ZAAWANSOWANE TECHNIKI WWW (WFAIS.IF-C112)

(zajęcia 17.12.2015 r.)

Dla przypomnienia uruchamianie serwera http podającego statyczną treść polegało na stworzeniu skryptu wraz z osadzonym kodem statycznej strony HTML i utworzeniem funkcji obsługującej żadania przychodzące do serwera:

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

var html = '<html>'+
           '<head>'+
           '<meta charset="UTF-8">'+
           '<title>To jest strona testowa</title>'+
           '</head>'+
           '<body>'+
           '<h1>Witaj świecie w NODE.JS</h1>'+
           '<h2>to jest strona statyczna</h2>'+
           '</body>'+
           '</html>';


var server = http.createServer(function (req, res) {
      res.writeHead(200, {'Content-Type': 'text/html'});
      res.end(html);
});

server.listen(1337, '127.0.0.1');
console.log('Server running at http://127.0.0.1:1337/');
```

W ramach tego kodu można tworzyć statycznie działające witryny internetowe bez konieczności instalacji serwera Apache.

Dodatkowo w celu zapewnienie sterowalności aplikacją i odpowiedzi na żadania przychopdzące od klientów w postaci adresów URL można wykorzytać informacje przekazywaną w nagłówku o wpisanym adresie i tym samym np. przygotowac podstrony serwisu:

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

http.createServer( function(req,res){
  if( req.url == '/'){
        res.writeHead(200,{'Content-Type': 'text/html'});
        res.end(' index ');
  }
  else if( req.url == '/podstrona1'){
        res.writeHead(200,{'Content-Type': 'text/html'});
        res.end('podstrona1');
  }
}).listen(1337);
```

Instalacja środowiska Express.js

Node.JS umożliwa również tworzenie złożonych i zaawansowanych aplikacji “webowych” w architekturze MVC (Model-View-Controller). Dziś poznamy kompletne narzędzia i metody do tworzenia aplikacji w tej architekturze w środkowisku NODE.JS.

Do tworzenia aplikacji w architekturze MVC w środkowsku NODE.JS służy pakiet “Express” (strona projektu: https://github.com/visionmedia/express). Jest to bardzo rozbudowane narzędzie ułatwiające budowanie aplikacji webowych w NODE.JS. Aby z niego skorzystać należy przeprowadzić instalację pakietu “express” składającą się z kilku kroków:

0) Domyślnie używamy powłoki terminala “bash”. Jeśli nie jest to domyślna powłoka należy się do niej przełączyć wpisując w terminalu polecenie:

```
> bash
```

1) Za pomocą standardowej metody instalacji pakietówm należy zainstalować w środowisku node.js moduł “express”:

```
> npm install express
```

2) Dodoatkowo musimy doinstalować pakiet generatora express:

```
> npm install -g express-generator
```

Na tym etapie środowisko NODE.JS jest gotowe do tworzenie aplikacji w architekturze MVC.

Tworzenie nowego projektu

Aby utworzyć nowy projekt należy wykonać następujące kroki:

1) Utworzyć w kartotece głównej NODE.JS nowy katalog np.:

```
> mkdir hello
```

2) Wygenerować strukturę aplikacji:

```
> express hello
```

3) Wejść do kartoteki hello:

```
> cd hello/.
```

4) Doinstalować starndardowe pakiety:

```
> npm install
```

5) Na tym etapie aplikacja jest gotowa do uruchomienia. Serwer uruchamiamy wydając polecenie:

```
> npm start
```

6) W przeglądarce, po wpisaniu adresu *“localhost:3000”* powinniśmy ujrzeć domyślną stronę projektu express.

7) Dodatkowo w konsoli w której uruchomiliśmy serwer dostajemy “logi” np.:

```
GET / 200 607ms - 270b
GET /stylesheets/style.css 200 8ms - 110b
```

które mówią o czasach wykonywania przychodzących żadań do serwera i pobieranych plikach.

