Baza Programisty

Web development
Receptury nowej generacji

Brian P. Hogan
Chris Warren
Mike Weber
Chris Johnson
Aaron Godin

Helion

Tytuł oryginału: Web Development Recipes
Tłumaczenie: Łukasz Piwko

ISBN: 978-83-246-5149-8

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: http://helion.pl (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/twstnr.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/twstnr
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.
Printed in Poland.

Spis treści

Receptura 10. Tworzenie szablonów HTML przy użyciu systemu Mustache

Problem

Do utworzenia naprawdę wyjątkowego interfejsu potrzebne jest zastosowanie zarówno dynamicznych, jak i asynchronicznych technik programowania. Dzięki Ajaksowi i takim bibliotekom jak jQuery możemy modyfikować interfejs użytkownika bez zmieniania jego kodu HTML, ponieważ potrzebne elementy dodamy za pomocą JavaScriptu. Elementy te zazwyczaj dodaje się, stosując technikę konkatenacji łańcuchów. Powstały w wyniku tego kod jest, niestety, zagmatwany i łatwo w nim popełnić błąd. Pełno w nim mieszających się ze sobą pojedynczych i podwójnych cudzysłowów oraz niekończących się łańcuchów wywołań metody `append()` z biblioteki jQuery.

Składniki

- jQuery
- *Mustache.js*

Rozwiązanie

Na szczęście, są nowoczesne narzędzia, takie jak Mustache, dzięki którym możemy pisać *prawdziwy* kod HTML, renderować przy jego użyciu dane oraz wstawiać go do dokumentów. Mustache to system szablonów HTML dostępny w kilku popularnych językach programowania. Implementacja JavaScript pozwala na pisanie widoków dla klienta przy użyciu czystego kodu HTML całkowicie oddzielonego od kodu JavaScript. Można w nim używać także instrukcji logicznych i iteracji.

Mustache pozwala uprościć tworzenie kodu HTML podczas generowania nowej treści. Składnię tego narzędzia poznamy na przykładzie aplikacji JavaScript do zarządzania produktami.

Obecnie aplikacja ta umożliwia dodawanie nowych produktów do listy. Ponieważ wszystkie żądania są obsługiwane przez JavaScript i Ajax, w przykładzie tym używamy naszego standardowego serwera roboczego. Gdy użytkownik wypełni formularz dodawania nowego produktu, wysyła do serwera żądanie zapisania

tego produktu i wyrenderowania nowego produktu na liście. Aby dodać produkt do listy, musimy zastosować konkatenację łańcuchów, której kod jest zagmatwany i nieelegancki:

mustache/submit.html
```
var newProduct = $('<li></li>');
newProduct.append('<span class="product-name">' +
  product.name + '</span>');
newProduct.append('<em class="product-price">' +
  product.price + '</em>');
newProduct.append('<div class="product-description">' +
  product.description + '</div>');

productsList.append(newProduct);
```

Aby użyć systemu *Mustache.js*, wystarczy go tylko załadować na stronę. Jedną z wersji tego pliku znajdziesz w pakiecie kodu towarzyszącym tej książce, ale możesz też pobrać jego najnowszą wersję z serwisu GitHub[7].

Renderowanie szablonu

Aby przerobić naszą istniejącą aplikację, przede wszystkim musimy dowiedzieć się, jak wyrenderować szablon przy użyciu Mustache. Najprostszym sposobem na zrobienie tego jest użycie funkcji `to_html()`.

```
Mustache.to_html(templateString, data);
```

Funkcja ta przyjmuje dwa argumenty. Pierwszy jest łańcuchem szablonowego kodu HTML, w którym ma odbywać się renderowanie, a drugi to dane, które mają zostać do tego szablonu wstawione. Zmienna `data` jest obiektem, którego klucze w szablonie zostają zamienione na lokalne zmienne. Spójrz na poniższy kod:

```
var artist = {name: "John Coltrane"};
var rendered = Mustache.to_html('<span class="artist name">{{ name }}</span>',
  artist);
$('body').append(rendered);
```

Zmienna `rendered` zawiera nasz ostateczny kod HTML zwrócony przez metodę `to_html()`. Aby umieścić własność `name` w naszym kodzie HTML, użyliśmy znaczników Mustache ograniczonych podwójnymi klamrami. W klamrach tych umieszcza się nazwy własności. Ostatni wiersz kodu dodaje wyrenderowany kod HTML do elementu `<body>`.

Jest to najprostsza technika renderowania szablonu przy użyciu Mustache. W naszej aplikacji będzie więcej kodu związanego z wysyłaniem żądania do serwera w celu pobrania danych, ale proces tworzenia szablonu pozostanie bez zmian.

[7] *https://github.com/janl/mustache.js*

Podmienianie istniejącego systemu

Skoro wiemy już, jak się renderuje szablony, możemy z naszego programu usunąć stary kod konkatenacji. Przyjrzymy mu się, aby zobaczyć, co można usunąć, a co trzeba podmienić.

mustache/submit.html

```
function renderNewProduct() {
var productsList = $('#products-list');

var newProductForm = $('#new-product-form');

var product = {};
product.name = newProductForm.find('input[name*=name]').val();
product.price = newProductForm.find('input[name*=price]').val();
product.description =
  newProductForm.find('textarea[name*=description]').val();

var newProduct = $('<li></li>');
newProduct.append('<span class="product-name">' +
  product.name + '</span>');
newProduct.append('<em class="product-price">' +
  product.price + '</em>');
newProduct.append('<div class="product-description">' +
  product.description + '</div>');

  productsList.append(newProduct);

productsList.find('input[type=text], textarea').each(function(input) {
  input.attr('value', '');
  });
}
```

Ten skomplikowany kod bardzo trudno jest rozszyfrować, a jeszcze trudniej go modyfikować. Dlatego zamiast metody `append()` jQuery do budowy struktury HTML użyjemy systemu Mustache. Możemy napisać prawdziwy kod HTML, a następnie wyrenderować dane przy użyciu Mustache! Pierwszym krokiem w kierunku pozbycia się plątaniny JavaScriptu jest zbudowanie szablonu. Później wyrenderujemy go wraz z danymi produktów w jednym prostym procesie.

Jeśli utworzymy element `<script>` z typem treści `text/template`, to będziemy mogli w nim umieścić kod HTML Mustache i pobierać go stamtąd do szablonu. Nadamy mu identyfikator, aby móc się do niego odwoływać z kodu jQuery.

```
<script type="text/template" id="product-template">
<!-- Szablon HTML -->
</script>
```

Następnie napiszemy kod HTML naszego szablonu. Mamy już produkt w postaci obiektu, a więc jego własności w szablonie możemy użyć jako nazw zmiennych:

Jaś pyta:
Czy można używać zewnętrznych szablonów?

Szablony wewnętrzne są poręczne, ale my chcemy oddzielić logikę szablonową od widoków serwera. Na serwerze należałoby utworzyć folder do przechowywania wszystkich plików widoków. Następnie, gdy trzeba wyrenderować jeden z szablonów, pobieramy go za pomocą jQery i żądania `GET`.

```
$.get("http://mojastrona.com/js_views/zewnetrzny_szablon.html",
  function(template) {
    Mustache.to_html(template, data).appendTo("body");
  }
);
```

W ten sposób można serwować widoki serwera osobno od widoków klienta.

```
<script type="text/template" id="product-template">
  <li>
    <span class="product-name">{{ name }}</span>
    <em class="product-price">{{ price }}</em>
    <div class="product-description">{{ description }}</div>
  </li>
</script>
```

Mając gotowy szablon, możemy wrócić do poprzedniego kodu, aby zmienić sposób wstawiania kodu HTML. Możemy pobrać odwołanie do szablonu przy użyciu jQuery i za pomocą metody `html()` pobrać treść wewnętrzną. Później trzeba jeszcze tylko przesłać kod HTML i dane do Mustache.

```
var newProduct = Mustache.to_html( $('#product-template').html(), product);
```

Wynik tego powinien być już zadowalający, chociaż gdy nie ma żadnego opisu, pole opisu nie powinno być widoczne. Innymi słowy, jeśli nie ma opisu, nie powinniśmy renderować jego elementu `<div>`. Na szczęście, w Mustache można używać instrukcji warunkowych. Przy ich użyciu sprawdzimy, czy opis istnieje, i jeśli tak, to wyrenderujemy dla niego element `<div>`.

```
{{#description}}
  <div class="product-description">{{ description }}</div>
{{/description}}
```

Przy użyciu tego samego operatora Mustache wykona iterację po tablicy. System sprawdzi, czy dana własność jest tablicą, i jeśli tak, to automatycznie zastosuje iterację.

Stosowanie iteracji

Ponieważ udało nam się wymienić znaczną część istniejącego kodu budującego nowy produkt, postanowiliśmy, że większą część logiki aplikacji napiszemy w JavaScripcie. Zamienimy stronę indeksową, na której wyświetlane są produkty wraz z uwagami, na kod JavaScript, który będzie robił to samo. Utworzymy tablicę produktów na jednej z własności obiektu danych i każdy produkt w tej tablicy będzie miał własność `notes`. Własność `notes` będzie tablicą, po której iteracja będzie się odbywać wewnątrz szablonu.

Najpierw pobierzemy i wyrenderujemy produkty. Przyjmujemy założenie, że serwer zwraca tablicę w formacie JSON wyglądającą tak:

```
$.getJSON('/products.json', function(products) {
  var data = {products: products};
  var rendered = Mustache.to_html($('#products-template').html(), data);
  $('body').append(rendered);
});
```

Teraz musimy zbudować szablon do wyrenderowania produktów. W Mustache iterację po tablicy wykonuje się, przekazując tę tablicę operatorowi `#`, np. `{{#zmienna}}`. Wewnątrz iteracji własności, które wywołujemy, znajdują się w kontekście obiektów w tablicy.

mustache/index.html

```
<script type="text/template" id="products-template">
  {{#products}}
    <li>
      <span class="product-name">{{ name }}</span>
      <em class="product-price">{{ price }}</em>
      <div class="product-description">{{ description }}</div>
      <ul class="product-notes">
        {{#notes}}
          <li>{{ text }}</li>
        {{/notes}}
      </ul>
    </li>
  {{/products}}
</script>
```

Teraz nasza strona indeksowa może być w całości generowana w przeglądarce przy użyciu szablonów i Mustache.

Szablony JavaScript są doskonałym narzędziem pozwalającym dobrze zorganizować kod aplikacji JavaScript. Nauczyłeś się renderować szablony, używać instrukcji warunkowych oraz stosować iterację. *Mustache.js* jest prostym narzędziem pozwalającym pozbyć się konkatenacji łańcuchów i budować strukturę HTML w czytelny i zgodny z semantyką sposób.

Kontynuacja

Szablony Mustache pozwalają zachować przejrzystość nie tylko kodu działającego po stronie klienta, ale również serwerowego. Istnieją implementacje tego systemu w językach Ruby, Java, Python, ColdFusion i wielu innych. Więcej informacji na ten temat można znaleźć na stronie Mustache[8].

