Wybrane elementy praktyki projektowania oprogramowania
Wykład 09/15
node.js: Express (2)

Wiktor Zychla 2023/2024

1 Spis treści

2 Domyślne middleware do obsługi nieobsłużonych ścieżek oraz zagrożenie Cross-Site Scripting

„Domyślne” middleware, dodane jako ostatnie, będzie obsługiwać wszystkie nieobsłużone do tej pory ścieżki. Można użyć tej techniki do przechwycenia żądań do nieobsługiwanych ścieżek:

```
// ... wcześniej inne mapowania ścieżek

app.use((req,res,next) => {
    res.render('404.ejs', { url : req.url });
});

http.createServer(app).listen(3000);
```

```
<!-- 404.ejs -->
<html>

<body>
    Strona <%= url %> nie została znaleziona.
</body>

</html>
```

Przy okazji przyjrzyjmy się temu że wartość **req.url** jest w obiekcie żądania zakodowana w standardzie Percent-encoding (URL Encoding), co ma chronić aplikację przed prostym atakiem Script injection – a konkretnie jego wersją nazwaną Cross-site scripting.

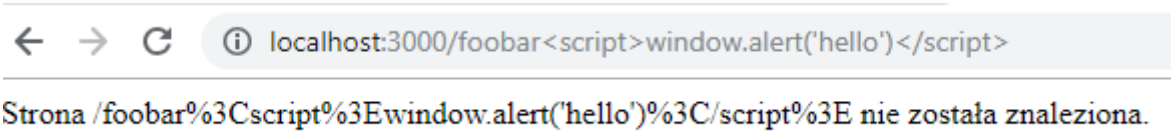
W tym ataku, atakujący preparuje dane, które przesyła na serwer (na przykład przez pasek adresowy albo przez jakieś pole tekstowe). Dane są spreparowane tak, żeby zawierać poprawny Javascript, który wykona przeglądarka.

Atakowany odwiedza stronę, na której zostanie wyrenderowany taki spreparowany Javascript. Skrypt wykrada jakieś wrażliwe dane ze strony i wysyła je na serwer atakującego.

W tym prostym przykładzie wyżej, strony która w treści strony zwraca wprost fragment ścieżki adresu: możliwy atak polegałby na przesłaniu przez atakującego spreparowanego odnośnika zawierającego fragment skryptu do wykonania:

http://localhost:3000/foobar<script>window.alert(‘hello’)</script>

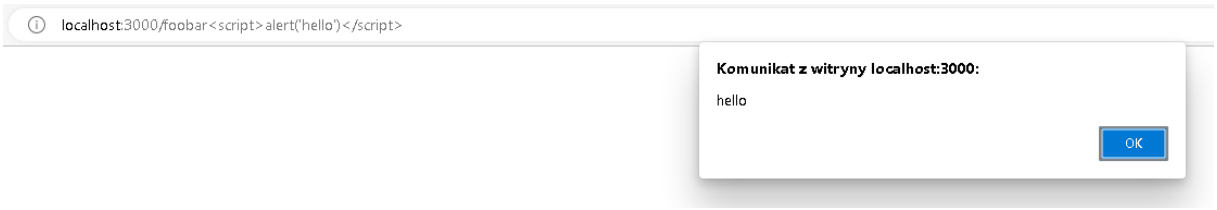
W zaprezentowanej powyżej wersji taki odnośnik spowoduje wyrenderowanie nieczytelnego



Widać, że użycie znacznika **<%= %>** chroni aplikację przed tego typu zagrożeniem – znacznik ten powoduje bowiem właśnie wzmiankowane wyżej zakodowanie zawartości, w szczególności fragmenty < > znaczników są zakodowane jako %3C i %3E.

