Programowanie Aplikacji Internetowych

Laboratorium nr 3

Projektowanie aplikacji w JavaScript z wykorzystaniem wzorca MVC

Poniżej znajdują się zadania, które należy wykonać w ramach laboratoriów, a następnie sporządzić sprawozdanie w formie archiwum .zip. Plik archiwum powinien mieć nazwę zgodną ze wzorem: PAI_Lab<nr_laboratoriu>_<pierwsza_litera_nazwiska>.<nazwisko_bez_polskich_znakow>.zip, np. PAI_Lab3_J.Kowalski.zip. W archiwum powinna znajdować się poniższa struktura katalogów i plik:

```
\todo_projekt
    | - \todo
    |     | - todo.html
    |     | - todo.ejs
    |     \ -todo.js
    | - can.custom.js
    \ - jquery-1.xx.x.js
```

Zadania **muszą** być wykonywane w zadanej kolejności – od 1 do .

Zadanie 1 Wprowadzenie do canJS

Zadaniem, które należy wykonać w ramach laboratoriów jest stworzenie interaktywnej listy TODO. Projekt zaczynamy od stworzenia foldera roboczego o nazwie */todo_projekt*. W następnej kolejności umieszczamy w nim dwa pliki .js, które znajdują się pod adresami: http://code.jquery.com/jquery-1.10.2.min.js oraz http://canjs.com/release/2.0.0/can.jquery.js. W następnej kolejności tworzymy katalog *todo_projekt/todos*, a w nim pliki *todos.html:*

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="pl" lang="pl">
  <head>
    <title>CanJS: Prosta lista TODO</title>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
  </head>
<body>
   <script src='../jquery-1.10.2.min.js'></script>
   <script src='../can.jquery.js'></script>
   <script src='todos.js'></script>
</body>
</html>
```

oraz *todos.js*:

```

```

W pasku przeglądarki wpisujemy `http://127.0.0.1/todo_projekt/todos/todos.html` i uruchamiamy narzędzia deweloperskie w przeglądarce (wciskamy F12[1]). Przechodzimy do konsoli JS i wpisujemy:

```
can
```

Powinien zostać utworzony obiekt.

W konsoli wpisujemy:

1 jeżeli w przeglądarce FireFox nie mamy zainstalowanego pluginu FireBug to instalujemy go uprzednio.

```
var Todo = function() {}
Todo.prototype.author = function() { return "Michal" }
```

definiujemy prototyp naszej klasy Todo oraz dodajemy funkcję składową, która zwraca nam autora,

```
var todo = new Todo();
todo.author()
```

tworzymy obiekt naszej klasy i wywołujemy funkcję składową zwracającą nam autora. Jest to tradycyjny obiektowy JavaScript. Sprawdźmy teraz jakie możliwości daje nam biblioteka canJS:

```
var Todo = can.Construct({
count : 0
},
{
author : function() { return "Michal"; }
})
```

tworzy prototyp klasy Todo,

```
Todo.count
```

sprawdzamy wartość składnika statycznego klasy,

```
var todo = new Todo()
todo.author()
```

tworzymy obiekt klasy Todo i za jego pośrednictwem wywołujemy funkcję składową zwracającą nam autora. Mechanizm dziedziczenia jest równie prosty:

```
var PrivateTodo = Todo({
isPrivate : function() { return true; }
})
```

tworzymy prototyp klasy potomnej,

```
var private = new PrivateTodo()
private.isPrivate()
private.author()
```

tworzymy obiekt potomnej klasy i za jego pośrednictwem wywołujemy nową i odziedziczoną funkcję składową. Definicję konstruktora naszej klasy umieszczamy w funkcji inicjującej (init):

```
var Todo = can.Construct({
count : 0
},
{
init : function(name, author) {
this.name = name;
this.authorName = author;
this.constructor.count++;
},
author : function() { return this.authorName; }
})
```

jak widzimy w powyższym kodzie do zmiennej statycznej klasy możemy odwoływać się za pośrednictwem *this.constructor*,

```
var todo = new Todo('naczynia', 'Michal')
```

```
todo.count
todo.author()
```

tworzymy obiekt i sprawdzamy jego elementy składowe.

Przejdźmy teraz do wzorca MVC i zacznijmy od sprawdzenia jakie możliwości daje nam canJS przy tworzeniu modelu:

```
var Todo = can.Model({
   findAll   :  'GET /todos',
   findOne   :  'GET /todos/{id}',
   create    :  'POST /todos',
   update    :  'PUT /todos/{id}',
   destroy   :  'DELETE /todos/{id}'
} , {})
```

na początku tworzymy funkcję konstruującą model

```
var todo = new Todo({name: 'kup mleko', complete: false})
```

i ponownie tworzymy instancję naszej klasy – naszego modelu. Sprawdzimy teraz czy poprawnie działa nasz model:

```
todo.save()
```

Czas w końcu umieścić trochę kodu w pliku *todo_projekt/todos/todos.js*:

```
(function(){
   var TODOS = [{ id: 1, name: 'pozmywaj'},
                {id: 2, name: 'uprasuj koszule'},
                {id: 3, name: 'zrób obiad'}];
   can.fixture('DELETE /todo_projekt/todos/{id}', function(){
      return {};
   });
   can.fixture('PUT /todos/{id}', function(request){
      $.extend( TODOS[ (+request.data.id)-1 ], request.data );
      return {};
   });
   can.fixture('POST /todos', function(request){
      var id = TODOS.length + 1;
      TODOS.push( $.extend({id: id}, request.data) );
      return {id: id};
   });
   can.fixture('GET /todos', function(){
      return TODOS;
   });
   can.fixture('GET /todos/{id}', function(request){
      return TODOS[(+request.data.id)-1];
   });
})();
```

```
var Todo = can.Model({
   findAll   :   'GET /todos',
   findOne   :   'GET /todos/{id}',
   create    :   'POST /todos',
   update    :   'PUT /todos/{id}',
   destroy   :   'DELETE /todos/{id}'
} , {});
```

Po odświeżeniu strony w przeglądarce (`http://127.0.0.1/todo/todos/todos.html`) w konsoli JS wpisujemy:

```
Todo.findAll( {} , function( todos ) {
   console.log( todos[0].name )
})
```

aby otrzymać wszystkie elementy (a wyświetlić na konsoli tylko pierwszy),

```
Todo.findOne( {id:2} , function( todos ) {
   console.log( todos.name )
})
```

aby otrzymać wybrany (drugi) element,

```
var todo = new Todo({name: "odkurz w jadalni"})
todo.save(function( todo ){
   console.log(todo.name)
})
```

aby dodać nowy element oraz

```
todo.id
```

aby sprawdzić czy rzeczywiście zwiększyła się liczba elementów,

```
todo.attr('name', 'umyj szklankę')
todo.save()
```

aby wprowadzić i zapisać zmiany,

```
var todo = new Todo({name: "*$#&(*##"})
todo.save()
todo.destroy()
```

aby stworzyć nowy element, zapisać go i usunąć.

Zadanie 2 Widoki

Czas wyświetlić 'coś' na naszej stronie. W tym celu posłużymy się szablonem EJS. EJS jest domyślnym językiem szablonów w CanJS, który zapewnia w połączeniu z obserwatorem mechanizm wiązania na bieżąco. EJS jest bardzo łatwy w użyciu, do tworzenia szablonów wykorzystuje język HTML w połączeniu z kilkoma 'magicznymi' znacznikami w miejscach gdzie chcemy uzyskać dynamiczne zachowanie. Wyróżniamy pięć rodzajów magicznych znaczników:

<% %> - uruchamia dowolny kod JavaScript umieszczony pomiędzy tymi znacznikami;

<%= %> - wykonuje wyrażenie JavaScript oraz umieszcza wynik w kodzie HTML szablonu (zastępując specjalne znaki „<” i „&” ciągiem znaków odpowiednio < oraz &).

<%== %> - wykonuje wyrażenie JavaScript oraz umieszcza wynik w kodzie HTML szablonu (nie zastępując specjalnych znaków).

<%% %> - dowolny kod JavaScript umieszczony pomiędzy tymi znacznikami załącza jako tekst w szablonie.

<%# %> - wstawianie komentarzy.

Mechanizm wiązania na bieżąco automatycznie aktualizuje strukturę DOM szablonu EJS, gdy tylko dane, które wykorzystuje szablon ulegną zmianie. Wszystko zawdzięczamy obserwatorowi, który śledzi zmiany w danych (w modelu). Utworzymy teraz plik *todo_projekt/todos/todos.ejs*:

```
<% for( var i = 0; i < this.length; i++) { %>
   <li><%= this[i].name %></li>
<% } %>
```

Następnie musimy wprowadzić kilka zmian w pliku *todo_projekt/todos/todos.js*, dopisujemy na końcu pliku:

```
Todo.findAll( {}, function( todos ) {
   var frag = can.view('todos.ejs', todos)
   $("#todos").html( frag );
})
```

Ostatecznie musimy w pliku *todo_projekt/todos/todos.html* dopisać element o id=”todos”:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="pl" lang="pl">
  <head>
    <title>CanJS: Prosta lista TODO</title>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
  </head>
<body>
   <ul id='todos'></ul>
   <script src='../jquery-1.10.2.min.js'></script>
   <script src='../can.custom.js'></script>
   <script src='todos.js'></script>
</body>
</html>
```

Na koniec odświeżamy stronę w przeglądarce (http://127.0.0.1/todo/todos/todos.html) .