Mustache można zatem używać jako systemu szablonów zarówno przy budowie frontu, jak i zaplecza aplikacji. Gdybyśmy na przykład mieli szablon Mustache reprezentujący wiersz tabeli HTML i użyli go wewnątrz pętli budującej tabelę przy wczytywaniu strony, to tego samego szablonu moglibyśmy też użyć w celu dodania wiersza do tej tabeli po udanym wykonaniu żądania Ajax.

Zobacz również

- Receptura 11.: „Dzielenie treści na strony”
- Receptura 13.: „Tworzenie interaktywnych interfejsów użytkownika przy użyciu biblioteki Knockout.js”
- Receptura 14.: „Organizacja kodu przy użyciu biblioteki Backbone.js”
- Receptura 20.: „Budowanie witryny przy użyciu JavaScriptu i CouchDB”

Receptura 11. Dzielenie treści na strony

Problem

Aby zaoszczędzić użytkownikom nadmiaru treści na jednej stronie, a przy okazji nie przeciążyć serwerów, należy ustalić limit ilości danych, jaka może zostać wyświetlona na jednej stronie. Najczęściej w tym celu dodaje się mechanizm dzielenia stron. Jego działanie polega na tym, że wyświetlana jest tylko część zawartości strony i użytkownik może w razie potrzeby przejrzeć także pozostałe części. Początkowo wyświetlana jest tylko mała cząstka tego, co można obejrzeć.

W toku ewolucji stron internetowych ich twórcy spostrzegli, że użytkownicy większość czasu spędzają na przeglądaniu treści w sposób liniowy. Najchętniej

[8] *http://mustache.github.com/*

przeglądaliby całe listy danych, aż do znalezienia szukanych informacji albo osiągnięcia końca zbioru. Naszym zadaniem jest umożliwienie im takiego przeglądania i jednocześnie uniknięcie przeciążenia serwera.

Składniki

- jQuery
- *Mustache.js*[9]
- QEDServer

Rozwiązanie

Paginacja to dobry sposób na zapanowanie nad zasobami, który dodatkowo ułatwia korzystanie ze strony użytkownikowi. Zamiast zmuszać użytkownika do wybrania następnej strony wyników i wczytywać cały interfejs od nowa, następną stronę wczytujemy w tle i dodajemy ją do bieżącej strony, podczas gdy użytkownik zbliża się do jej końca.

Chcemy umieścić na stronie listę naszych produktów, ale jest ich za dużo, aby wyświetlić je wszystkie naraz. Dlatego zastosujemy paginację z ograniczeniem polegającym na wyświetlaniu maksymalnie 10 produktów naraz. Żeby jeszcze bardziej ułatwić życie użytkownikom, pozbędziemy się przycisku *Następna strona* i zamiast tego będziemy automatycznie wczytywać kolejne strony, gdy uznamy, że należy to zrobić. Od strony użytkownika będzie to wyglądało tak, jakby cała lista była dostępna na stronie od samego początku.

Do budowy prototypu użyjemy QEDServera i jego katalogu produktów. Cały kod źródłowy umieścimy w folderze *public* w przestrzeni roboczej QEDServera. Uruchom QEDServer i utwórz nowy plik o nazwie *products.html* w folderze *public* utworzonym przez QEDServer. Jeśli nie wiesz, czym jest QEDServer, zajrzyj do „Wstępu", w którym znajduje się objaśnienie.

Aby utrzymać porządek w kodzie, użyjemy biblioteki szablonów Mustache, o której była mowa w recepturze 10., „Tworzenie szablonów HTML przy użyciu systemu Mustache". Pobierz tę bibliotekę i umieść ją także w folderze *public*.

Zaczniemy od utworzenia prostego szkieletu strony *index.html* w HTML5. Dołączymy do niej biblioteki jQuery, Mustache oraz plik *endless_pagination.js*, który będzie zawierał nasz kod paginacji.

[9] *http://github.com/documentcloud/underscore/blob/master/underscore.js*

endlesspagination/products.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset='utf-8'>
    <title>Produkty AwesomeCo</title>
    <link rel='stylesheet' href='endless_pagination.css'>
    <script type="text/javascript"
      src="http://ajax.googleapis.com/ajax/libs/jquery/1.7/jquery.min.js">
    </script>
    <script type="text/javascript" src="mustache.js"></script>
    <script src="endless_pagination.js"></script>
  </head>
  <body>
    <div id="wrap">
      <header>
        <h1>Produkty</h1>
      </header>
    </div>
  </body>
</html>
```

W strukturze tej strony umieściliśmy kontener na treść i obraz wirującego kółka widoczny na rysunku 2.8. Animacja ta ma na celu zasygnalizować użytkownikowi, że trwa wczytywanie następnej strony, które powinno się odbywać w tle.

endlesspagination/products.html

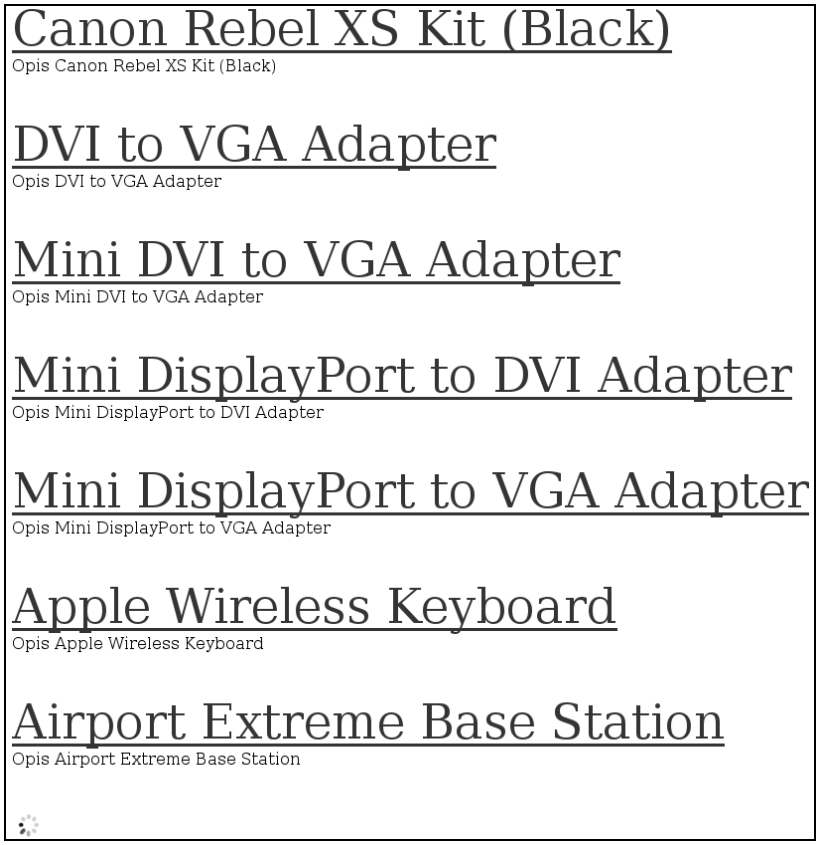
```
<div id='content'>
</div>
<img src='spinner.gif' id='next_page_spinner'>
```

API QEDServer jest tak skonfigurowane, że zwraca stronicowane wyniki i reaguje na żądania JSON. Możemy się o tym przekonać, otwierając adres *http://localhost:8080/Products.json?page=2*.

Wiedząc, jakie informacje otrzymujemy od serwera, możemy rozpocząć budowę kodu, który będzie aktualizował interfejs. W tym celu napiszemy funkcję pobierającą tablicę w formacie JSON, znakującą ją przy użyciu szablonu Mustache i wynik tego działania dodającą na końcu strony. Cały ten kod umieścimy w pliku o nazwie *endless_pagination.js*. Zaczniemy od napisania funkcji pomocniczych. Na pierwszy ogień pójdzie funkcja renderująca odpowiedź w formacie JSON do HTML.

endlesspagination/endless_pagination.js

```
function loadData(data) {
  $('#content').append(Mustache.to_html("{{#products}} \
    <div class='product'> \
      <a href='/products/{{id}}'>{{name}}</a> \
      <br> \
      <span class='description'>{{description}}</span> \
    </div>{{/products}}", { products: data }));
}
```



Rysunek 2.8. Widok dolnej części strony

W procesie iteracji szablon utworzy dla każdego produktu element `<div>`, w którym treść jest nazwą produktu w postaci łącza. Następnie nowe elementy zostaną dodane na końcu listy produktów i pojawią się na stronie.

Następnie, jako że po dotarciu przez użytkownika na koniec strony chcemy wczytać kolejną stronę, musimy znaleźć sposób na określenie, czym jest ta następna strona. W tym celu możemy przechowywać bieżącą stronę jako zmienną globalną, a następnie gdy będziemy gotowi — utworzyć URL dla następnej strony.

endlesspagination/endless_pagination.js

```
var currentPage = 0;
function nextPageWithJSON() {
  currentPage += 1;
  var newURL = 'http://localhost:8080/products.json?page=' + currentPage;

  var splitHref = document.URL.split('?');
  var parameters = splitHref[1];
```

```
  if (parameters) {
    parameters = parameters.replace(/[?&]page=[^&]*/, '');
    newURL += '&' + parameters;
  }
  return newURL;
}
```

Funkcja `nextPageWithJSON()` zwiększa wartość zmiennej `currentPage` i dodaje ją do bieżącego adresu jako wartość parametru `page=`. Zapamiętujemy też wszystkie inne parametry znajdujące się w bieżącym URL. Jednocześnie upewniamy się, że stary parametr `page`, jeśli istnieje, zostanie nadpisany. Dzięki temu otrzymamy właściwą odpowiedź od serwera.

Funkcje wyświetlające nową treść i określające adres następnej strony są już gotowe. Czas w takim razie napisać funkcję pobierającą treść z serwera. W istocie funkcja ta będzie po prostu żądaniem Ajax wysyłanym do serwera. Musimy tylko zaimplementować w niej podstawowy mechanizm zapobiegający wysyłaniu niepotrzebnych żądań. Utworzymy zmienną globalną o nazwie `loadingPage()`, którą zainicjujemy wartością `0`. Przed wykonaniem żądania będziemy ją zwiększać, a po jego zakończeniu — zmniejszać z powrotem. Jest to coś w rodzaju muteksu, czyli mechanizmu blokującego. Bez tej blokady do serwera wysyłane byłyby dziesiątki żądań, a serwer by na nie skwapliwie odpowiadał, mimo że nie o to nam chodziło.

endlesspagination/endless_pagination.js

