

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

Wiktor Zychła 2019/2020

---

1	Spis treści	
2	Express.....	2
3	Middleware .....	3
4	EJS.....	5
5	Dodatkowe elementy architektury aplikacji .....	7
5.1	Middleware do plików statycznych.....	7
5.2	Kontrola typu zwracanej odpowiedzi.....	8
5.3	Renderowanie zbiorów danych.....	8
5.4	Obsługa nawigacji między „stronami”.....	9
5.5	Odczytywanie parametrów przekazanych w pasku adresowym .....	9
6	Ścieżki dla żądań różnego typu (GET/POST).....	10

## 2 Express

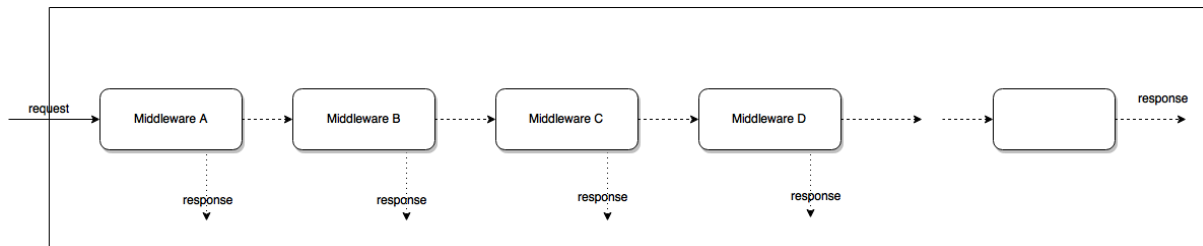
Framework [Express](#) jest najpopularniejszym i najprzystępniejszym frameworkiem do wytwarzania aplikacji internetowych w node.js. W przykładzie aplikacji od której zaczniemy wykład (w notkach z poprzedniego wykładu) ujawniają się najpoważniejsze problemy takiego „naiwnego” podejścia.

Podsumujmy 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
- 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 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](#).

```
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 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.

```
<!-- 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);
```

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);

```

## 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);
```

## 5.2 Kontrola typu zwracanej odpowiedzi

Za pomocą ustawiania nagłówków możliwe jest kontrolowanie 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>
    <% przelew.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**

#### 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. Zwyczajowo używa się tego do odróżnienia pierwszego żądania które renderuje zawartość od kolejnych, które obsługują zwrótnie odesłany formularz:

```
<!-- 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>  
  </form>  
</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}));  
  
app.get('/', (req, res) => {
```

```
    res.render('index', {username:''});
  });

app.post('/', (req, res) => {
  var username = req.body.username;
  res.render('index', {username:username});
});

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