Wystarczyłoby jednak omyłkowo (ale nieświadomie) w kodzie po stronie serwera użyć funkcji **decodeURIComponent** i równocześnie w widoku użyć <%- %> zamiast **<%= %>** (tag **<%- %>** służy do wypisania zawartości **bez** dodatkowego kodowania), aby otworzyć podatność. Jej źródłem jest bezrefleksyjne zwrócenie użytkownikowi zawartości, której część pochodzi od niego samego lub innego użytkownika.

```
<!-- 404.ejs, ups! -->
<html>

<body>
    Strona <%- decodeURIComponent(url) %> nie została znaleziona.
</body>

</html>
```

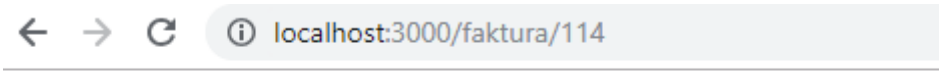


Ten przykład jest oczywiście sztuczny, osiągnięty efekt nie jest trwały, a zasięg ataku jest lokalny (można „zaatakować" samego siebie). Ale proszę sobie wyobrazić że złośliwa wartość (w żargonie mówimy ***payload***) jest utrwalona na serwerze (w bazie danych) i jest renderowana nie dla jednego ale dla wielu (wszystkich?) użytkowników.

3 Użycie parametrów w ścieżkach i zagrożenie Web Parameter Tampering

Możliwe jest dynamiczne parametryzowanie ścieżek

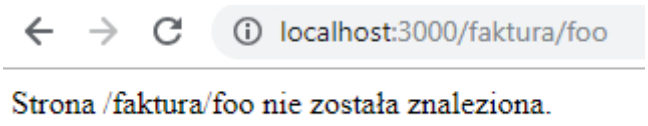
```
var http = require('http');
var express = require('express');

var app = express();

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

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

app.get("/",
    (req, res) => { res.end("default page")});

app.get("/faktura/:id",
    (req, res) => { res.end(`dynamicznie generowana faktura:
${req.params.id}`)});

// ... wcześniej inne mapowania ścieżek

app.use((req,res,next) => {
    res.render('404.ejs', { url : req.url });
});

http.createServer(app).listen(3000);
```




```
dynamicznie generowana faktura: 114
```

Taka ścieżka dopasowuje się do całej klasy ścieżek, a ograniczenie dopasowania polega na możliwości użycia wyrażenia regularnego, na przykład wymuszającego tylko liczby

```
app.get("/faktura/:id(\\d+)",
    (req, res) => { res.end(`dynamicznie generowana faktura:
${req.params.id}`)});
```



Więcej informacji o możliwościach routingu jest w dokumentacji.

Użycie parametrów w ścieżkach otwiera aplikację na kolejny typ podatności o którym warto wspomnieć, tzw. Web Parameter Tampering (znane też jako Query-String Tampering). Atak ten polega na tym że pasek adresowy jest pod kontrolą użytkownika przeglądarki, a programiście aplikacji webowej zdarza się o tym zapomnieć.

Typowa sytuacja w której dochodzi do podatności polega na tym, że aplikacja konkretnemu, zalogowanemu użytkownikowi generuje linki do jego zasobów (faktury, powiadomienia, itp.) i udostępnia takie linki użytkownikowi tak, żeby mógł z nich korzystać bez logowania się (np. wysyła przez e-mail czy sms).

Link zawiera wskazane rodzaju zasobu i jakiś dodatkowy parametr identyfikujący zasób (na przykład identyfikator z bazy danych).

Na przykład coś w stylu https://rejestr.faktur.pl/faktura/1448219

Użytkownik po otwarciu linka dostaje się do zasobu przewidzianego dla niego. Ciekawski użytkownik być może spróbuje jednak modyfikować pasek adresowy i zamieniać identyfikator na inny identyfikator – w nieodpowiednio zabezpieczonej aplikacji udaje się mu w ten sposób uzyskać dostęp do nieswoich danych.

Nie zawsze jest to problem, proszę pomyśleć o interfejsie np. porównywarki cen produktów lub jakimś innym systemie prezentującym katalog jakichś produktów. Do takich danych chce się żeby miał dostęp każdy niezalogowany użytkownik i nawet jeśli dochodzi do manipulacji zawartością paska adresowego, to nie ma w tym żadnego zagrożenia.

Jednym z typowych mechanizmów **ochrony** przez tym zagrożeniem jest użycie dodatkowego parametru weryfikującego poprawność odnośnika, wykorzystującego mechanizm HMAC (Hash Message Authentication Code). Na serwerze, parametr ścieżki adresowej (ten identyfikator zasobu, tu: 1148219) jest łączony z **kluczem tajnym** a otrzymana wartość jest użyta jako argument do jednokierunkowej funkcji skrótu (np. SHA2).