```
var loadingPage = 0;
function getNextPage() {
  if (loadingPage != 0) return;

  loadingPage++;
  $.getJSON(nextPageWithJSON(), {}, updateContent).
    complete(function() { loadingPage-- });
}

function updateContent(response) {
  loadData(response);
}
```

Po otrzymaniu odpowiedzi na żądanie Ajax przekazujemy ją do funkcji `loadData()`, której definicję przedstawiono wcześniej. Po dodaniu nowej treści przez funkcję `loadData()` aktualizujemy adres URL przechowywany w zmiennej `nextPage`. Jesteśmy gotowi na wykonanie *kolejnego* żądania Ajax.

Mając funkcję żądającą następnej strony, teraz musimy zająć się sprawdzaniem, czy użytkownik w ogóle jest gotowy na wczytanie kolejnej strony. Normalnie wyświetlilibyśmy po prostu łącze *Następna strona*, ale nam chodzi o coś innego. Potrzebujemy funkcji, która będzie zwracać `true`, gdy dolna krawędź okna przeglądarki znajdzie się w określonej odległości od dołu strony.

endlesspagination/endless_pagination.js
```
function readyForNextPage() {
  if (!$('#next_page_spinner').is(':visible')) return;

  var threshold = 200;
  var bottomPosition = $(window).scrollTop() + $(window).height();
  var distanceFromBottom = $(document).height() - bottomPosition;

  return distanceFromBottom <= threshold;
}
```

Na koniec dodajemy procedurę obsługi zdarzenia przewijania kółkiem myszy, która wywołuje funkcję `observeScroll()`. Gdy użytkownik przewinie stronę za pomocą kółka myszy, nastąpi wywołanie funkcji pomocniczej `readyForNextPage()`. Gdy funkcja ta zwróci `true`, wywołamy funkcję `getNextPage()`, aby wykonać żądanie Ajax.

endlesspagination/endless_pagination.js
```
function observeScroll(event) {
  if (readyForNextPage()) getNextPage();
}

$(document).scroll(observeScroll);
```

Część dotyczącą „nieskończonego wyświetlania treści" mamy za sobą, ale przecież kiedyś ta nasza treść jednak się skończy. Po wyświetleniu ostatniego produktu wirujące kółko powinno zostać ukryte, ponieważ jeśli będzie widoczne, użytkownik pomyśli, że albo coś jest nie tak z jego łączem internetowym, albo z naszą stroną. Aby usunąć wirujące kółko, dodamy test, który będzie powodował jego ukrycie, gdy serwer zwróci pustą listę.

endlesspagination/endless_pagination.js
```
function loadData(data) {
  $('#content').append(Mustache.to_html("{{#products}} \
    <div class='product'> \
      <a href='/products/{{id}}'>{{name}}</a> \
      <br> \
      <span class='description'>{{description}}</span> \
    </div>{{/products}}", { products: data }));
  if (data.length == 0) $('#next_page_spinner').hide();
}
```

To wszystko. Gdy dotrzemy do końca listy, wirujące kółko zniknie.

Kontynuacja

Opisana tu technika jest doskonałą metodą wyświetlania długich list danych w sposób zgodny z oczekiwaniami użytkowników. Dzięki podziałowi rozwiązania na funkcje łatwo je będzie przystosować do różnych projektów. Można zmienić

Funkcjonalność przeglądarki IE 8

W przeglądarce IE 8 ten kod nie działa. Przeglądarka ta wymaga, aby nagłówki żądań JSON były w bardzo specyficznym formacie, np. strona kodowa UTF-8 musi zostać wysłana jako utf8. Bez poprawnych nagłówków żądanie Ajax nie powiedzie się i na stronie będzie wyświetlone tylko wirujące kółko. Należy o tym pamiętać podczas pracy z formatem JSON na serwerze i w przeglądarce IE.

wartość zmiennej `treshold`, aby treść była wczytywana wcześniej lub później, lub zmodyfikować funkcję `loadData()`, aby zwracała odpowiedź w formacie HTML lub XML zamiast JSON. A najlepsze jest to, że możemy być spokojni o dostępność naszej strony także wówczas, gdy z jakiegoś powodu biblioteka jQuery nie będzie obsługiwana. Możesz to sprawdzić, wyłączając JavaScript w swojej przeglądarce.

W następnej recepturze pokażemy Ci, jak ulepszyć ten kod poprzez dodanie obsługi zmian adresu URL i przycisku *Wstecz*.

Zobacz również

- Receptura 12.: „Zapamiętywanie stanu w Ajaksie”
- Receptura 10.: „Tworzenie szablonów HTML przy użyciu systemu Mustache”

Receptura 12. Zapamiętywanie stanu w Ajaksie

Problem

Jedną z największych zalet internetu jest możliwość dzielenia się odnośnikami z innymi ludźmi. Jednak wraz z nadejściem ery Ajaksa nie wszystko jest takie proste. Klikając łącze Ajax, nie mamy gwarancji, że spowoduje to zmianę adresu URL w pasku przeglądarki. To nie tylko utrudnia wymianę odnośnikami z innymi ludźmi, ale również powoduje wadliwe działanie przycisku *Wstecz* przeglądarki. Strony zawierające takie łącza nie są dobrymi obywatelami internetu, ponieważ gdy się je wyłączy, nie da się wrócić bezpośrednio do poprzedniego stanu.

Niestety, skrypt paginacji, który napisaliśmy w recepturze 11., „Dzielenie treści na strony", również nie należy do wzorowych obywateli. Gdy użytkownik przewija stronę i przechodzi do kolejnych porcji informacji, adres URL cały czas pozostaje taki sam. A przecież każde wczytanie oznacza nowy stan, w którym prezentowane są inne dane niż bezpośrednio po wczytaniu strony. Gdyby na przykład spodobał się nam produkt ze strony piątej i wysłalibyśmy znajomemu odnośnik do niej, znajomy ten mógłby nie znaleźć tego, o czym mu pisaliśmy, ponieważ zobaczyłby inną listę niż my.

To nie wszystko. Gdy użytkownik kliknie przycisk *Wstecz* przeglądarki na stronie zbudowanej w całości na bazie Ajaksa, to nie przejdzie tam, gdzie by chciał, tylko na poprzednią stronę, z której trafił do nas. Zdziwiony kliknie przycisk *Dalej* i pogubi się całkowicie. Na szczęście, znamy rozwiązanie tych problemów.

Składniki

- jQuery
- *Mustache.js*[10]
- QEDServer

Rozwiązanie

W tej recepturze dokończymy pracę rozpoczętą w recepturze 11., „Dzielenie treści na strony". Mimo iż zastosowana tam metoda ogólnie działa, to jednak ma tę wadę, że uniemożliwia odwiedzającym dzielenie się linkami do stron. Aby spełnić wymogi dobrego projektowania stron i nie utrudniać życia użytkownikom, musimy sprawić, aby nasza lista produktów śledziła swój stan. Innymi słowy, gdy zmieni się strona, na którą patrzymy, wraz z nią powinien zmieniać się adres URL. W specyfikacji HTML5 wprowadzono funkcję JavaScript o nazwie `pushState()`, która w większości przeglądarek pozwala na zmianę adresu URL bez opuszczania strony. To doskonała wiadomość dla programistów stron internetowych! Dzięki temu możemy napisać stronę w całości opartą na Ajaksie, której przeglądanie nigdy nie wymaga wykonywania całego cyklu żądań i przeładowań. Oznacza to, że nie musimy już wczytywać takich zasobów, jak nagłówek i stopka dokumentu, wysyłać w nieskończoność żądań plików graficznych, arkuszy stylów i skryptów JavaScript za każdym razem, gdy przechodzimy na nową stronę

[10] *http://github.com/documentcloud/underscore/blob/master/underscore.js*

w obrębie witryny. A użytkownicy mogą bez problemu przesyłać linki znajomym i nigdy się nie pogubią, w którym miejscu w serwisie aktualnie się znajdują. Najlepsze jest to, że przycisk *Wstecz* przeglądarki również będzie działał poprawnie.

Funkcja pushState()

Funkcja `pushState()` jest jeszcze dopracowywana. Większość starych przeglądarek jej nie obsługuje, ale istnieją rozwiązania awaryjne wykorzystujące część adresu URL za znakiem `#`. Rozwiązania te może nie są eleganckie, ale działają. Poza tym nie chodzi tylko o piękne adresy URL. Internet ma bardzo dobrą pamięć długotrwałą. Stworzono go nie tylko do zabawy i pogaduszek, lecz również po to, aby można było nawet po latach znaleźć stare strony, do których kiedyś utworzyło się łącze, a które zostały przeniesione na nowy serwer (pod warunkiem że ich twórcy są dobrymi obywatelami internetu i stosują poprawne przekierowania HTTP 301). Jeśli będziemy używać znaku # w adresach URL jako tymczasowego rozwiązania dla ważnych informacji, to może się okazać, że nigdy się ich nie pozbędziemy[11]. Ponieważ znaki # z adresów URL nigdy nie są wysyłane do serwera, nasza aplikacja musiałaby dalej przekierowywać ruch po tym, gdy funkcja `pushState()` stanie się standardem.

Uzbrojeni w tę nową wiedzę, zobaczmy, co trzeba zrobić, aby nasza niekończąca się strona z produktami uzyskała świadomość swojego stanu.

Parametry, które trzeba śledzić

Ponieważ nie wiemy, na którą stronę użytkownik wejdzie za pierwszym razem, będziemy śledzić zarówno stronę startową, jak i bieżącą. Jeśli użytkownik wejdzie od razu na trzecią stronę, to chcemy, aby przy kolejnych wizytach mógł na nią wrócić. Jeśli odwiedzający skorzysta z kółka myszy na stronie trzeciej i wczyta kilka kolejnych stron, np. do strony siódmej, to również chcemy o tym wiedzieć. Potrzebujemy sposobu na zapamiętanie strony startowej i końcowej, aby w przypadku odświeżenia użytkownik nie musiał przewijać wszystkiego od początku.

Musimy też znaleźć sposób na wysyłanie startowej i końcowej strony z klienta. Najbardziej oczywistym rozwiązaniem w tym przypadku wydaje się dodanie tych parametrów do adresu URL w żądaniu `GET`. Przy pierwszym wczytaniu strony ustawimy parametr `page` adresu na bieżącą stronę i przyjmiemy założenie, że użytkownik chce obejrzeć tylko tę stronę. Gdy klient przekaże dodatkowo parametr `start_page`, będziemy wiedzieć, że użytkownik chce obejrzeć kilka stron, od

[11] *http://danwebb.net/2011/5/28/it-is-about-the-hashbangs*

start_page do page. Wracając do poprzedniego przykładu, gdybyśmy byli na stronie siódmej, ale zaczęli przeglądanie od strony trzeciej, to nasz adres URL wyglądałby następująco *http://localhost:8080/products?start_page=3&page=7*.

Te parametry powinny nam wystarczyć do odtworzenia listy produktów i pokazania użytkownikowi takiej samej strony za każdym razem.

statefulpagination/stateful_pagination.js

```
function getParameterByName(name) {
  var match = RegExp('[?&]' + name + '=([^&]*)')
    .exec(window.location.search);

  return match && decodeURIComponent(match[1].replace(/\+/g, ' '));
}

var currentPage = 0;
var startPage = 0;

