Wybrane elementy praktyki projektowania oprogramowania

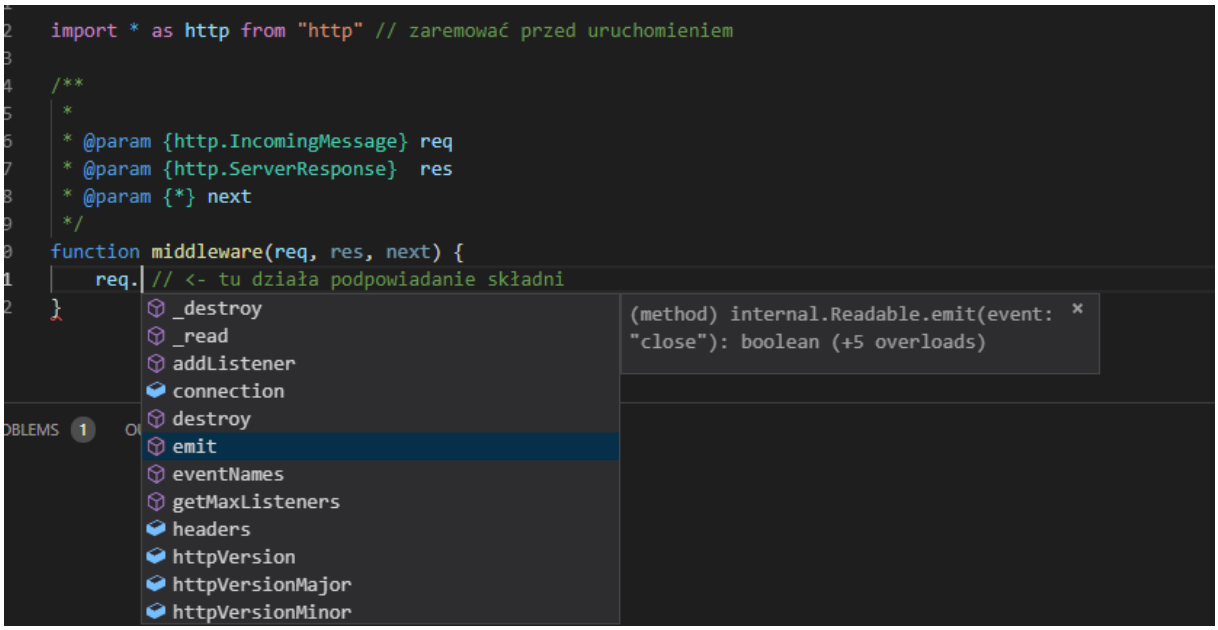
Wykład 09/15
node.js: Express (2)