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

/* tajny klucz, znany tylko na serwerze */
var secret    = 'this is a secret';

/* parametr który zobaczy użytkownik */
var parameter = '1448219';

/* podpis który też zobaczy użytkownik */
var hmac =
```

```
    crypto
        .createHmac('sha256', secret)
        .update(parameter)
        .digest('hex');

console.log( hmac );

// d4d07c762f416897d9d4016f51b2ff3b634ec1020ce33c6d86fdd151f7a12e3e
```

Wynik jest dodawany do adresu jako jego „podpis”:

https://rejestr.faktur.pl/faktura/1448219?mac=d4d07c7....

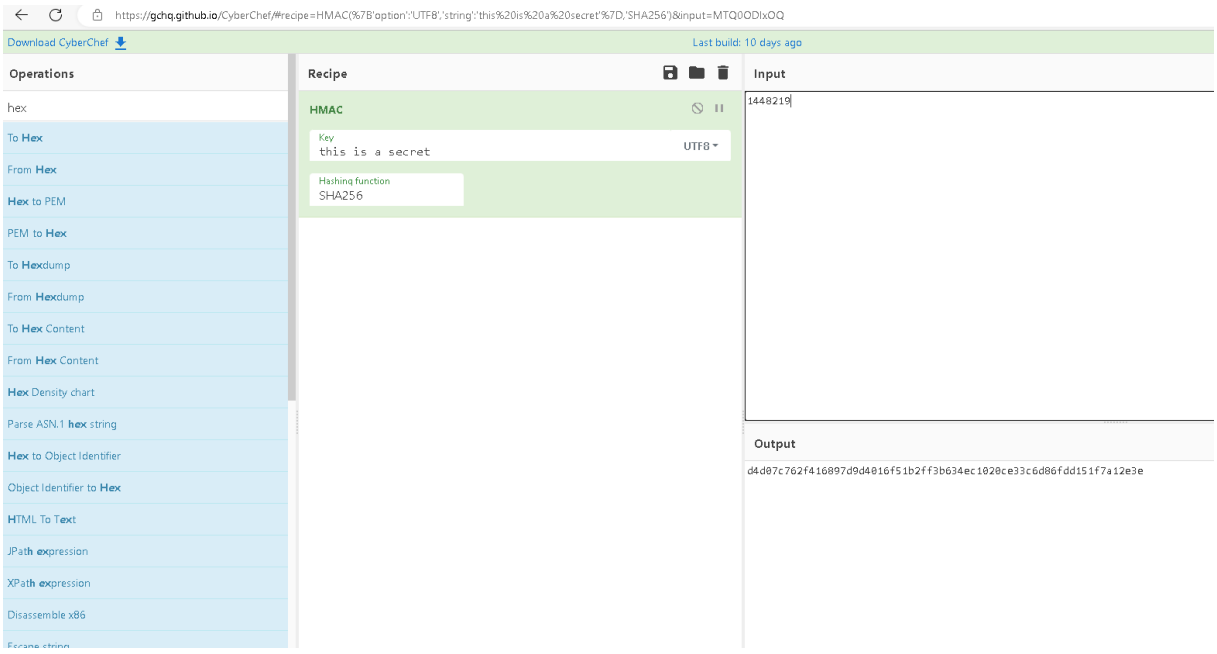
Kiedy takie żądanie przychodzi do serwera, w celu jego walidacji operacja wyliczenia „podpisu” jest powtarzana.

Następnie sprawdza się czy wyliczona wartość zgadza się z tą która przyszła w żądaniu. Niezgodność podpisu wyliczonego z oczekiwanym oznacza że użytkownik **zmodyfikował** wartość parametru. Taką sytuację można jawnie obsłużyć i zwrócić komunikat błędu.

Bezpieczeństwo tej metody jest zagwarantowane przez niemożność wyznaczenia argumentu dla znanej wartości funkcji skrótu. Atakujący żeby móc wyznaczać wartość podpisu dla dowolnej wartości parametru, musiałby znać wartość tajnego klucza. Proszę w szczególności zwrócić więc uwagę na to, gdyby „podpis” **pomijał tajny składnik**, czyli gdyby podpis był po prostu wartością funkcji skrótu dla zadanego parametru, atakujący sam z łatwością wyznaczałby poprawne wartości dla dowolnych parametrów (!).

Mechanizm HMAC jako gwarant autentyczności (prawdziwości) wiadomości jest prosty i bardzo skuteczny.

Przy okazji tego typu eksperymentów, warto zapoznać się z narzędziem CyberChef