$(function() {
  startPage = parseInt(getParameterByName('start_page'));
  if (isNaN(startPage)) {
    startPage = parseInt(getParameterByName('page'));
  }
  if (isNaN(startPage)) {
    startPage = 1;
  }
  currentPage = startPage - 1;

  if (getParameterByName('page')) {
    endPage = parseInt(getParameterByName('page'));
    for (i = currentPage; i < endPage; i++) {
      getNextPage(true);
    }
  }

  observeScroll();
});
```

Skrypt ten sprawdza parametry start_page i page, a następnie wysyła żądanie odpowiednich stron do serwera. Użyliśmy funkcji bardzo podobnej do getNext
↪Page() z poprzedniej receptury, tylko z obsługą wielu żądań naraz. W odróżnieniu od sytuacji, gdy użytkownik korzysta z kółka myszy i chcemy zapobiec nakładaniu się żądań, w tym przypadku nie przeszkadza nam to, ponieważ wiemy dokładnie, których stron ma dotyczyć żądanie.

Podobnie jak śledziliśmy wcześniej wartość zmiennej currentPage, teraz będziemy śledzić startPage. Parametr ten będziemy pobierać z adresu URL, dzięki czemu będziemy mogli wykonywać żądania stron, które nie były jeszcze wczytywane. Liczba ta nie będzie się zmieniać, ale musi być dodawana do adresu URL i być w nim przy każdym żądaniu nowej strony.

Aktualizowanie adresu URL w przeglądarce

Do aktualizacji adresu URL w przeglądarce napiszemy funkcję o nazwie `updateBrowserUrl()`, wywołującą funkcję `pushState()` oraz ustawiającą parametry `start_page` i `page`. Należy przy okazji pamiętać, że nie wszystkie przeglądarki obsługują funkcję `pushState()`, i przed jej wywołaniem sprawdzać, czy jest zdefiniowana. W tych przeglądarkach nasze rozwiązanie po prostu nie będzie działać, ale powinniśmy przygotowywać nasze aplikacje z myślą o przyszłości.

statefulpagination/stateful_pagination.js

```
function updateBrowserUrl() {
  if (window.history.pushState == undefined) return;

  var newURL = '?start_page=' + startPage + '&page=' + currentPage;
  window.history.pushState({}, '', newURL);
}
```

Funkcja `pushState()` przyjmuje trzy parametry. Pierwszy jest obiekt stanu, który jest w istocie obiektem w formacie JSON. Argument ten moglibyśmy wykorzystać do przechowywania informacji dotyczących stanu, ponieważ w wyniku przewijania od serwera otrzymujemy dane właśnie w formacie JSON. Ponieważ jednak nasze dane są proste i łatwe do pobrania z serwera, wydaje się, że nie jest to warte zachodu. Drugi argument to tytuł nowego stanu. Nie jest on na razie szeroko obsługiwany przez przeglądarki, ale w naszym przypadku to nie problem, bo i tak byśmy tego nie potrzebowali. W związku z tym przekazujemy w nim pusty łańcuch.

W końcu dochodzimy do najważniejszego elementu funkcji `pushState()`. Trzeci parametr określa, co ma się zmienić w adresie. Może to być zarówno bezwzględna ścieżka, jak i zestaw parametrów, które mają zostać zmienione na końcu adresu. Ze względów bezpieczeństwa nie można zmieniać domeny, ale wszystko, co znajduje się za nią — tak. Ponieważ nas interesuje tylko zmienianie parametrów adresu URL, na początku trzeciego parametru funkcji `pushState()` wpisaliśmy znak `?`. Następnie wpisaliśmy ustawienia parametrów `start_page` i `page`. Jeśli parametry te będą znajdować się w adresie, funkcja sama je zaktualizuje.

statefulpagination/stateful_pagination.js

```
function updateContent(response) {
  loadData(response);
  updateBrowserUrl();
}
```

Na koniec, aby nasz mechanizm paginacji zaczął rozpoznawać swój stan, dodaliśmy wywołanie funkcji `updateBrowserUrl()` do funkcji `updateContent()`. Od tej pory użytkownicy mogą bez przeszkód używać przycisku *Wstecz*, aby wyjść ze strony, i przycisku *Dalej*, aby na nią wrócić do tego samego miejsca. Także odświeżenie

strony niczego nie zepsuje. Co jednak najważniejsze, teraz odwiedzający mogą wysyłać znajomym linki do naszych stron. Dzięki ciężkiej pracy programistów przeglądarek internetowych udało nam się sprawić, aby nasza strona indeksowa była dobrym obywatelem internetu.

Kontynuacja

Dodając kolejne skrypty JavaScript i Ajax do swoich stron, powinniśmy mieć świadomość działania używanych interfejsów. Metoda `pushState()` HTML5 i API History pozwalają nam przywrócić normalne działanie kontrolek, do których użytkownicy są przyzwyczajeni. Warstwy abstrakcji, takie jak *History.js*[12], jeszcze to ułatwiają, ponieważ dostarczają eleganckich rozwiązań awaryjnych dla starych przeglądarek.

Opisane przez nas rozwiązania zaczynają też być implementowane w bibliotekach JavaScript, jak np. *Backbone.js*, co oznacza, że możemy spodziewać się jeszcze lepszej obsługi przycisku *Wstecz* nawet na najbardziej skomplikowanych jednostronicowych aplikacjach.

Zobacz również

- Receptura 10.: „Tworzenie szablonów HTML przy użyciu systemu Mustache”
- Receptura 14.: „Organizacja kodu przy użyciu biblioteki Backbone.js”

Receptura 13. Tworzenie interaktywnych interfejsów użytkownika przy użyciu biblioteki Knockout.js

Problem

Programując nowoczesne aplikacje sieciowe, staramy się, aby w reakcji na działania użytkownika aktualizowana była jak najmniejsza część interfejsu. Odwołania do serwera są zawsze czasochłonne, a odświeżanie całej strony może spowodować zniechęcenie użytkownika.

[12] *http://plugins.jquery.com/plugin-tags/pushstate*

Niestety, kod JavaScript używany do implementacji tych mechanizmów często szybko wymyka się spod kontroli. Na początku obserwowanych jest tylko kilka zdarzeń, ale z czasem dodajemy kolejne funkcje zwrotne do aktualizowania różnych obszarów strony i utrzymanie tego wszystkiego w ryzach staje się bardzo kłopotliwe.

Knockout to prosta, a zarazem bardzo funkcjonalna biblioteka, pozwalająca wiązać obiekty z interfejsem i automatycznie aktualizować jedną część interfejsu, podczas gdy zmieniana jest inna część, bez potrzeby używania wielu zagnieżdżonych procedur obsługi zdarzeń.

Składniki

- *Knockout.js*[13]

Rozwiązanie

Knockout.js używa **modeli widoków**, które zawierają logikę widoku związaną ze zmianami interfejsu. Własności tych modeli można wiązać z elementami interfejsu.

Chcemy, aby użytkownicy naszej strony mogli zmieniać liczbę elementów w koszyku i od razu otrzymać zaktualizowaną należną za nie sumę. Do budowy ekranu aktualizacji naszego koszyka możemy wykorzystać modele widoków Knockout. W koszyku każdy produkt będzie przedstawiony w postaci wiersza tabeli. W każdym wierszu będzie się znajdować pole na liczbę egzemplarzy danego produktu oraz przycisk pozwalający usunąć ten produkt z koszyka. Gdy zmieni się liczba egzemplarzy któregoś z produktów, aplikacja natychmiast obliczy nową sumę cząstkową oraz sumę za wszystkie towary. Gdy skończymy pracę, finalny efekt będzie wyglądał jak na rysunku 2.9.

Produkt	Cena	Liczba	Suma	
Macbook Pro 15 inch	1699	1	1699	Usuń
Przejściówka Mini Display Port na VGA	29	1	29	Usuń
Magic Trackpad	69	1	69	Usuń
Klawiatura bezprzewodowa Apple	69	1	69	Usuń
Należność			1866	

Rysunek 2.9. Interfejs koszyka

[13] *http://knockoutjs.com*

Podstawy Knockout

Modele widoków Knockout to zwykłe obiekty JavaScript z własnościami i metodami. Oto prosty obiekt Person z metodami dla imienia, nazwiska oraz imienia i nazwiska.

knockout/binding.html

```
var Person = function(){
  this.firstname = ko.observable("Jan");
  this.lastname = ko.observable("Kowalski");
  this.fullname = ko.dependentObservable(function(){
    return(
      this.firstname() + " " + this.lastname()
    );
  }, this);
};
ko.applyBindings( new Person );
```

Metody i logikę tego obiektu z elementami interfejsu wiążemy przy użyciu atrybutu `data-` języka HTML5.

knockout/binding.html

```
    <p>Imię: <input type="text" data-bind="value: firstname"></p>
    <p>Nazwisko: <input type="text" data-bind="value: lastname"></p>
    <p>Imię i nazwisko:
       <span aria-live="polite" data-bind="text: fullname"></span>
    </p>
```

Gdy zmienimy imię albo nazwisko w jednym z pól, pod spodem zostanie wyświetlone zaktualizowane imię i nazwisko. Ponieważ aktualizacja odbywa się dynamicznie, może sprawiać problemy osobom niewidomym, które korzystają z czytników ekranu. Rozwiązaniem tego problemu jest użycie atrybutu `aria-live`, informującego czytniki, że ta część strony dynamicznie się zmienia.

To był bardzo prosty przykład, więc teraz pokopiemy trochę głębiej i utworzymy jeden wiersz danych naszego koszyka, w którym po zmianie liczby produktów będzie odpowiednio zmieniała się suma należna. Później na tej bazie zbudujemy cały koszyk. Zaczniemy od utworzenia modelu danych.

Pojedynczy wiersz będzie reprezentować obiekt JavaScript o nazwie `LineItem` mający własności `name` i `price`. Utwórz stronę HTML i dołącz do niej bibliotekę *Knockout.js* w sekcji nagłówkowej:

knockout/item.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Aktualizacja liczby produktów</title>
    <script type="text/javascript" src="knockout-1.3.0.js"></script>
  </head>
```

```
  <body>
  </body>

</html>
```

Na dole strony, nad znacznikiem `</body>`, wstaw element `<script>` i wpisz w nim następujący kod:

knockout/item.html
```
var LineItem = function(product_name, product_price){
  this.name = product_name;
  this.price = product_price;
};
```

W JavaScripcie funkcje są konstruktorami obiektów, a więc można ich używać do naśladowania klas. W tym przypadku konstruktor egzemplarza `LineItem` przyjmuje nazwę i cenę.

Teraz musimy poinformować Knockout, że chcemy użyć tej klasy `LineItem` jako naszego modelu widoku, aby jej własności były widoczne dla kodu HTML. W tym celu do skryptu dodajemy poniższy kod.

knockout/item.html
```
var item = new LineItem("Macbook Pro 15", 1699.00);
ko.applyBindings(item);
```

Tworzymy nowy egzemplarz obiektu `LineItem`, ustawiając w nim nazwę i cenę produktu, i wywołujemy na nim metodę Knockout `applyBindings()`. Później to zmienimy, ale na razie wystarczy nam zakodowanie danych na stałe.