Zgodnie podziałem na trójwarstwową strukturę aplikacji (Model-View-Controller) poszczególne moduły aplikacji są rozproszone w różnych kartotekach. Podstawowa struktura kartotek jest następująca:

```
-rw-r--r--  05-26 10:52 app.js              // główny plik aplikacji
drwxr-xr-x  05-26 10:52 bin                 // katalog z serwerem
```

```
drwxr-xr-x  05-26 10:52 node_modules      // dodatkowe moduły node.js
-rw-r--r--  05-26 10:52 package.json
drwxr-xr-x  05-26 10:52 public            // pliki publiczne np. css
drwxr-xr-x  05-26 10:52 routes            // kontroler
drwxr-xr-x  05-26 10:52 views             // widok
```

Kartoteka “models” nie jest standardowo tworzona.

Architektura trójwarstwowa wymusza podział na akcje i widoki. Gdzie “akcje” (znajdujące się kontrolerze) są odpowiedzialne za wykonywanie operacji i obsługę żadań przychądzących do serwera, a widoki za wyświetlanie danych przychodzących z kontrolera. Standardowo “akcje” przechowywane są w kartotece “routes”, a widoki w kartotece “views”.

Rozpocznijmy od kontrolera: w kartotece “routes” znajdują sie dwa plik jeden o nazwie “index.js” drugi “user.js”. Zajmijmy się tym pierwszym, który odpowiada za obsługę strony głównej w naszej aplikacji. Jego struktura jest bardzo prosta i zawiera tylko jedną funckję obsługującą żądania GET przychodzące do serwera.

Dygresja:

GET -> następuje kiedy wchodzimy na stonę (wpisujemy adres w przeglądarce) i w odpowiedzi na żadanie tego typu serwer zwraca treść do wyświetlania.

POST -> następuje kiedy przesyłamy dane do serwera (np. wysyłając dane wpisane do formularza) które mają zostać przetworzone.

```
/* GET home page. */
router.get('/', function(req, res) {
  res.render('index', { title: 'Express' });
});
```

W tej funkcji obsługujemy żadanie GET przychodzące od klienta (z przeglądarki), wymuszące wyświetlenie strony głównej aplikacji. W odpowiedzi na przychodząse żadanie “req” formowana jest odpowiedz “res”, która renderuje stonę “index.html” na podstawie widoku znajdującego się w kartotece “views” (ale o tym za chwile). Przenalizujmy, jakie wartości przymuje metoda “render”.
Jako pierwszy argument podajemy nazwę pliku widoku który ma zostać wyświetlony w wyniku realizacji tego żadania, a drugi parametr to obiekt z danymi przekazywanymi do widoku które mają zostać wyświetlone ostatecznie na stronie. Można to porównać do funkcji która zwraca wartość (tu w postaci złożonego obiektu) do innej funkcji.

Przejdzmy teraz do widoku. W oparciu o architekturę MVC środowisko node.js wraz z modułem “express” wymusza wprowadzenie nowej formuły tworzenie widoków. Widoki poszczególnych strone przechowywane są w plikach “jade”. Pliki te nie mają postaci standardowego kodu html, pewnego rodzaju kodu “semi-html”. Strona zbudowana jest w oparciu o głowny widok przechowywany w pliku “layout.jade” oraz plików dodatkowych obsługujących poszczególne strony będące rozszerzeniem widoku głownego. W najprostszym ujęciu plik “layout” przechowuje część “head” strony, a pliki z konkretnymi podstronami zwierają tylko sekcję “body” która jest dynamicznie podmieniana w zależności od wysłanego żadania do serwera:

```
doctype html
html
  head
    title= title
    link(rel='stylesheet', href='/stylesheets/style.css')
  body
    block content
```

W klasycznym pliku kodzie html składnia powyższa składnia jade odpowiadałaby:

```
<!DOCTYPE html>
<html>
<head>
     <title> ….. </title>
     <link rel="stylesheet" href="/stylesheets/style.css" />
</head>
<body>
    ……..
</body>
</html>
```

Jak widać składnia “semi-html” jade, jest pozbawiony znaków określających znaczniki oraz jest również samo uzupełniający się ponieważ nie ma znaczników zamykających!!!!
Przykład widoku wyświetlajacego stronę główną jest umieszczony w pliku “index.jade”:

```
extends layout

block content
  h1= title
  p Welcome to #{title}
```

Polecenie “extends layout” mówie serwerowi, że ten plik rozszerza funkcjonalność głównego widoku umieszczonego w pliku layout.