Wiktor Zychla 2021/2022

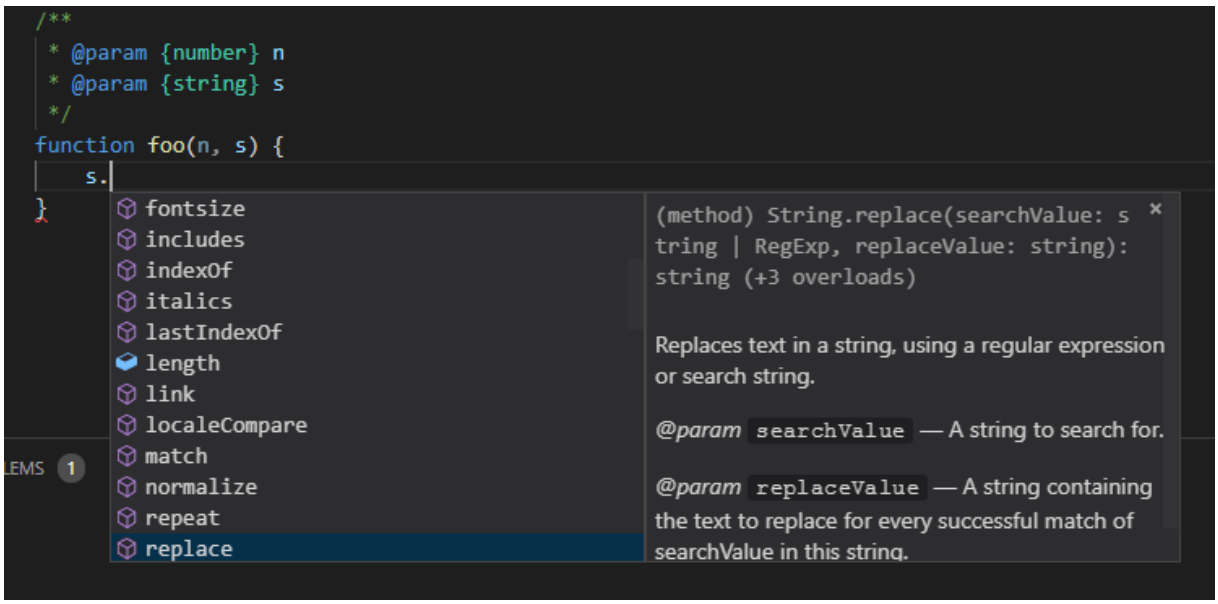
1 Spis treści

2 Silne typowanie argumentów funkcji w VS Code

Na przykładzie funkcji typu middleware można zademonstrować sposób na uzyskanie efektu podpowiadania typu obiektu. W tym celu należy użyć elementów TypeScript (**import**) oraz napisać komentarz w składni JSDoc.



Tego sposobu można użyć do dowolnego typu argumentów



3 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.

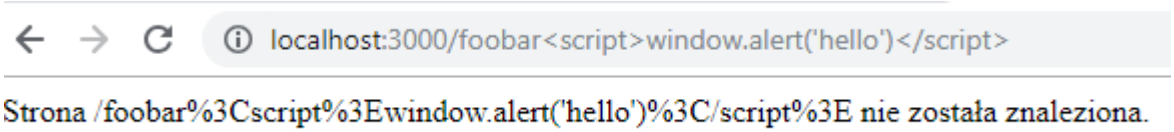
W tym ataku, atakujący preparuje dane, które przesyła na serwer serwerze (na przykład przez url albo nawet przez jakieś pole tekstowe). Dane są spreparowane tak, żeby zawierać poprawny Javascript, który wykona przeglądarka.

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



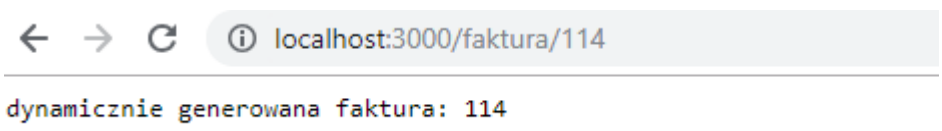
Widać, że użycie znacznika **<%= %>** chroni aplikację przed tego typu zagrożeniem – znacznik ten powoduje bowiem właśnie wzmiankowane wyżej zakodowanie zawartości, w szczególności fragmenty < > znaczników są zakodowane jako %3C i %3E.

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.

4 Użycie parametrów w ścieżkach i zagrożenie Web Parameter Tampering

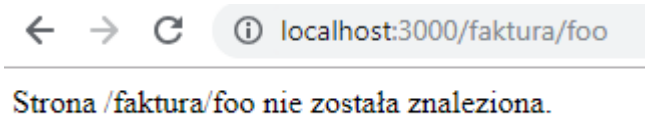
Możliwe jest dynamiczne parametryzowanie ścieżek

```
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);
```