Mając obiekt, możemy zbudować interfejs i pobrać z tego obiektu dane. Koszyk zbudujemy na bazie tabeli HTML z elementami strukturalnymi `<thead>` i `<tbody>`.

knockout/item.html
```
    <div role="application">
      <table>
        <thead>
          <tr>
            <th>Produkt</th>
            <th>Cena</th>
            <th>Liczba</th>
            <th>Suma</th>
          </tr>
        </thead>
        <tbody>
          <tr aria-live="polite">
            <td data-bind="text: name"></td>
            <td data-bind="text: price"></td>
          </tr>
        </tbody>
      </table>
    </div>
```

Ponieważ wiersze tabeli są aktualizowane danymi wprowadzanymi przez użytkownika, wierszom tym nadaliśmy atrybut `aria-live`, aby czytniki ekranu wiedziały, że należy się w nich spodziewać zmian. Cały koszyk dodatkowo umieściliśmy w elemencie `<div>` z atrybutem HTML5-ARIA `application`, który informuje czytniki, że jest to aplikacja interaktywna. Więcej informacji na temat tych atrybutów można przeczytać w specyfikacji HTML5[14].

Szczególną uwagę należy zwrócić na poniższe dwa wiersze kodu:

knockout/item.html

```
<td data-bind="text: name"></td>
<td data-bind="text: price"></td>
```

Teraz nasz egzemplarz `LineInstance` jest widoczny globalnie na całej stronie, a więc tak samo widoczne są jego własności `name` i `price`. Te dwa wiersze kodu oznaczają, że tekst (`text`) tych elementów chcemy pobierać z własności o określonych nazwach.

Gdy teraz wczytamy naszą stronę w przeglądarce, to zauważymy, że zaczyna nabierać kształtu oraz że pola nazwy i ceny produktu są wypełnione!

Teraz dodamy pole, w którym użytkownik będzie mógł zmienić liczbę produktów w zamówieniu.

knockout/item.html

```
<td><input type="text" name="quantity"
     data-bind='value: quantity, valueUpdate: "keyup"'>
</td>
```

W bibliotece Knockout do odwoływania się do pól danych w postaci elementów HTML służy parametr `text`, ale elementy formularzy HTML takie jak `<input>` mają atrybut `value`. Dlatego tym razem związaliśmy atrybut `value` z własnością `quantity`.

Własność `quantity` służy nie tylko do wyświetlania danych, lecz również do ich ustawiania. A gdy ustawimy dane, musimy też uruchomić zdarzenia. Do tego celu używamy funkcji Knockout `ko.observable()` jako wartości własności `quantity` naszej klasy.

```
this.quantity = ko.observable(1);
```

Funkcji `ko.observable()` przekazaliśmy domyślną wartość, aby po wyświetleniu strony po raz pierwszy pole tekstowe coś już zawierało.

Teraz możemy już wpisać liczbę, ale chcielibyśmy jeszcze dodatkowo wyświetlić sumę cząstkową dla każdego wiersza. Dodamy do tabeli kolumnę na tę kwotę:

[14] *http://www.w3.org/TR/html5-author/wai-aria.html*

knockout/item.html
```
<td data-bind="text: subtotal "></td>
```

Podobnie jak w przypadku kolumn nazwy i ceny, tekst komórki ustawiamy na wartość własności `subtotal` modelu widoku.

To doprowadziło nas do jednej z najważniejszych części biblioteki *Knockout.js*, metody `dependentObservable()`. Własność `quantity` zdefiniowaliśmy jako obserwowalną, co oznacza, że gdy pole zmieni wartość, zmiana ta będzie zauważona przez inne elementy. Deklarujemy metodę `dependentObservable()`, która będzie wykonywać kod w reakcji na zmianę wartości obserwowanego pola, oraz przypisujemy tę metodę do własności naszego obiektu, aby można ją było związać z naszym interfejsem użytkownika.

```
this.subtotal = ko.dependentObservable(function() {
  return(
    this.price * parseInt("0"+this.quantity(), 10)
  ); //<label id="code.subtotal">
}, this);
```

Ale skąd metoda `dependentObservable()` wie, które pola obserwować? Przegląda obserwowalne własności, które wymieniamy w definiowanej funkcji. Ponieważ mnożymy cenę i liczbę, Knockout śledzi obie te własności i wykonuje kod, gdy którakolwiek z nich się zmieni.

Metoda `dependentObservable()` przyjmuje także drugi parametr, który określa kontekst dla własności. Ma to związek ze sposobem działania funkcji i obiektów w JavaScripcie. Więcej na ten temat można przeczytać w dokumentacji biblioteki *Knockout.js*.

To wszystko, jeśli chodzi o pojedynczy wiersz tabeli. Gdy zmienimy liczbę produktów w zamówieniu, cena zostanie natychmiast zaktualizowana. Teraz rozbudujemy uzyskaną strukturę, aby utworzyć koszyk na wiele produktów, wyświetlający dodatkowo sumy cząstkowe i ogólną sumę należności.

Wiązania przepływu sterowania

Wiązanie obiektów z elementami HTML jest bardzo wygodne, ale w koszyku rzadko kiedy jest tylko jeden produkt, a duplikowanie całego tego kodu byłoby bardzo żmudne, nie mówiąc już o dodatkowych komplikacjach związanych z większą liczbą obiektów `LineItem`. Musimy coś zmienić.

Zamiast obiektu `LineItem` w roli modelu widoku, do reprezentowania koszyka użyjemy innego obiektu. Nadamy mu nazwę `Cart` i będziemy w nim przechowywać wszystkie obiekty `LineItem`. Wiedząc, jak działa metoda `dependentObservables()`,

możemy w obiekcie `Cart` utworzyć własność obliczającą sumę należności, gdy zmieni się którykolwiek z elementów koszyka.

A co z kodem HTML dla pojedynczego produktu? Duplikowania kodu możemy uniknąć dzięki użyciu tzw. **wiązania przepływu sterowania** (ang. *control flow binding*) i nakazując Knockout wyrenderowanie kodu HTML dla każdego produktu w koszyku.

Najpierw zdefiniujemy tablicę elementów, których użyjemy do napełnienia koszyka.

knockout/update_cart.html
```
var products = [
  {name: "Macbook Pro 15 inch", price: 1699.00},
  {name: "Przejściówka Mini Display Port na VGA", price: 29.00},
  {name: "Magic Trackpad", price: 69.00},
  {name: "Klawiatura bezprzewodowa Apple", price: 69.00}
];
```

W realnej aplikacji dane te pobieralibyśmy z usługi sieciowej lub wywołania Ajax albo generowalibyśmy je na serwerze podczas serwowania strony.

Teraz utworzymy obiekt `Cart` do przechowywania produktów. Zdefiniujemy go w taki sam sposób, jak obiekt `LineItem`.

knockout/update_cart.html
```
var Cart = function(items){
  this.items = ko.observableArray();

  for(var i in items){
    var item = new LineItem(items[i].name, items[i].price);
    this.items.push(item);
  }
}
```

Musimy zmienić wiązanie z `LineItem` na klasę `Cart`.

knockout/update_cart.html
```
var cartViewModel = new Cart(products);
ko.applyBindings(cartViewModel);
```

Produkty są zapisywane w koszyku za pomocą metody `observableArray()`, która działa tak samo jak `observable()`, ale ma właściwości tablicy. Gdy utworzyliśmy nowy egzemplarz naszego koszyka, przekazaliśmy do niego tablicę danych. Nasz obiekt iteruje po elementach danych i tworzy nowe egzemplarze `LineItem`, które są zapisywane w tablicy produktów. Ponieważ tablica ta jest obserwowalna, interfejs zmieni się po każdej zmianie zawartości tej tablicy. Oczywiście, teraz mamy więcej niż jeden produkt, a więc musimy zmodyfikować nasz interfejs.

Jaś pyta:
Jak wygląda sprawa dostępności w przypadku biblioteki Knockout?

Interfejsy w dużym stopniu oparte na JavaScripcie często bardzo słabo wypadają pod względem dostępności, jednak użycie tego języka samo w sobie o niczym jeszcze nie świadczy.

W tej recepturze użyliśmy ról i atrybutów HTML5 ARIA, aby pomóc czytnikom ekranu w zrozumieniu działania naszej aplikacji. Jednak kwestie dostępności dotyczą nie tylko czytników. W dostępności chodzi ogólnie o umożliwienie dostępu do treści jak najszerszemu gronu odbiorców.

Knockout to rozwiązanie napisane w JavaScripcie, a więc działa tylko wówczas, gdy obsługa tego języka jest włączona. Trzeba to brać pod uwagę. Najlepiej jest najpierw napisać aplikację, która jest użyteczna bez JavaScriptu, a następnie za pomocą biblioteki Knockout dodać różne *ulepszenia*. W naszym przykładzie zawartość koszyka jest renderowana za pomocą biblioteki Knockout, ale gdybyśmy użyli którejś z technologii serwerowych, moglibyśmy renderować kod koszyka i stosować wiązania Knockout do wyrenderowanego kodu HTML. Dostępność aplikacji zależy przede wszystkim od sposobu jej zaimplementowania, a nie od konkretnej użytej do jej budowy biblioteki.

Następnie zmodyfikujemy naszą stronę HTML i nakażemy Knockout utworzyć wiersz tabeli dla każdego produktu za pomocą wywołania `data-bind` na elemencie `<tbody>`.

knockout/update_cart.html

```
<tbody data-bind="foreach: items">
  <tr aria-live="polite">
    <td data-bind="text: name"></td>
    <td data-bind="text: price"></td>
    <td><input type="text" name="quantity" data-bind='value: quantity'></td>
    <td data-bind="text: subtotal "></td>
  </tr>
</tbody>
```

Nakazaliśmy bibliotece Knockout wyrenderować zawartość elementu `<tbody>` dla każdego elementu tablicy `items`. Nie musimy nic więcej zmieniać.

Teraz na naszej stronie może być wyświetlanych wiele wierszy tabeli i dla każdego z nich będzie wyświetlana suma cząstkowa. Zajmiemy się obliczaniem całkowitej kwoty do zapłaty i usuwaniem elementów.