Inny, prostszy, ale nie zawsze możliwy do zastosowania sposób ochrony przed tym zagrożeniem pojawia się w sytuacji gdy link dotyczy **zalogowanego użytkownika**, który na przykład – dostał listę danych z bazy danych, powiązanych jakoś ze swoimi danymi. Czyli, w uproszczeniu – jest zapytanie do bazy danych, które dla zalogowanego użytkownika zwraca listę dozwolonych identyfikatorów i ta lista dozwolonych identyfikatorów pojawia się jako część listy odnośników (na przykład rekordy faktur o identyfikatorach 2, 6, 19 dają odnośniki /faktura/2, faktura/6 i faktura/19).

W takiej sytuacji – jeżeli przychodzi żądanie z przeglądarki i zawiera **jakiś** identyfikator, to aplikacja sprawdza następujące warunki:

- Jeżeli użytkownik jest niezalogowany – zwracana jest informacja o błędzie
- Jeżeli użytkownik jest zalogowany – zapytanie wyciągające zbiór identyfikatorów jest **powtarzane** a następnie sprawdza się czy przychodząca wartość parametru znajduje się w zbiorze
    - Nie znajduje się – zwracana jest informacja o błędzie
    - Znajduje się – dopiero wtedy zwraca się dane

To **podwójne** sprawdzenie poprawności danych – zarówno w momencie renderowania linku jak i w momencie kiedy użytkownik klika w ten link skutecznie uniemożliwia nadużycia.

4 Obsługa ciastek

Do obsługi ciasteczek służy middleware **cookie-parser**. Wartość ciastka utworzona na serwerze jest dodawana do odpowiedzi w nagłówku **Set-cookie**, a następnie dołączana przez przeglądarkę w każdym żądaniu. Na serwerze można odczytać wartości przychodzących ciastek.

Uwaga! To przeglądarka dba o to żeby ciasteczka z witryny X nie były wysyłane do witryny Y. Odpowiada za to fragment specyfikacji, RFC6265.

Proszę zwrócić przy okazji uwagę na **app.disable('etag')** które wyłącza pewne mechanizmy cache – włączone cache bywa że utrudnia trochę śledzenie aplikacji za pomocą debuggera.

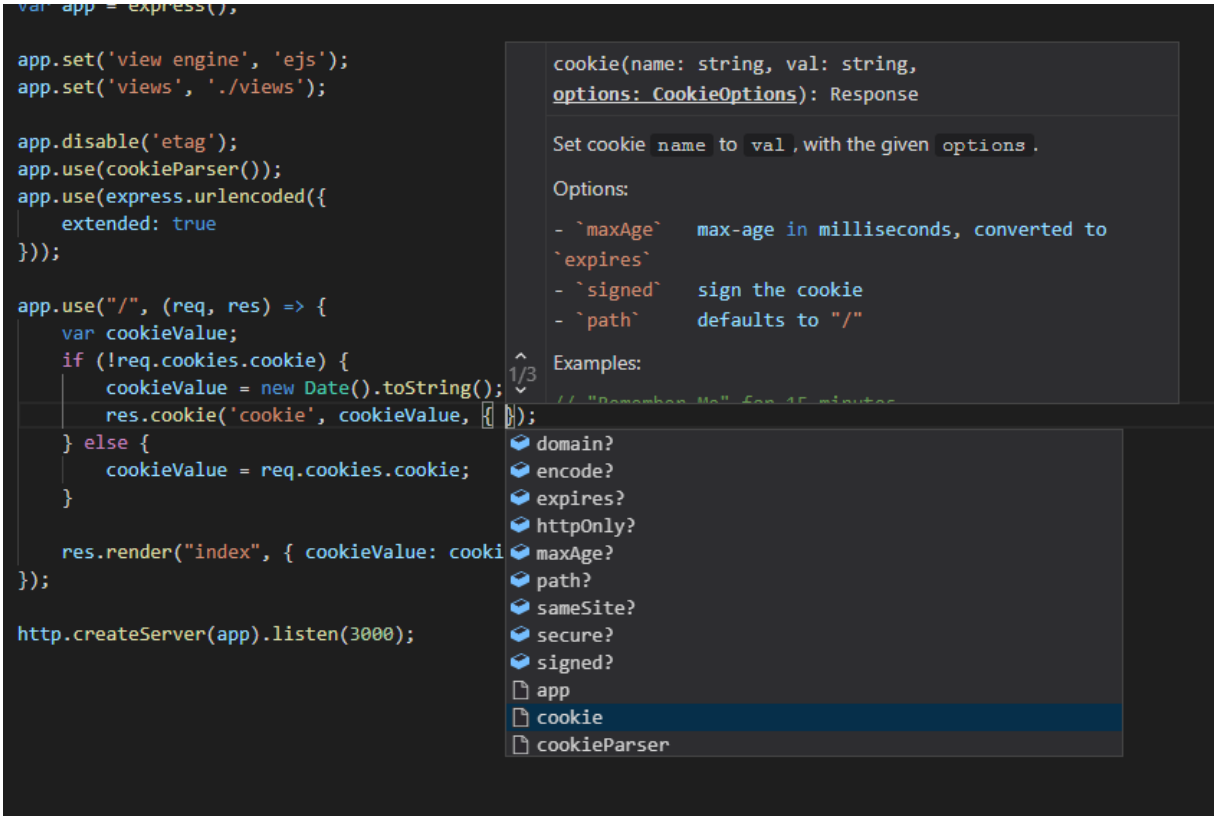
```
var http = require('http');
var express = require('express');
var cookieParser = require('cookie-parser');