Zadanie 3 Dynamiczne dodawanie/usuwanie/edytowanie elementów listy

Zacznijmy od małej zmiany, która udostępni nam obiekt todos z naszego pliku *todo_projekt/todos/todos.js* w konsoli. W tym celu do naszej funkcji Todo.findAll() dopisujemy:

```
Todo.findAll( {}, function( todos ) {
   var frag = can.view('todos.ejs', todos)
   $("#todos").html( frag );
   window.todos = todos;
})
```

Następnie odświeżamy stronę i w konsoli JS wpisujemy:

```
todos[0].attr("name")
```

otrzymamy pierwszy element naszej listy. Zmieńmy go:

```
todos[0].attr("name", "teraz pozmywaj")
todos.push(new Todo({name: "nakarm psa"}))
```

niestety treść strony się nie zmieniła. Aby treść zmieniała się dynamicznie musimy wprowadzić kilka zmian w naszym pliku *todo_projekt/todos/todos.ejs*:

```
<% this.each(function(todo) { %>
   <li><%= todo.attr('name'); %></li>
<% }) %>
```

Zastępując pętlę for iteratorem jQuery.each mamy pewność, że iterować będziemy za każdym razem po wszystkich elementach przekazywanych nam przez funkcję can.view w jako parametr todos. Teraz po odświeżeniu strony ponownie spróbujmy wprowadzić zmiany na naszej liście. W konsoli JS wpisujemy:

```
todos[0].attr("name", "teraz pozmywaj")
todos.push(new Todo({name: "nakarm psa"}))
Todo[2].destroy()
```

Ostatecznie wprowadzimy specjalną konstrukcję jQuery do szablonu EJS w pliku *todo_projekt/todos/todos.ejs*:

```
<% this.each(function(todo) { %>
   <li <%= (el) -> el.data('todo', todo) %> >
      <%= todo.attr('name'); %>
   </li>
<% }) %>
```

Dzięki temu będziemy mogli tworzyć instancję z elementu. Po odświeżeniu strony wpisujemy w konsoli:

```
var todo = $('li:first').data('todo')
todo.destroy()
```

Zadanie 4 Kontroler

Na początku musimy edytować plik *todo_projekt/todos/todos.js*. Powinien wyglądać w następujący sposób (pogrubioną czcionką zaznaczone są fragmenty, które uległy zmianie):

```
(function(){
   var TODOS = [{ id: 1, name: 'pozmywaj'},
                {id: 2, name: 'uprasuj koszule'},
                {id: 3, name: 'zrób obiad'}];
   can.fixture('DELETE /todo/todos/{id}', function(){
      return {};
   });
   can.fixture('PUT /todo/todos/{id}', function(request){
      $.extend( TODOS[ (+request.data.id)-1 ], request.data );
      return {};
```

```
    });
    can.fixture('POST /todo/todos', function(request){
        var id = TODOS.length + 1;
        TODOS.push( $.extend({id: id}, request.data) );
        return {id: id};
    });
    can.fixture('GET /todo/todos', function(){
        return TODOS;
    });
    can.fixture('GET /todo/todos/{id}', function(request){
        return TODOS[(+request.data.id)-1];
    });
})();

var Todo = can.Model({
    findAll   :  'GET /todo/todos',
    findOne   :  'GET /todo/todos/{id}',
    create    :  'POST /todo/todos',
    update    :  'PUT /todo/todos/{id}',
    destroy   :  'DELETE /todo/todos/{id}'
} , {});

var Todos = can.Control({
    defaults: {
        view: 'todos.ejs'
    }
},
{
    init: function( element, options ){
        Todo.findAll( {}, function( todos ) {
            element.html( can.view(options.view , todos) );
        });
    },
    "li click"   :  function( li , event ){
        var todo = li.data('todo');
        li.trigger('selected', todo);
    }
});

var todosControl = new Todos("#todos");
```

Przeanalizujmy zmieniony kod. Tworzymy klasę Todo, która będzie naszym kontrolerem. Na początku ustawiamy domyślne parametry zmiennej *options.view*. Następnie definiujemy funkcję konstruktora, w której definiujemy statyczną funkcję Todo.findAll(). Funkcja ta do elementu struktury DOM, przekazanego do konstruktora jako parametr będzie dołączać kod HTM wygenerowany przez can.view

na podstawie naszego szablonu *todo_projekt/todos/todos.ejs*. Dalej zdefiniowana jest funkcja obsługująca zdarzenie kliknięcia na element listy. Na dole pliku tworzymy obiekt naszego konstruktora.