Kwota do zapłaty

Sposób działania metody Knockout `dependentObservable()` poznaliśmy już przy okazji obliczania sumy cząstkowej dla każdego produktu. Dodając ją do obiektu `Cart`, możemy jej użyć także do obliczania sumy całkowitej.

```
this.total = ko.dependentObservable(function(){
  var total = 0;
  for (item in this.items()){
    total += this.items()[item].subtotal();
  }
  return total;
}, this);
```

Kod ten zostanie uruchomiony za każdym razem, gdy zmieni się którykolwiek z produktów. Aby wyświetlić sumę całkowitą, musimy oczywiście jeszcze dodać jeden wiersz do tabeli. Ponieważ ma to być suma całkowita należności za wszystkie produkty, wiersz ten umieścimy poza elementem `<tbody>`, w elemencie `<tfoot>` umieszczonym bezpośrednio pod zamykającym znacznikiem `</thead>`. Umieszczenie stopki **nad** treścią tabeli pomaga niektórym przeglądarkom i technologiom pomocniczym w szybszym rozpracowaniu struktury tabeli.

knockout/update_cart.html

```
<tfoot>
  <tr>
    <td colspan="4">Należność</td>
    <td aria-live="polite" data-bind="text: total()"></td>
  </tr>
</tfoot>
```

Gdy odświeżymy stronę i zmienimy liczbę przy którymś z produktów, nastąpi automatyczna aktualizacja sumy cząstkowej i całkowitej. Teraz przejdziemy do przycisku usuwania produktów.

Usuwanie produktów

Na zakończenie musimy jeszcze dodać przycisk *Usuń* obok każdego produktu, służący do jego usunięcia z koszyka. Dzięki całej wykonanej do tej pory pracy zadanie to jest już bardzo łatwe. Najpierw dodamy przycisk do tabeli.

```
<td>
  <button
    data-bind="click: function() { cartViewModel.remove(this) }">Usuń
  </button>
</td>
```

Tym razem zamiast wiązać dane z interfejsem, wiążemy zdarzenie i funkcję. Przekazujemy `this` do metody `remove()` wywoływanej na rzecz egzemplarza `cartViewModel`. Przycisk ten jednak nie działa, ponieważ jeszcze nie zdefiniowaliśmy metody `remove()`. Jej definicja znajduje się poniżej:

Żyj w zgodzie z serwerem!

Coraz większą popularność zdobywają koszyki na zakupy, których aktualizacja odbywa się w całości wyłącznie po stronie klienta. Czasami po prostu niemożliwe jest wysyłanie żądań Ajax za każdym razem, gdy użytkownik zmieni coś w interfejsie.

Stosując to podejście, musisz zadbać o synchronizację danych w koszyku na kliencie z danymi na serwerze. Przecież nie chciałbyś, aby ktoś zmieniał ceny produktów za Ciebie!

Gdy użytkownik przechodzi do kasy, należy przesłać zaktualizowane wartości na serwer i tam obliczyć sumy przed sfinalizowaniem transakcji.

knockout/update_cart.html

```
this.remove = function(item){ this.items.remove(item); }
```

To wszystko! Ponieważ tablica `items` jest obserwowalna (`observableArray`), aktualizowany jest cały interfejs, wraz sumą całkowitą!

Kontynuacja

Biblioteka Knockout jest doskonałym narzędziem do tworzenia dynamicznych jednostronicowych interfejsów, a dzięki temu, że nie jest związana z żadnym frameworkiem sieciowym, możemy jej używać, gdzie tylko chcemy.

Co ważniejsze, modele widoków używane w tej bibliotece są zwykłym kodem JavaScript, dzięki czemu można jej używać do implementowania wielu często potrzebnych funkcji interfejsu użytkownika. Na przykład przy użyciu Ajaksa z łatwością można by było utworzyć funkcję wyszukiwania bieżącego, zbudować kontrolki do edycji danych przesyłające dane na serwer w celu ich zachowania, a nawet aktualizować zawartość jednego menu rozwijanego na podstawie wartości wybranej w innym.

Zobacz również

- Receptura 14.: „Organizacja kodu przy użyciu biblioteki Backbone.js”

Receptura 14. Organizacja kodu przy użyciu biblioteki Backbone.js

Problem

W odpowiedzi na rosnące wymagania użytkowników w kwestii niezawodności i interaktywności aplikacji działających po stronie klienta, programiści ciągle tworzą nowe niesamowite biblioteki JavaScript. Jednak im bardziej skomplikowana jest dana aplikacja, tym bardziej jej kod wygląda jak pobojowisko pełne porozrzucanych bez ładu i składu rozmaitych bibliotek, wiązań zdarzeń, wywołań Ajax jQuery i funkcji przetwarzających dane w formacie JSON.

Potrzebna jest nam metoda tworzenia aplikacji działających po stronie klienta w taki sam sposób, jak od lat tworzymy aplikacje serwerowe. Krótko mówiąc, potrzebujemy jakiegoś frameworku. Mając solidny framework JavaScript, utrzymamy porządek w programie, zredukujemy powtarzalność kodu oraz zastosujemy standard zrozumiały także dla innych programistów.

Ponieważ Backbone to skomplikowana biblioteka, ta receptura jest znacznie dłuższa i bardziej skomplikowana od innych.

Składniki

- *Backbone.js*[15]
- *Underscore.js*[16]
- *JSON2.js*[17]
- Mustache[18]
- jQuery
- QEDServer

[15] *http://documentcloud.github.com/backbone*

[16] *http://documentcloud.github.com/underscore/*

[17] *https://github.com/douglascrockford/JSON-js*

[18] *http://mustache.github.com/*

Rozwiązanie

To zadanie możemy wykonać przy użyciu wielu różnych frameworków, ale Backbone.js jest jednym z najpopularniejszych, dzięki swojej elastyczności, niezawodności i ogólnie wysokiej jakości kodu. W chwili pisania tych słów był jeszcze względnie nowy, a miał już wielu użytkowników. Przy użyciu Backbone możemy wiązać zdarzenia w podobny sposób, jak to robiliśmy przy użyciu Knockout w recepturze 13., „Tworzenie interaktywnych interfejsów użytkownika przy użyciu biblioteki Knockout.js", ale teraz otrzymujemy modele współpracujące z serwerem oraz system trasowania żądań, za pomocą którego można śledzić zmiany w adresach URL. Dzięki Backbone otrzymujemy bardziej niezawodny zrąb aplikacji, który może doskonale sprawdzić się w przypadku skomplikowanych aplikacji serwerowych, ale stanowić przerost formy nad treścią w przypadku prostszych programów.

Użyjemy Backbone do poprawienia wrażliwości interfejsu naszego sklepu internetowego, tzn. sprawimy, że będzie żywiej reagował na działania użytkownika. Z naszych logów i badań zachowań użytkowników wynika, że odświeżanie strony trwa zbyt długo, a wiele rzeczy, które wykonuje się za pośrednictwem serwera, można by było zrobić na kliencie. Nasz kierownik zasugerował, abyśmy cały interfejs zarządzania produktami zmieścili na pojedynczej stronie, na której będzie można dodawać i usuwać produkty bez odświeżania strony.

Zanim rozpoczniemy budowę naszego interfejsu, bliżej poznamy Backbone i dowiemy się, jak za pomocą tej biblioteki możemy rozwiązać nasz problem.

Podstawy Backbone

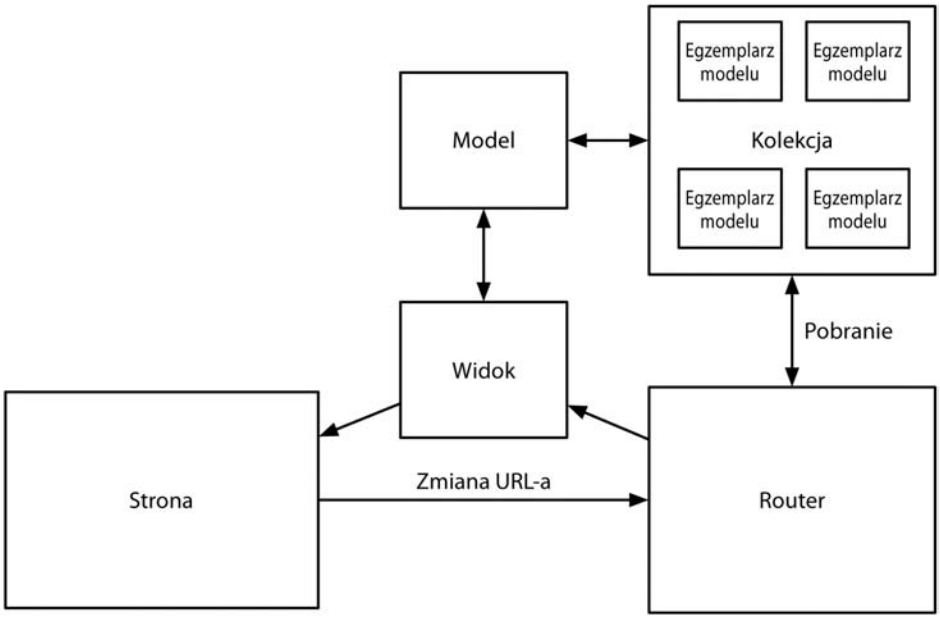
Backbone to działająca po stronie klienta implementacja wzorca model-widok-kontroler, na powstanie której duży wpływ miały serwerowe frameworki, takie jak ASP.NET MVC i Ruby on Rails. Backbone ma kilka komponentów pomagających dobrze zorganizować kod związany z komunikacją z serwerem.

Modele reprezentują dane i mogą współpracować z naszym zapleczem za pośrednictwem Ajaksa. Ponadto są doskonałym miejscem na wpisanie logiki biznesowej i kodu sprawdzającego poprawność danych.

Widoki w Backbone nieco różnią się od widoków w innych frameworkach. Są nie tyle warstwą prezentacji, co raczej „kontrolerami widoku". W interfejsie typowej aplikacji działającej po stronie klienta może być wiele zdarzeń. Kod wywoływany przez te zdarzenia jest przechowywany właśnie w tych widokach. Następnie mogą one renderować szablony i modyfikować nasz interfejs użytkownika.

Routery śledzą zmiany adresu URL i mogą wiązać ze sobą modele i widoki. Gdy chcemy pokazać w interfejsie różne „strony" lub karty, do obsługi żądań i w celu wyświetlania różnych widoków możemy użyć routerów. W Backbone obsługują one także przycisk *Wstecz* przeglądarki.

W końcu w Backbone dostępne są kolekcje, dzięki którym możemy łatwo pobierać zbiory egzemplarzy modeli, z którymi chcemy pracować. Na rysunku 2.10 pokazano, jak poszczególne elementy Backbone ze sobą współpracują oraz w jaki sposób użyjemy ich do budowy naszego interfejsu do zarządzania produktami.



Rysunek 2.10. Składniki Backbone

Domyślnie modele Backbone komunikują się z serwerową aplikacją RESTful przy użyciu metody `ajax()` z biblioteki jQuery i formatu JSON. Zaplecze musi akceptować żądania `GET`, `POST`, `PUT` i `DELETE` oraz rozpoznawać dane w formacie JSON w treści tych żądań. Są to jednak tylko ustawienia domyślne, które można zmodyfikować. W dokumentacji Backbone znajdują się informacje na temat tego, jak zmodyfikować kod działający po stronie klienta tak, aby współpracował z różnymi rodzajami zapleczy.

Nasze zaplecze będzie obsługiwać domyślne ustawienia, a więc będziemy mogli wywoływać niektóre metody na modelach Backbone, a framework będzie niepostrzeżenie serializował i deserializował informacje o naszych produktach.

