Wybrane elementy praktyki projektowania oprogramowania
Wykład 08/15
node.js: Express

Wiktor Zychla 2023/2024

1 Spis treści

2 Express

Framework Express jest najpopularniejszym i najprzystępniejszym frameworkiem do wytwarzania w node.js aplikacji internetowych opartych o klasyczny schemat Request-Reply. Nie jest oczywiście jedynym frameworkiem i warto zapoznać się z alternatywami (koa, sails.js – ekosystem jest tu dość rozwinięty).

O innym podejściu do architektury aplikacji internetowych (tzw. Single-Page Applications), w którym strony nie „przeładowują się" z każdą interakcją użytkownika, będziemy jeszcze rozmawiać. To podejście to frameworki takie jak React czy Angular.

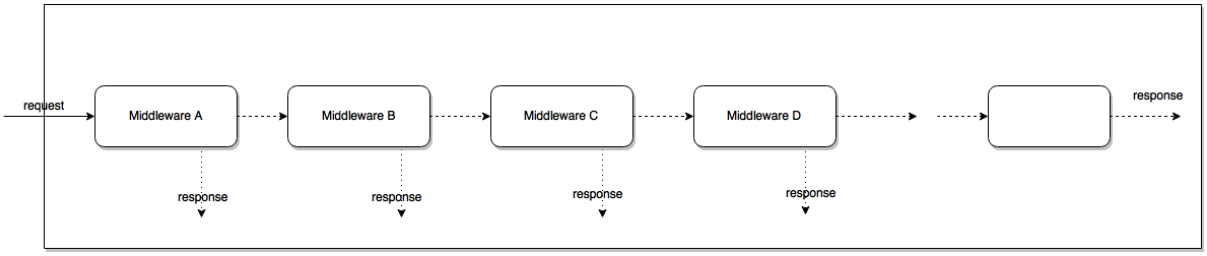
W przykładzie aplikacji od której zaczniemy wykład (w notkach z poprzedniego wykładu) ujawniają się najpoważniejsze problemy „naiwnego" podejścia.

Przypomnijmy więc dlaczego jest ono niewystarczające i potrzebne jest lepiej zorganizowane narzędzie:

- Wsparcie dla routingu - obsługa żądań do wielu adresów - w wersji bez frameworka byłby to duży "if" w funkcji serwera
- Rozdzielenie części imperatywnej (kod) od deklaratywnej (html) - w wersji bez silnika od biedy można użyć szablonowanych napisów, ale już z pętlą (np. renderowanie wierszy tabelki) byłby problem
- Wsparcie parsowania adresu (Query String) i parsowania parametrów POST
- Wsparcie dla tworzenia ciastek
- Wsparcie dla obsługi sesji po stronie serwera
- Wsparcie dla szablonowania renderowania zawartości z ochroną przed atakiem Cross-site scripting (XSS)

3 Warstwa kontroli - Middleware

Architektura aplikacji Express oparta jest o pojęcie middleware. Middleware to funkcja która obsługuje żądanie i która może delegować część pracy do kolejnej funkcji typu middleware. Obsługa pojedynczego żądania jest w związku z tym wywołaniem funkcji, która może wywołać inną funkcję, w ten sposób tworząc możliwy *łańcuch* wywołań



Rysunek 1 https://dzone.com/articles/understanding-middleware-pattern-in-expressjs

Celem takiej architektury jest możliwe rozdzielenie obsługi autentykacji od obsługi renderowania od obsługi błędów itp.

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

var app = express();

app.use( (req, res, next) => {
    res.write("1");
    next();
    res.write("3");
    res.end();
});

app.use( (req, res, next) => {
    res.write("2");
});

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

Do obsługi błędów służy przeciążenie funkcji middleware z czterema argumentami. W poniższym przykładzie pokazano jak Express odróżnia wywołanie przekazujące błąd od zwykłego – przekazanie błędu wymaga przekazania jednego parametru do **next**.

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

var app = express();

app.use( (req, res, next) => {
    if ( true ) // jakiś warunek błędu
```

```
        next("description");
    else
        res.end("poprawne działanie");
});

app.use( (err, req, res, next) => {
    res.end( `Error handling request: ${err}`);
});

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

4 Warstwa widoku - EJS

EJS (Embedded Javascript) jest jednym z wielu silników renderowania dla Express. Zwalnia on z konieczności ręcznego zarządzania szablonami „stron”. Wymaga foldera w którym zapisane będą widoki.