Teraz zmodyfikujemy plik *todo_projekt/todos/todos.ejs* tak aby na widoku dodany był link umożliwiający usuwanie wybranego elementu listy:

```
<% this.each(function(todo) { %>
   <li <%= (el) -> el.data('todo', todo) %> >
      <%= todo.attr('name'); %>
      <a href="javascript://" class='destroy'>X</a>
   </li>
<% }) %>
```

Odświeżamy stronę i w konsoli JS wpisujemy:

```
$('#todos').bind('selected',
   function( ev, todo ){
   console.log("selected", todo.name );
})
```

Następnie kliknij na dowolny element listy. Czas dodać mechanizm usuwania elementów listy za pomocą X. W tym celu modyfikujemy klasę kontrolera w pliku *todo_projekt/todos/todos.js:*

```
...
var Todos = can.Control({
   defaults: {
      view:  'todos.ejs'
   }
},
{
   init: function( element, options ){
      Todo.findAll( {}, function( todos ) {
         element.html( can.view(options.view , todos) );
      });
   },
   "li click"    :  function( li , event ){
      var todo = li.data('todo');
      li.trigger('selected', todo);
   },
   "li .destroy click":  function( el, event ){
      var todo = el.closest('li').data('todo');
      todo.destroy();
      event.stopPropagation();
   }
});

var todosControl = new Todos("#todos");
```

W dopisanym fragmencie kodu zdefiniowaliśmy funkcję obsługującą zdarzenie kliknięcia na X. W

ciele funkcji pobieramy odpowiedni (najbliższy) element listy i usuwamy go. Na koniec zatrzymujemy propagację zdarzenia do elementów rodzica. Odświeżamy teraz stronę i klikamy na X przy dowolnym elemencie listy.

Dopiszemy teraz obsługę pola tekstowego, które pozwoli nam na edycję elementów na liście. W tym celu na końcu pliku *todo_projekt/todos/todos.js* musimy dopisać kontroler pola edycji:

```
var Editor = can.Control({
   todo: function( todo) {
      this.options.todo = todo;
      this.on();
      this.setName();
      this.element.show();
   },
   setName: function() {
      this.element.val( this.options.todo.name )
   },
   "change": function() {
      var todo = this.options.todo;
      todo.attr( "name", this.element.val() )
            .save();
   },
   "{todo} destroyed": function() {
      this.element.hide();
   }
})

var editorControl = new Editor("#editor");

$("#todos").bind("selected", function( ev , todo ){
   editorControl.todo( todo );
})
```

Kontroler na pierwszy rzut oka nie wyróżnia się niczym specjalnym w porównaniu do kontrolera listy Todos. Należy jednak zauważyć, że kontroler Edytor posiada funkcję o nazwie todo, w której ciele znajduje się wywołanie funkcji this.on(). Funkcja ta pozwala na odwiązanie i ponowne przywiązanie wszystkich uchwytów zdarzeń kontrolera poprzez wywołanie jej na nim. Jest to przydatne, gdy kontroler wykrywa zdarzenia w danym modelu, a chcemy aby zaczął wykrywać zdarzenia w innym modelu. Jakie to ma zastosowanie w naszym przypadku? Takie, że nasze pole edycji elementu będzie obsługiwać element, który uprzednio wybierzemy kliknięciem myszy na nim. Nasze pole tekstowe będzie pojawiać się po wybraniu dowolnego elementu listy i znikać po usunięciu go. Na koniec tworzymy instancję naszego kontrolera edycji elementu i przypinamy funkcję obsługi zdarzenia do naszej listy elementów.

Ostatecznie musimy jeszcze w pliku *todo_projekt/todos/todos.html* dodać pole tekstowe:

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="pl" lang="pl">
  <head>
    <title>CanJS: Prosta lista TODO</title>
```

```
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
  </head>
<body>
   <ul id='todos'></ul>
   <input id="editor" type='text' />
   <script src='../jquery-1.10.2.min.js'></script>
   <script src='../can.custom.js'></script>
   <script src='todos.js'></script>
</body>
</html>
```

Odświeżamy stronę i klikamy na dowolny element listy.

Zadanie 5 Trasowanie (Routing)

Rdzeniem funkcjonalności trasowania w CanJS jest mechanizm can.route. Jest to specjalny obserwator, który aktualizuje window.location.hash, gdy właściwości can.route zmienią się oraz aktualizuje właściwości can.route, gdy window.location.hash się zmieni. Można przekazać szablon do can.route, dzięki któremu będzie can.route tłumaczył adresy URL do swoich wartości. Jednakże, jeśli nie przekażemy żadnego trasowania do can.route, to po prostu can.route będzie kodować adres URL w standardowej notacji. Zobaczmy jak to działa, w konsoli JS wpisujemy:

```
location.hash
```

- pusto, a can.route?

```
can.route.attr()
```

- pusty obiekt. Jeżeli zmienimy location.hash:

```
location.hash = "#!id=7"
can.route.attr()
```