I jeszcze jedna uwaga na koniec: jak napisaliśmy w ramce „Jak wygląda sprawa dostępności w przypadku biblioteki Knockout?", frameworków typu Backbone najlepiej jest używać **jako nakładek** na już istniejące strony internetowe, aby ulepszyć ich cechy użytkowe. Jeśli Twój kod działający po stronie klienta będzie zbudowany na solidnej podstawie, łatwiej będzie opracować rozwiązanie działające także bez JavaScriptu. W tej recepturze pracujemy z interfejsem, który już ma wersję działającą bez JavaScriptu.

Budowa interfejsu

Zbudujemy prosty, mieszczący się na jednej stronie interfejs do zarządzania produktami w naszym sklepie. Jego schemat przedstawiony jest na rysunku 2.11. Na górze strony będzie się znajdował formularz do dodawania produktów, a pod nim umieścimy listę produktów. Dane z magazynu produktów będziemy pobierać i modyfikować przy użyciu Backbone i jego interfejsu w stylu REST:

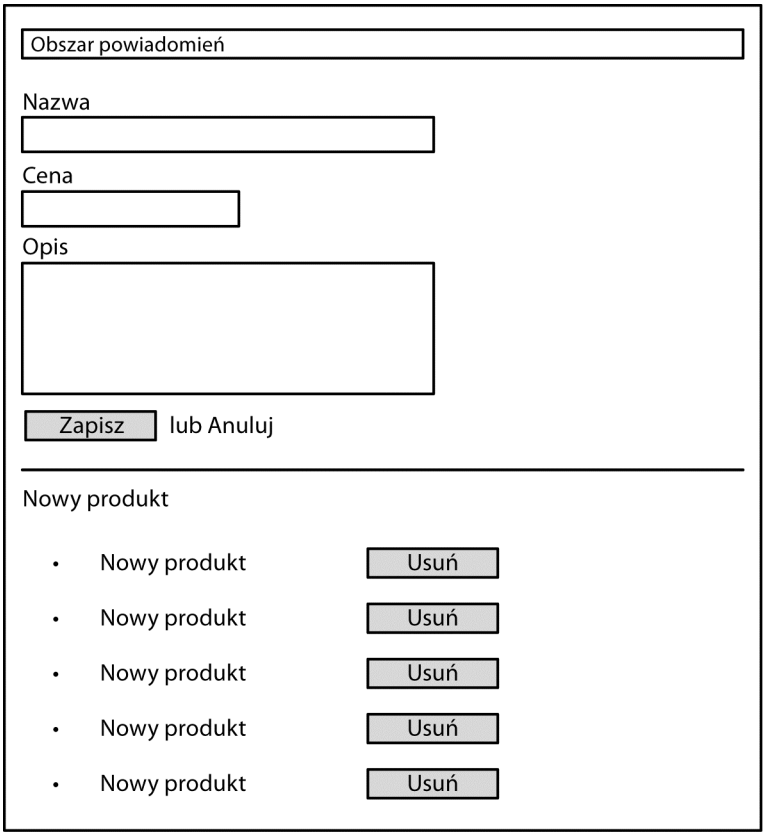
- Żądanie `GET()` do *http://przykład.com/products.json* pobiera listę produktów.
- Żądanie `GET` do */products/1.json* pobiera dane produktu o identyfikatorze `1` w formacie JSON.
- Żądanie `POST` do */products.json* z reprezentacją produktu w formacie JSON w treści tworzy nowy produkt.
- Żądanie `PUT` do *http://example.com/products/1.json* z reprezentacją produktu w formacie JSON w treści aktualizuje produkt o identyfikatorze `1`.
- Żądanie `DELETE` do */products/1.json* usuwa produkt o identyfikatorze `1`.

Ponieważ żądania Ajax muszą być wykonywane do tej samej domeny, do testowania użyjemy serwera QEDServer i jego API do zarządzania produktami. Wszystkie nasze pliki umieścimy w folderze *public* utworzonym przez serwer w przestrzeni roboczej.

Budowę interfejsu rozpoczniemy od utworzenia modelu produktu i kolekcji do przechowywania wielu modeli produktów. Żądania wyświetlenia listy produktów i formularza dodawania nowego produktu obsłużymy za pomocą routera. Ponadto utworzymy widoki listy produktów i formularza produktu.

Najpierw utworzymy folder *lib* na bibliotekę Backbone i jej zależności.

```
$ mkdir javascripts
$ mkdir javascripts/lib
```



Rysunek 2.11. Nasz interfejs do zarządzania produktami

Następnie pobierzemy Backbone.js wraz ze wszystkimi składnikami ze strony internetowej[19]. W tej recepturze używamy Backbone 0.5.3. Biblioteka ta wymaga obecności biblioteki *Underscore.js* zawierającej pewne funkcje wykorzystywane wewnętrznie przez Backbone, a dzięki którym my możemy zaoszczędzić na pisaniu kodu. Dodatkowo potrzebujemy biblioteki JSON2, która ma rozszerzone możliwości w zakresie obróbki danych w formacie JSON w przeglądarkach. Dodatkowo przyda nam się język szablonowy biblioteki Mustache[20]. Pobierz wszystkie te pliki i umieść je w folderze *javascripts/lib*.

Na koniec utworzymy plik o nazwie *app.js* w folderze *javascripts*. W pliku tym będą się znajdować wszystkie nasze składniki Backbone i nasz własny kod.

[19] *http://documentcloud.github.com/backbone/*

[20] Aby nie tracić czasu, wszystkie potrzebne pliki znajdziesz w kodzie źródłowym dołączonym do tej książki.

Mimo iż można by było utworzyć z tego dwa pliki, to jednak w ten sposób przy każdym wczytywaniu strony oszczędzamy jedno odwołanie do serwera.

Mając przygotowane wszystkie pliki, utworzymy bardzo prosty szkielet HTML w pliku *index.html*, w którym będą znajdowały się składniki naszego interfejsu użytkownika oraz będą dołączone pozostałe pliki. Zaczniemy od zdefiniowania typowych elementów strukturalnych oraz utworzymy elementy `<div>` na wiadomości dla użytkownika, formularz i listę produktów (`<ul>`).

backbone/public/index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Zarządzanie produktami </title>
  </head>
  <body role="application">
    <h1>Produkty</h1>
    <div aria-live="polite" id="notice">
    </div>
    <div aria-live="polite" id="form">
    </div>
    <p><a href="#new">Nowy produkt</a></p>

    <ul aria-live="polite" id="list">
    </ul>
  </body>
</html>
```

Te obszary będą aktualizowane bez odświeżania strony, w związku z czym dodaliśmy atrybuty ARIA HTML5, aby poinformować czytniki ekranu, jak mają obsługiwać te zdarzenia[21].

Pod tymi obszarami, a bezpośrednio *nad* zamykającym znacznikiem `</body>` umieścimy jQuery i Backbone z zależnościami oraz plik *app.js*:

Backbone/public/index.html

```
<script type="text/javascript"
        src="http://ajax.googleapis.com/ajax/libs/jquery/1.7/jquery.min.js">
</script>
<script type="text/javascript"
        src="javascripts/lib/json2.js"></script>
<script type="text/javascript"
        src="javascripts/lib/underscore-min.js"></script>
<script type="text/javascript"
        src="javascripts/lib/backbone-min.js"></script>
<script type="text/javascript"
        src="javascripts/lib/mustache.js"></script>
<script type="text/javascript"
        src="javascripts/app.js"></script>
```

[21] *http://www.w3.org/TR/html5-author/wai-aria.html*

Teraz możemy rozpocząć tworzenie listy produktów.

Tworzenie listy produktów

Aby utworzyć listę produktów, pobierzemy produkty z naszego zaplecza Ajax. Do tego celu potrzebne nam będą model i kolekcja. Model będzie reprezentował pojedynczy produkt, a kolekcja — grupę produktów. Tworząc i usuwając produkty, będziemy korzystać bezpośrednio z modelu. Natomiast pobierając listę produktów z serwera, możemy użyć kolekcji w celu pobrania rekordów i otrzymania grupy modeli Backbone, z którymi będziemy mogli pracować.

Najpierw utworzymy model. W pliku *javascripts/app.js* umieścimy następującą definicję obiektu `Product`:

backbone/public/javascripts/app.js
```
var Product = Backbone.Model.extend({

  defaults: {
    name: "",
    description: "",
    price: ""
  },
  url : function() {
    return(this.isNew() ? "/products.json" : "/products/" + this.id + ".json");
  }
});
```

Ustawiliśmy kilka domyślnych wartości dla przypadków, w których nie będzie żadnych danych, jak np. gdy tworzymy nowy egzemplarz. Następnie informujemy model, skąd ma pobierać swoje dane. Backbone używa do tego celu metody `url()` modelu, którą musimy wypełnić.

Mając zdefiniowany model, możemy utworzyć kolekcję, której użyjemy do pobrania wszystkich produktów dla naszej strony listy.

backbone/public/javascripts/app.js
```
var ProductsCollection = Backbone.Collection.extend({
  model: Product,
  url: '/products.json'
});
```

Kolekcja, podobnie jak model, ma metodę `url()`, którą musimy zaimplementować. Ponieważ jednak chcemy tylko pobrać listę wszystkich produktów, możemy po prostu na sztywno wpisać adres URL */products.json*.

Ponieważ z kolekcji tej będziemy korzystać w kilku miejscach naszej aplikacji, utworzymy jej egzemplarz. Obiekt ten zostanie utworzony na samym początku pliku *javascripts/app.js*.

backbone/public/javascripts/app.js
```
$(function(){
  window.products = new ProductsCollection();
```

Obiekt kolekcji wiążemy z obiektem `window`, dzięki czemu później będziemy mieli do niego łatwy dostęp z różnych widoków.

Mamy zdefiniowane model i kolekcję, a więc możemy przejść do widoku.

Szablon listy i widok

Widoki w Backbone zawierają logikę odpowiedzialną za zmienianie interfejsu w reakcji na zdarzenia. Do renderowania naszej listy produktów użyjemy dwóch widoków. Jeden z nich będzie reprezentował pojedynczy produkt przy użyciu szablonu Mustache i obsługiwał wszystkie zdarzenia związane z tym produktem. Natomiast drugi widok będzie iterował przez kolekcję produktów i renderował pierwszy widok dla każdego obiektu, umieszczając wyniki na naszej stronie. Dzięki temu będziemy mieli szczegółową kontrolę nad każdym składnikiem.

Najpierw utworzymy prosty szablon Mustache, którego nasze widoki Backbone będą używać do iterowania przez kolekcję produktów. Szablon ten dodamy do strony *index.html*, *nad* elementami `<script>` dołączającymi biblioteki:

backbone/public/index.html
```
<script type="text/html" id="product_template">
  <li>
    <h3>
      {{name}} - {{price}}
      <button class="delete">Usuń</button>
    </h3>
    <p>{{description}}</p>
  </li>
</script>
```

Wyświetlamy nazwę produktu, cenę i opis oraz przycisk do usuwania tego produktu.

Następnie utworzymy nowy widok, o nazwie `ProductView`, rozszerzając klasę Backbone `View` i definiując kilka kluczowych elementów.

backbone/public/javascripts/app.js
```
ProductView = Backbone.View.extend({
  template: $("#product_template"),
  initialize: function(){
    this.render();
  },
  render: function(){
  }
});
```

Najpierw za pomocą jQuery pobraliśmy ze strony indeksowej nasz szablon Mustache, posługując się jego identyfikatorem, i zapisaliśmy go we własności o nazwie `template`. Dzięki temu nie będziemy musieli pobierać tego szablonu ze strony za każdym razem, gdy chcemy wyrenderować jakiś produkt.

Następnie definiujemy funkcję `initialize()`, która będzie uruchamiana po utworzeniu nowego egzemplarza `ListView` i która będzie wywoływała funkcję `render()` widoku.

Każdy widok ma domyślną funkcję `render()`, którą trzeba przedefiniować, aby robiła to, co chcemy. W naszym przypadku będzie ona renderowała szablon Mustache pobierany ze zmiennej `template`. Ponieważ w zmiennej tej przechowywany jest obiekt jQuery, musimy wywołać metodę `html()`, aby pobrać zawartość szablonu z tego obiektu.

backbone/public/javascripts/app.js