Taka architektura, w której silnik szablonów stron jest niezależny od silnika samej aplikacji, zasługuje na podkreślenie.

Przy okazji warto zapoznać się z tym jak VS Code implementuje system pomocy dla edycji HTML, Emmet.

Przy okazji proszę zwrócić uwagę na to jak konfiguruje się

- Wybór silnika renderowania – parametr **view engine**
- Lokalizację (folder) z widokami – parametr **views**

```
<!-- 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>
    hello world
</body>
</html>
```

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

var app = express();

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

app.use( (req, res) => {
    res.render('index');
});

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

Warto też zwrócić uwagę na to że domyślne rozszerzenie stron (**.ejs**) można zmienić na **.html** z dodatkową prostą konfiguracją

```
app.engine('html', ejs.renderFile);
app.set('view engine', 'html');
```

Najważniejszą cechą EJS jest możliwość mieszania elementów deklaratywnych i imperatywnych wewnątrz widoków, w tym m.in. deklarowanie zmiennych lokalnych.

- ogranicznikami struktury imperatywnej są znaczniki <% i %>
- wypisanie wartości możliwe jest dzięki znacznikom <%= i %>
- wypisanie wartości zakodowanej (tzw. HTML encoding) to <%- i %>

```
<!-- 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>
    <% var foo = 'bar' %>

    <% for ( var i=0; i<5; i++ ) { %>
        <div>
        Element <%= i %>
        </div>
    <% } %>

    Zmienna foo: <%= foo %>
</body>
</html>
```

EJS pozwala również na przekazanie *modelu* z definicji funkcji middleware do widoku:

```
<!-- 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>
    Username: <%= username %>
</body>
```

```
</html>
```

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

var app = express();

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

app.use( (req, res) => {
    res.render('index', {username: 'foo'});
});

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

Uwaga! Jeśli widok odwołuje się do zmiennej która nie występuje w modelu to przy renderowaniu widoku pojawia się błąd. Prawidłowy sposób sprawdzenia istnienia zmiennej wymaga użycia jej pełnej nazwy, w Express jest to postać **locals.{nazwa_zmiennej}**. Ten sposób renderowania warunkowego pojawia się w aplikacjach często – tak renderuje się na przykład na przykład informacja o błędzie walidacji.

```
    <div class="foo">
        <% if (locals.username) { %>
        Username: <%= username2 %>
        <% } %>
    </div>
```

Wzorzec który się tu pojawia nosi w inżynierii oprogramowania nazwę **MVC** – Model-View-Controller. Formalnie występują tu trzy elementy:

- kontroler – jest to kod **Express** który obsługuje żądania przychodzące do serwera z przeglądarki
- widok – jest to widok **EJS** który ostatecznie trafi do użytkownika jako wynik jego żądania
- model – są to dane które kontroler buduje na podstawie parametrów żądania i przekazuje do widoku. Dzięki modelowi przekazywanemu z kontrolera, widok może zawierać dane dynamiczne (różne dla różnych użytkowników / żądań)

W wielu interesujących przypadkach, kontroler odwołuje się do pomocniczej bazy danych, dostępnej na serwerze bazy danych obecnym gdzieś w zasięgu serwera obsługującego aplikację (ale niedostępnym bezpośrednio z przeglądarki użytkownika). Kontroler odczytuje dane z bazy, buduje z nich model i przekazuje do widoku.

5 Dodatkowe elementy architektury aplikacji

5.1 Middleware do plików statycznych

Middleware **express.static** pozwala określić folder z którego serwowane są pliki statyczne – w strukturze plików aplikacji jest to zwykle podfolder ale z punktu widzenia http adresowanie jest względne do roota aplikacji. W poniższym przykładzie plik fizycznie znajduje się w **/static/style.css** ale adresowany jest http://localhost:3000/style.css.

```
/* static/style.css */
.foo {
    color: blue
}
```

```
<!-- 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>
    <link rel="stylesheet" href="style.css" />
</head>
<body>
    <div class="foo">
        Username: <%= username %>
    </div>
</body>
</html>
```

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

var app = express();

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

app.use( express.static( "./static" ) );

app.use( (req, res) => {
    res.render('index', {username: 'foo'});
});

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

Uwaga! Express domyślnie wymusza na przeglądarce keszowanie zawartości plików statycznych dodając nagłówek **ETag.** Jeśli to jest problemem to należy to keszowanie wyłączyć:

```
app.use( express.static('./static', { etag: false } ) );
app.set('etag', false);
```

5.2 Kontrola typu zwracanej odpowiedzi

Za pomocą ustawiania nagłówków możliwe jest kontrolowane typu odpowiedzi. Na przykład wymuszenie potraktowania odpowiedzi jako pliku do pobrania możliwe jest za pomocą nagłówka Content-disposition.