Dodatkowo w obu przypadkach widoku głównego, jak i widoku rozszerzającego widać również jak należy odbierać zmienne przekazywane przez w obiekcie funkcji render. Jak pamiętamy przekasywaliśmy w kontrolerze objekt:

```
{ title: ‘Express’}
```

Jest on teraz wyświetlany w dwóch miejscach w elemencie <h1> oraz <p>. W obu przypadkach dostęp do obiektu jest inny, jednak są to dwie równoważne metody. Z doświadczenie polecam metodę drugą czyli **`#{ nazwa zmiennej }`**.

Zapisywanie znaczników w JADE:

Przykład 1. (który już znamy):

`<div>`
` Element blokowy`
`</div>`
	`div`
	`  | Element blokowy`

Przykład 2.

`<html>`
`  <head>`
`  <title> Jade </title>`
`  </head>`
`  <body>`
`  <p>`
`  JADE`
`  </p>`
`  </body>`
`</html>`
	`html`
	`...head`
	`.....title Jade`
	`...body`
	`.....p`
	`.....| JADE`

W przykładzie powyżej kropki oznaczają wcięcia zrobione za pomocą spacji. Alternatywnie można używać “tabulatora” (stałego odstępu). Nalezy jednak w pojedyńczym pliku .jade używać spacji lub tabulacji (nigdy obu). Powyższy przykład pokazuje również niezwykłą zwięzłość tego typu kodowania.

Wewnątrz tekstu języka JADE można używać znaczników HTML !!!!!

Przykład 3.

`Plain text can include`
`<strong>html</strong>`
`<p>`
`  It must always be on its own line`
`</p>`
	`| Plain text can include`
	`<strong>html</strong>`
	`p`
	`
		It must always be on its own line`

Znaczniki HTML zostaną przesłane do pprzegląderki gdzie zostaną zinterpretowa

Przykład 4 - Lista

`<ul>`
`  <li>Item A</li>`
`  <li>Item B</li>`
`</ul>`
	`ul`
	`  li Item A`
	`  li Item B`
	`  li Item C`

Tworzenie atrybutów znaczników:

`<a href="google.com">Google</a>`
`<a href="google.com"`
`class="button">Google</a>`
	`a(href='google.com') Google`
	`a(class='button', href='google.com')`
	`Google`

Klasy:

`<a class="button"></a>` `<div class="content"></div>`	`a.button` `.content`

Identyfikatory:

`<a id="main-link"></a>` `<div id="content"></div>`	`a#main-link` `#content`

Prosta logika w JADE:

JADE umożliwia wykonywanie bardzo prostych operacji wyliczeniowych:

`<ul>`
`  <li>1</li>`
`  <li>2</li>`
`  <li>3</li>`
`  <li>4</li>`
`  <li>5</li>`
`</ul>`
	`ul`
	`  each val in [1, 2, 3, 4, 5]`
	`  li= val`

Możliwe jest wykonywanie prostych zadań które mają być wykonane w pętli:

`<li>item</li>` `<li>item</li>` `<li>item</li>`	`- for (var x = 0; x < 3; x++)` `  li item`

Dostęna jest również instrukcja warunkowa wyboru:

`<p>you have 10 friends</p>`	`- var friends = 10`
	`case friends`
	`  when 0`
	`  p you have no friends`
	`  when 1`
	`  p you have a friend`
	`  default`
	`  p you have #{friends} friends`

Przekazywanie zaminnych przez adres - metoda GET w node.js.

W kilku sytuacjach niezbędne jest jawne przekazanie wartości zmiennych przez adres (metoda GET), podobanie jak to było w PHP. Jako, że w tym przpadku metoda wysyłamy zmienne to GET możemy posłużyć się funckja: “router.get()”.
Jak już wiemy w metodzie “.get()” pierwszym aprametrem jest adres obsługiwanego żądania. W tym miejscu możemy również przekazywać wartości zmiannych.