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}`)});
```



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.

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.

5 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.

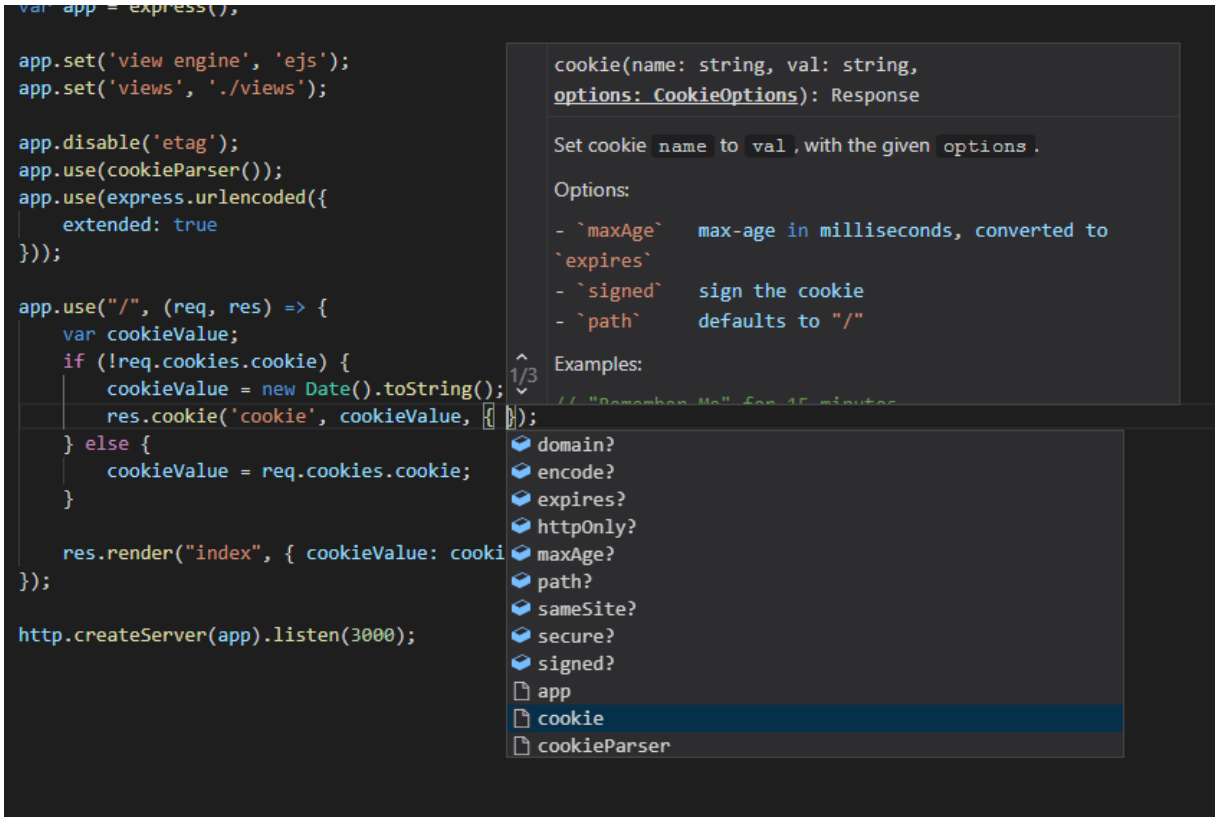
```
var http = require('http');
var express = require('express');
var cookieParser = require('cookie-parser');

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);
```

6 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 klucz w kontenerze, a stan zapamiętuje na serwerze pod kluczem. 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>
```

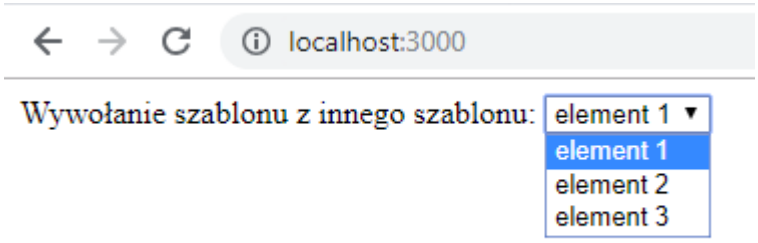
```
    <form method="POST">
        Wartość z session: <%= sessionValue %>
        <button>Zapisz</button>
    </form>
</body>

</html>
```

7 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);
```