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

var app = express();

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

app.use( (req, res) => {
    res.header('Content-disposition', 'attachment; filename="foo.txt"');
    res.end('tekst');
});

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

5.3 Renderowanie zbiorów 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>
    <table>
        <tr>
            <th>Data</th>
            <th>Kwota</th>
            <th></th>
```

```
        </tr>
        <% przelewy.forEach( przelew => { %>
        <tr>
            <td>
                <%= przelew.data %>
            </td>
            <td>
                <%= przelew.kwota %>
            </td>
            <td><a href='/przelew/<%= przelew.id %>'>Więcej</a></td>
        </tr>
        <% }) %>
    </table>
</body>

</html>
```

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

var app = express();

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

app.use( (req, res) => {
    var przelewy = [
        { kwota : 123, data : '2016-01-03', id : 1 },
        { kwota : 124, data : '2016-01-02', id : 2 },
        { kwota : 125, data : '2016-01-01', id : 3 },
    ];
    res.render('index', {przelewy:przelewy});
});

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

5.4 Obsługa nawigacji między „stronami”

Do przekierowania żądania do innego adresu służy metoda **redirect** obiektu odpowiedzi. Serwer odsyła do przeglądarki status **302 Object moved** z nagłówkiem Location wskazującym na nowy adres. Przeglądarka sama udaje się pod nowy adres z żądaniem typu **GET**

Uwaga! Przekierowania do wskazanego adresu kierowane z serwera pojawiają się w aplikacjach opartych na architekturze Request-Reply częściej niż się może wydawać. Ma to związek z rekomendowanym sposobem obsługi żądań typu POST z formularzy, tzw. Wzorcem POST-REDIRECT-GET (**PRG**).

To bardzo ważny wzorzec. W kolejnym rozdziale zobaczymy przykład

5.5 Odczytywanie parametrów przekazanych w pasku adresowym

Odczyt parametrów przekazanych w pasku adresowym możliwy jest za pomocą właściwości **query** obiektu żądania:

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

var app = express();

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

app.use( (req, res) => {
    var p = req.query.p;
    res.end(`p: ${p}`);
});

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

6 Ścieżki dla żądań różnego typu (GET/POST)

Express umożliwia powiązanie funkcji middleware z typem żądania (GET/POST). Zwyczajowo używa się tego do odróżnienia pierwszego żądania które renderuje zawartość (GET) od kolejnych, które obsługują zwrotnie odesłany formularz (POST).

Należy zwrócić uwagę, że odczyt danych z formularza (POST) wymaga dodatkowego middleware, **express.urlencoded**.

```
<!-- 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">
        <div>
            Username: <input type='text' name='username' value='<%= username
%>' />
        </div>
        <button>Zapisz</button>
        <% if (locals.message) { %>
        <div>
            <%= locals.message %>
        </div>
        <% } %>
    </form>
</body>

</html>

<!-- views/userinfo.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>
    Username: <%= username %>
</body>

</html>
```

```
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}));

// przy nawigacji do witryny z przeglądarki
app.get( '/', (req, res) => {
    res.render('index', {username:''});
});

// po odesłaniu formularza z widoku index.ejs
// proszę prześledzić jak w praktyce działa wzorzec Post-Redirect-Get
app.post('/', (req, res) => {
    var username = req.body.username;
    // tu walidacja
    // może być prosta, typu czy niepuste
    // może być złożona, np. czy określony format
    if ( username && username.length > 5 ) {
        res.redirect('/userinfo?username='+username);
    } else {
        res.render('index', {
            username:username,
            message: 'Nazwa użytkownika musi być dłuższa niż 5 znaków'
        });
    }
});

// po przekierowaniu z poprzedniej strony
app.get('/userinfo', (req, res) => {
    var username = req.query.username;
    res.render('userinfo', {username})
});

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

[redacted]