var app = express();

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

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

app.use("/", (req, res) => {
    var cookieValue;
    if (!req.cookies.cookie) {
        cookieValue = new Date().toString();
        res.cookie('cookie', cookieValue);
    } else {
        cookieValue = req.cookies.cookie;
    }

    res.render("index", { cookieValue: cookieValue } );
});

http.createServer(app).listen(3000);
```

```
<!-- views/index.ejs -->
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
```

```
</head>

<body>
    <form method="POST">
        Wartość z ciastka: <%= cookieValue %>
        <button>Zapisz</button>
    </form>
</body>

</html>
```

Spośród możliwych parametrów ciastka interesują nas

- **maxAge** umożliwiające sterowanie czasem życia ciastka w przeglądarce, w tym usunięcie ciastka (maxAge: -1)
- **signed** dodające do ciastka podpis **HMAC** wygenerowany z klucza dostarczonego jako argument funkcji **cookieParser()** - dostęp do podpisanych ciastek wymaga odwołania się do właściwości **signedCookies** obiektu **request**




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

var app = express();

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

```
app.disable('etag');
app.use(cookieParser('xzufybuixyfbuxziyfbuzixfuyb'));
app.use(express.urlencoded({
    extended: true
}));

app.use("/", (req, res) => {
    var cookieValue;
    if (!req.signedCookies.cookie) {
        cookieValue = new Date().toString();
        res.cookie('cookie', cookieValue, { signed: true });
    } else {
        cookieValue = req.signedCookies.cookie;
    }

    res.render("index", { cookieValue: cookieValue } );
});

http.createServer(app).listen(3000);
```

5 Obsługa kontenera sesji na serwerze

Kontener sesji to zbiornik na dane po stronie serwera, który znosi ograniczenie rozmiaru ciastek – zamiast transferować cały stan do użytkownika, wysyła mu się w ciasteczku **klucz (identyfikator sesji)**, a stan zapamiętuje na serwerze w zasobniku (na przykład w pamięci) w mapie kojarzącej identyfikatory sesji użytkowników z zapamiętanymi wartościami.

Za obsługę sesji odpowiada middleware express-session.

Co ważne – architektura sesji zakłada możliwość użycia na serwerze *dostawcy* kontenera, którym może być np. zewnętrzna baza danych.

```
var http = require('http');
var express = require('express');
var session = require('express-session');

var app = express();

