenia tokenu zostało przeprowadzone uwierzytelnianie użytkownika wobec serwera, który również posiada nasz sekret (czyli jeżeli ta strona zaufała użytkownikowi i wygenerowała mu token to my, po zweryfikowaniu podpisu na tokenie, co w naszej strategii wykonywane jest automatycznie, również mu ufamy). W funkcji wystarczy więc sprawdzić, czy dany identyfikator jest na liście użytkowników. Następnie kopiujemy istniejący kod obsługi metod GET i POST (z istniejącego punktu końcowego */*) i tworzymy dla tych metod nowy punkt końcowy **/jwt**. W obydwu funkcjach dodajemy ochronę za pomocą uwierzytelniania ‘basic’, np, dla GET:

```
...
```

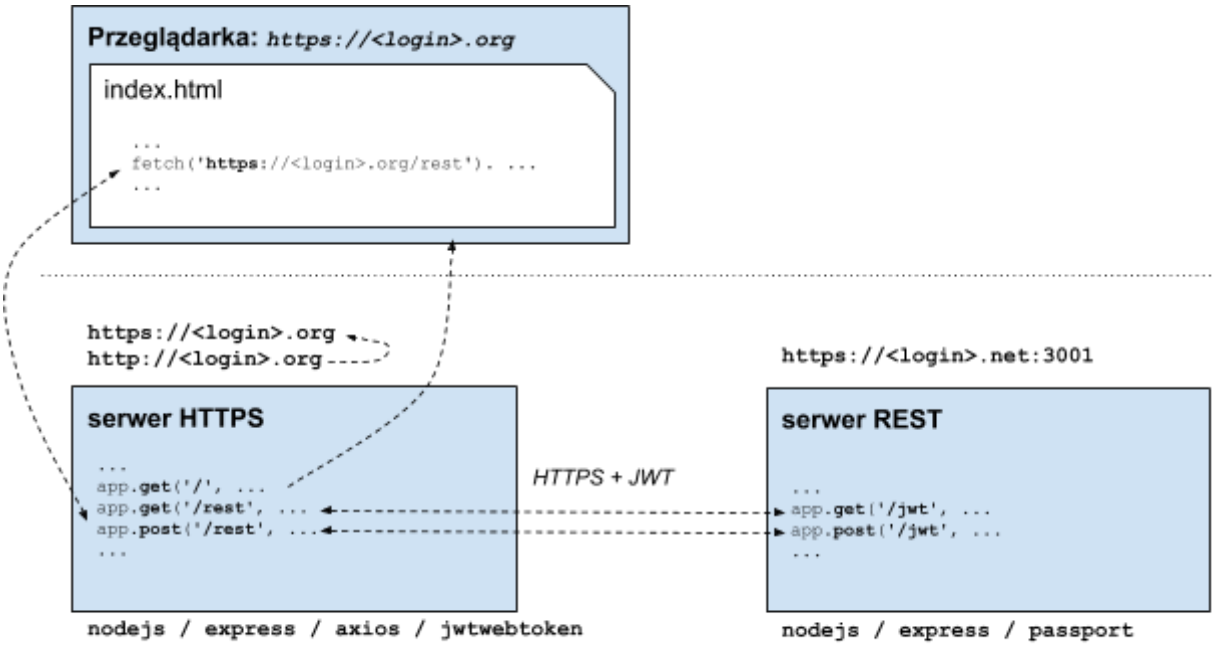
[11] https://medium.com/@siddharthac6/json-web-token-jwt-the-right-way-of-implementing-with-node-js-65b8915d550e
[12] https://medium.com/front-end-hacking/learn-using-jwt-with-passport-authentication-9761539c4314
[13] https://scotch.io/@devGson/api-authentication-with-json-web-tokensjwt-and-passport

```
        app.get('/basic', passport.authenticate('jwt', {session: false}), ...
        ...
```

12. W serwerze HTTPS modyfikujemy funkcje obsługi żądań GET i POST tak, aby odwoływały się do zabezpieczonych punktów końcowych serwera REST (**https:<login>.net:3001/jwt**). Importujemy pakiet **jsonwebtoken**[14], z którego wykorzystamy metodę **sign**. Za jej pomocą generujey token zawierający identyfikator wybranego użytkownika podpisany sekretem (sekretu używamy również w serwerze REST). W wywołaniu klienta **axios** każdorazowo generujemy nowy token (ustalając mu krótki, np. 2 minutowy okres ważności) i umieszczamy go w opcjach jako nagłówek **Authentication**. Token wyświetlamy również w konsoli **nodejs** aby po skopiowaniu można było go użyć w **Insomnia**. Wywołanie może przyjąć np. następującą postać:

```
...
const token = jwt.sign({id: 2}, 'nieprawdopodobnySekret', {expiresIn: 120});
console.log(token);
axios.get('https://wmackow.net:3001/jwt', {
        httpsAgent,
        headers: {
                'Authorization': `Bearer ${token}`
        }
})
...
```

Komunikację testujemy zarówno za pomocą strony udostępnianej przez serwer HTTPS jak i odwołując się bezpośrednio do punktów końcowych **/jwt** serwera REST za pomocą aplikacji **Insomnia**.




*Rys.2 Schemat komunikacji pomiędzy przeglądarką a serwerami (wykonanie ćwiczenia do kroku 12).*

[14] https://github.com/auth0/node-jsonwebtoken