Przykład:
Chcemy przekazać 4 następujące zmienne o wartościach całkowitych: a=4, b=5, c=6, d=7. Aby to uczynić musimy w odpowiedni sposób spreparować maskę adresu dla metody “.get()”. Zakładamy że nasz funkcja będzie obsługiwała docelowo adres “/test” oraz w odpowiedzi będzie rederować strone widoku o nazie “testowy.jade”. W standardowym wypadku szablon naszej funkcji będzie następujący:

```
router.get('/test', function(req, res){
                var obj = 0;
                res.render('testowy', obj);
});
```

Teraz musimy tak zmodyfikowac maskę obsługiwanego adresu aby można było podać zmienne (oczywiście nie wpisujemy ich na sztywno w adresie!). Zmiane w adresie są symbolizowane przez:

```
:nazwa_zmiennej          // dwukropek i nazwa zmiennej
```

przykładowo:

```
router.get('/test/:a', function(req, res)
```

gdzie "**:a**" jest nazwą naszej zmiennej którą będziemy mogli posługiwać się do odebrania wartości. Jesmy mamy tych zmiennych wiecej oddzialamy je przecinkami:

```
 router.get('/test/:a,:b,:c,:d', function(req, res)
```

Możliwe są również, inne kombinacje np:

```
router.get('/test/:a/:b,:c/:d', function(req, res)
router.get('/test/:a/:b/:c/:d', function(req, res)
router.get('/test/:a/:b,:c,:d/:e/:f/:g,:h', function(req, res)
```

W adresach można przesyłać dowolne wartości tj. liczby, znaki etc.

Aby odebrać wartości ze zmiennych adresowych należy posługiwać się obiektem żadania i nazwą zmiennej zdefiniowanej w adresie:

```
req.params.nazwa_zmiannej;
```

np:

```
req.params.a;
req.params.b;
```

Wykorzystanie tej funkcjonalności najczęściej polega na wpisaniu na sztywno np. odnośnikach wartości zmiennych. W widoku ma to np następującą postać:

```
a(href="/test/4,5,6,7") kliknij mnie
```

Po kliknięciu w link zostanie wywołana metoda get obsługująca wywołany adres.

Przekazywanie zaminnych przez frmularze - metoda POST w node.js.

Zacznimy od zdefiniowania w "routes/index.js" nowej funkcji która będzie odpowiadała za wyświetlanie naszej strony oraz przetworzenie danych wpisanych do formularza.
Standardowo w tym pliku mamy objeekt który obsługuje nam zapytania do serwera żadające wyświetlanie danej strrony:

```
/* GET home page. */
router.get('/', function(req, res) {
  res.render('index', { title: 'Express' });
});
```

W tym wypadku w odpowiedzi renderowany jest szablon zapisany w pliku "views/index.jade", będacy rozszerzeniem domyślnego wyglądu strony. Aby sobłużyć formularz obsługujący dane musimy posłuzyć

się do tego celu metodą odbierającą dane z formularza czyli “POST” (wyjaśnienie różnicy pomiędzy GET a POST było przedstawione w poprzednich matariałach).
Szkielet funkcji jest następujący:

```
router.post( ‘[link-wyswietlany]’, function(req,res){
    res.render(' [nazwa.jade] ',{ } );
});
```

W naszym przypadku chialibyśmy utworzyć formularz który posiada jedno pole typu “text” do którego wpisujemy dane, a następnie te dane są wyświetlane jak tytuł strony (w nagłówku h1) strony wyświetlonej po przeładowaniu. Załużmy że strona po przeładowaniu będzie się nazywac “studentuj”, a do szablonu jade przekazywana będzie wartośc pola z formularza. Do dobierania danych z pól formularza, posługujemy się obiektem “req” który jest generowany w momencie wysyłania żadania do serwera. Dostęp do wszystkich pól formularza jest możliwy przez obiekt body i podanie nazwy danego pola w formularzu np:. req.body.student. Pełny kod funkcji renderującej stonę na podstawie danych z formularza jest następujący:

```
router.post('/studentuj',function(req,res) {
 var student = {
    name: req.body.student  //nazwa pola
   };
res.render('index', {title: req.body.student});
});
```

Musimy odpowiednio zmodyfikować plik “index.jade” tak aby zawierał sam formularz:

```
extends layout

block content
  h1= title
  p Welcome to #{title}
  form(method="POST" action="/studentuj")
    input(type="text" name="student")
    input(type="submit")
```

Formularz ten “POSTuje” (czyli wsysyła dane) pod adres “/studentuj” który własnie jest obsługiwany w kontrolerze przez wcześniej napisaną funkcję.

Zadanie:
Proszę przymyśleć zadaną konstrukcję obsługi formularza w architekturze “View-Controller” i spróbować stworzyć projekt który odwzorowuje omówiony przykład wraz ze swoim autorskim formularzem.