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

app.disable('etag');

app.use(session({resave:true, saveUninitialized: true, secret:
'qewhiugriasgy'}));

app.use("/", (req, res) => {
    var sessionValue;
    if (!req.session.sessionValue) {
        sessionValue = new Date().toString();
        req.session.sessionValue = sessionValue;
    } else {
        sessionValue = req.session.sessionValue;
    }

    res.render("index", { sessionValue: sessionValue } );
});

http.createServer(app).listen(3000);
```

```
<!-- views/index.ejs -->
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
</head>
```

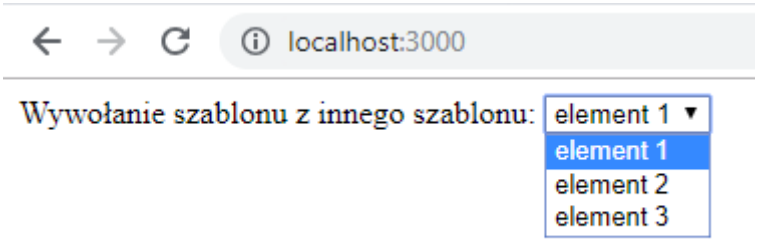
```
<body>
    <form method="POST">
        Wartość z session: <%= sessionValue %>
        <button>Zapisz</button>
    </form>
</body>

</html>
```

6 Złożone szablony z parametrami

Istnieje możliwość wywołania szablonu z innego szablonu. W sposób pokazany poniżej, szablon główny nie korzysta z pomocniczych zmiennych przekazanych z kodu aplikacji.

```
<!-- views/index.ejs -->
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
</head>

<body>
    Wywołanie szablonu z innego szablonu:
    <%- include('select', {
        name: 'combo1',
        options: [
        { value : 1, text : 'element 1' },
        { value : 2, text : 'element 2' },
        { value : 3, text : 'element 3' }
      ]
    }) %>
</body>

</html>
```

```
<!-- select.ejs -->
<select name='<%= name %>'>
    <% options.forEach( option => { %>
    <option value='<%= option.value %>'>
        <%= option.text %>
    </option>
    <% }) %>
</select>
```



Nic jednak nie stoi na przeszkodzie żeby przykład rozbudować. W rzeczywistej aplikacji dane ładowane do formantów pochodzą oczywiście często z zewnętrznych źródeł, na przykład z baz danych.

```
<!-- views/index.ejs -->
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
</head>

<body>
    Wywołanie szablonu z innego szablonu:
    <%- include('select', locals.combo1 ) %>
    <%- include('select', locals.combo2 ) %>
</body>

</html>
```

```
// app.js
var http = require('http');
var express = require('express');

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

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

    var combo1 = {
        name: 'combo1',
        options: [
        { value : 1, text : 'element 1' },
        { value : 2, text : 'element 2' },
        { value : 3, text : 'element 3' }
      ]
```

```
    };
    var combo2 = {
        name: 'combo2',
        options: [
        { value : 4, text : 'element 4' },
        { value : 5, text : 'element 5' },
        { value : 6, text : 'element 6' }
      ]
    }

    res.render('index', { combo1, combo2 });
});

http.createServer(app).listen(3000);
```

7 Express a TypeScript

TypeScript może być używany do rozwijania aplikacji w node + Express. Poniższy przykład zakłada że aplikacja będzie rozwijana przy pomocy **ts-node**.

Aby przygotować środowisko do pracy należy się upewnić że skonfigurowano następujące elementy.

Po pierwsze, proszę się upewnić że w **tsconfig.json** jest aktywna opcja **sourceMap: true**. Dzięki niej podczas zwykłej kompilacji (przez **tsc**) tworzą się mapy przejść TypeScript – JavaScript, a dzięki mapom możliwe jest debugowanie. W przypadku **ts-node** ta opcja może nie mieć znaczenia ale bezpieczniej jest o niej pamiętać.

Po drugie, proszę się upewnić że **ts-node** jest skonfigurowany w **launch.json**

```
        {
            "type": "pwa-node",
            "request": "launch",
            "name": "Launch Program",
            "skipFiles": [
                "<node_internals>/**"
            ],
            //"program": "${workspaceFolder}\\app.js"
            "args": ["${workspaceFolder}\\app.ts"],
            "runtimeArgs": ["-r", "ts-node/register"],
            "cwd": "${workspaceRoot}",
        }
```

Po trzecie, proszę się upewnić że zainstalowano informacje o typach dla node i express

```
npm i --save-dev @types/node
npm i --save-dev @types/express
```

Po czwarte, proszę pamiętać że w TypeScript trzeba inaczej importować/eksportować informacje między modułami

```
// app.ts
import * as http from 'http';
import express, { Express, Request, Response } from 'express';

var app: Express = express();

app.get('/', (req: Request, res: Response) => {
    res.write('hello world');
    res.end();
});

var server = http.createServer(app);
```

```
server.listen(3000);

console.log( 'started' );
```

Praca w tandemie TypeScript + express pozwala wychwycić na etapie kompilacji część problemów, z których co prawda wiele jest wychwytywanych przez VS kiedy kod pisany jest w Javascript, tyle ze to „wychwytywanie" przez Javascript jest oczywiście bardzo „miękkie" – sygnałem ewentualnego problemu może być co najwyżej brak mechanizmu podpowiedzi w trakcie pisania kodu:

W powyższym przykładzie, programista ma ochotę napisać **req.foo** co być może ma sens, być może nie, brak podpowiedzi sugeruje że prawdopodobnie nie. W każdym z dwóch języków, Javascript i TypeScript, zachowanie będzie różne – w szczególności w takich miejscach jak to, TypeScript nie przepuści **req.foo** bo z niczego nie wynika że obiekt typu **Request** może posiadać takie pole.

W pewnych przypadkach, interfejs programistyczny przygotowany jest na „rozszerzenia" i warto pokazać jak to wygląda na przykładzie typu **Request**.

Już wiemy że **Request** ma właściwości **query, body** i **params**. Każda z tych trzech właściwości to mapa klucz-wartość i każdy typ może być doprecyzowany.

Przykładowo **Request** - jest typem generycznym, dopuszczającym parametry typowe:

```
interface Request<
    P = core.ParamsDictionary,
    ResBody = any,
    ReqBody = any,
    ReqQuery = core.Query,
    Locals extends Record<string, any> = Record<string, any>
> extends core.Request<P, ResBody, ReqBody, ReqQuery, Locals> {}
```

Proszę zwrócić uwagę, że parametry typowe mają tu domyślne wartości. Ich nadpisanie powoduje to doprecyzowanie, na przykład …

```
app.get('/', (req: Request<
    { id: number }, {},
    { username: string },
    { foo: string }
    >, res: Response) => {

    res.write('hello world');
    res.end();
});
```

… rozszerza poszczególne typy o dodatkowe informacje. W tym przypadku

- W **params** pojawia się **id**
- W **body** pojawia się **username**
- W **query** pojawia się **foo**

Pozostaje do rozwiązania problem taki oto – chciałoby się czasem, jak w Javascript, do obiektu o jakimś ustalonym interfejsie dodać własne pole. Na przykład – do obiektu **Request** już lada chwila będziemy chcieli dodać nasze własne pole, **user**.

Są dwa sposoby.

Pierwszy – globalne rozszerzenie definicji typu. W tym podejściu tworzymy plik modułu z definicjami typów (rozszerzenie **.d.ts**) i umieszczamy w projekcie. W teorii może być gdziekolwiek w projekcie, ale konwencja jest taka żeby pilnować porządku i utworzyć folder **/src/types/{foo}** gdzie **foo** jest nazwą biblioteki którą się rozszerza a w nim plik **index.d.ts**. W naszym przypadku byłoby to więc

/src/types/express/index.d.ts

a w nim

```
// src/types/express/index.d.ts
export {}

declare global {
  namespace Express {
    export interface Request {
      user: string;
    }
  }
}
```

(pusta dyrektywa **export** to [znacznik modułu])

Od tego momentu obiekt **Request** ma rozszerzony interfejs.

Drugi sposób – to lokalne rozszerzenie definicji typu, co niestety wymaga rzutowania wewnątrz funkcji. Niestety – ponieważ ładnie byłoby użyć rozszerzonego typu w sygnaturze funkcji zwrotnej, niestety, system typów na to nie pozwoli.

Czyli można tak:

```
type RequestWithUser = Request & { user: string }

app.get('/', (req: Request, res: Response) => {

    const requ = req as RequestWithUser;

    requ.user

    res.write('hello world');
    res.end();
});
```

ale nie tak

```
type RequestWithUser = Request & { user: string }

// błąd kompilacji!
app.get('/', (req: RequestWithUser, res: Response) => {

    res.write('hello world');
    res.end();
});
```

W TypeScript nie jest żadnym problemem typowanie funkcji middleware pisanych „luzem”:

```
const m = (req: Request, res: Response, next: NextFunction) {

}

app.use( m );
```

Proszę zwrócić uwagę na to że gdyby to była funkcja anonimowa, używana wewnątrz **use**, to typy nie są nawet potrzebne, bo są inferowane:

```
app.use( (req, res, next ) => {
```

```
} );
```