```
render: function(){
  var html = Mustache.to_html(this.template.html(), this.model.toJSON() );
  $(this.el).html(html);
  return this;
}
```

W metodzie tej użyliśmy odwołania do modelu `this.model`, który będzie zawierał potrzebną nam listę produktów. Gdy utworzymy nowy egzemplarz widoku, możemy do niego przypisać model lub kolekcję, aby móc się do nich odwoływać w metodach widoku bez konieczności ich przekazywania, podobnie jak zrobiliśmy z szablonem Mustache. Wywołujemy funkcję `toJSON()` na naszym modelu, który przekazujemy do szablonu, aby dane tego modelu były łatwo dostępne w szablonie.

Metoda `render()` zapisuje kod HTML wyrenderowany z szablonu Mustache we własności widoku o nazwie `el` i zwraca ten egzemplarz widoku `ProductView`. Gdy wywołamy tę metodę, pobierzemy wyniki z tej własności i dodamy je do strony.

W tym celu utworzymy widok o nazwie `ListView` i strukturze bardzo podobnej do struktury widoku `ProductView`, tylko zamiast renderować szablon Mustache, będzie on iterować po kolekcji produktów i dla każdego z nich renderować widok `ProductView`.

backbone/public/javascripts/app.js

```
ListView = Backbone.View.extend({
  el: $("#list"),

  initialize: function() {
    this.render();
  },

  renderProduct: function(product){
```

```
    var productView = new ProductView({model: product});
    this.el.append(productView.render().el);
  },

  render: function() {
    if(this.collection.length > 0) {
      this.collection.each(this.renderProduct, this);
    } else {
      $("#notice").html("Brak produktów do wyświetlenia.");
    }
  }
});
```

Musimy aktualizować zawartość obszaru `list` na naszej stronie z listą produktów. Odwołanie do tego obszaru przechowujemy we własności o nazwie `el`. Dzięki temu mamy wygodny dostęp do tej listy w metodzie `render()`. W podobny sposób postąpiliśmy z szablonem Mustache w widoku `ProductView`.

Backbone korzysta z biblioteki *Underscore.js*, zawierającej funkcje ułatwiające pracę z kolekcjami. W metodzie `render()` iterujemy przez kolekcję przy użyciu metody `each()` i wywołujemy naszą metodę `renderProduct`. Metoda `each()` automatycznie przekazuje produkt. Jako drugi parametr przekazujemy `this`, co oznacza, że kontekstem dla metody `renderProduct()` ma być widok. Bez tego metoda `each()` szukałaby metody `renderProduct()` w kolekcji, co uniemożliwiłoby jej działanie.

Zdefiniowaliśmy model, kolekcję i dwa widoki oraz dodaliśmy szablon, ale wciąż nie mamy dla niego nic do wyświetlenia. Musimy to wszystko połączyć ze sobą podczas ładowania strony w przeglądarce. Do tego celu użyjemy routera.

Obsługa zmian adresów URL przy użyciu routerów

Podczas wczytywania strony będzie uruchamiany kod pobierający kolekcję produktów z API Ajax. Następnie kolekcja ta będzie przekazywana do nowego egzemplarza widoku `ListView` w celu wyświetlenia produktów. Routery Backbone umożliwiają wywoływanie funkcji w reakcji na zmiany adresu URL.

Utworzymy router o nazwie `ProductsRouter`. W tym celu rozszerzymy router Backbone i zdefiniujemy **trasę** mapującą część adresu URL znajdującą się za znakiem # na funkcję w naszym routerze. Dla domyślnego przypadku odpowiadającego sytuacji, gdy w adresie URL nie ma znaku #, zdefiniujemy pustą trasę, którą związemy z funkcją o nazwie `index()`. Ta domyślna trasa będzie uruchomiona przy wczytywaniu strony *index.html*.

backbone/public/javascripts/app.js

```
ProductsRouter = Backbone.Router.extend({
    routes: {
      "": "index"
    },
```

```
    index: function() {
    }
});
```

W akcji `index()` wywołujemy metodę `fetch()` naszej kolekcji produktów, aby pobrać dane z serwera.

backbone/public/javascripts/app.js

```
index: function() {
  window.products.fetch({
    success: function(){
      new ListView({ collection: window.products });
    },
    error: function(){
      $("#notice").html("Nie można załadować produktów.");
    }
  });
}
```

Metoda `fetch()` pobiera funkcje zwrotne `success` i `error`. Gdy wystąpi jakiś błąd, wyświetlamy komunikat o błędzie w obszarze `notice` strony. Gdy natomiast otrzymamy kolekcję danych, zostaje wywołana funkcja zwrotna `success()` i następuje utworzenie egzemplarza widoku. Ponieważ dzięki naszej metodzie widoku `initialize()` widok listy jest renderowany automatycznie, pozostaje nam jedynie utworzenie nowego egzemplarza routera, aby to wszystko uruchomić.

W pliku *javascripts/app.js* pod definicją `window.productsCollection` tworzymy egzemplarz routera. Następnie nakazujemy Backbone śledzenie zmian w adresie URL.

backbone/public/javascripts/app.js

```
window.products = new ProductsCollection();
$.ajaxSetup({ cache: false });
window.router = new ProductsRouter();
Backbone.history.start();
```

Do rozpoczęcia śledzenia zmian w adresie URL Backbone zmusza wiersz kodu `Backbone.history.start();`. Jeśli zapomnimy o nim, router nie będzie działał i na stronie nic się nie będzie działo.

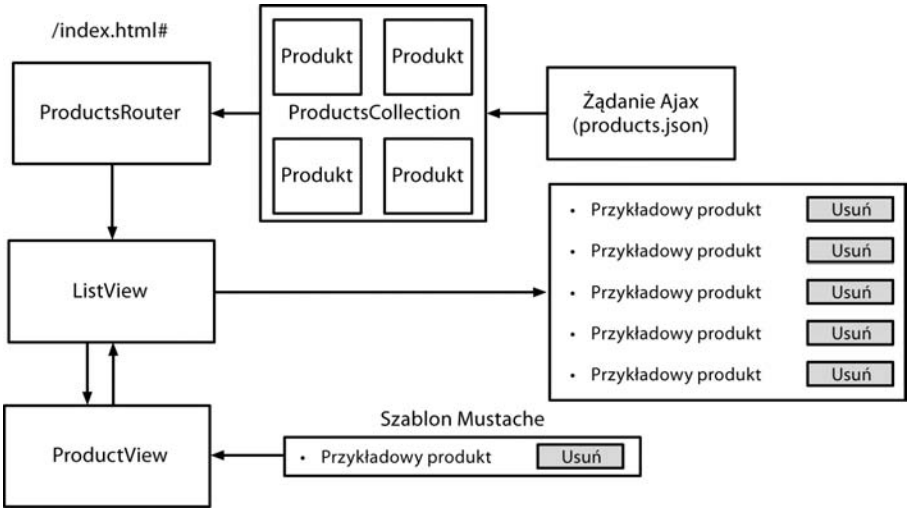
Poniższy wiersz wyłącza zapisywanie przez niektóre przeglądarki w pamięci podręcznej odpowiedzi Ajax z serwera:

```
$.ajaxSetup({ cache: false });
```

Gdy teraz wejdziemy na stronę *http://localhost:8080/index.html*, w końcu zobaczymy listę naszych produktów.

Podsumujmy, co do tej pory zrobiliśmy. Mamy router obserwujący adres URL i wywołujący metodę, która przy użyciu naszej kolekcji pobiera modele z naszej usługi sieciowej. Kolekcja ta jest następnie przekazywana do widoku, który

renderuje szablon i wysyła go do interfejsu użytkownika. Interakcje te są przedstawione w postaci schematu na rysunku 2.12. Może się wydawać, że to wszystko jest zbyt skomplikowane, jak na tak proste zadanie. Jednak w miarę ewolucji kodu pozwoli to zaoszczędzić ogromne ilości czasu. Stworzyliśmy podstawową infrastrukturę do dodawania, aktualizowania i usuwania produktów i nie będziemy musieli już się tym więcej przejmować. Teraz dodamy możliwość dodawania produktów.



Rysunek 2.12. Wyświetlanie listy produktów przy użyciu Backbone

Tworzenie nowego produktu

Mechanizm tworzenia nowych produktów będzie działał w ten sposób, że gdy użytkownik kliknie łącze *Nowy produkt*, na stronie pojawi się specjalny formularz. Gdy użytkownik wypełni ten formularz, pobierzemy z niego dane, przekażemy je do naszego zaplecza, a następnie wyświetlimy na liście.

Zaczniemy od dodania szablonu Mustache dla formularza na stronie *index.html*. Wstawimy go *pod* szablonem produktu, ale *nad* elementami `<script>` dołączającymi nasze biblioteki:

backbone/public/index.html

```
<script type="text/html" id="product_form_template">
  <form>
    <div class="row">
      <label>Nazwa<br>
        <input id="product_name" type="text" name="name"
               value="{{name}}">
      </label>
```

```
    </div>
    <div class="row">
      <label>Opis<br>
        <textarea id="product_description"
                  name="description">{{description}}</textarea>
      </label>
    </div>
    <div class="row">
      <label>Cena<br>
        <input id="product_price" type="text" name="price"
               value="{{price}}">
      </label>
    </div>
    <button>Zapisz</button>
  </form>
  <p><a id="cancel" href="#">Anuluj</a></p>
</script>
```

Znaczniki szablonu Mustache będą pobierać wartości z modelu i wstawiać je do pól formularza. Dlatego właśnie ustawiliśmy domyślne wartości w modelu Backbone. Szablonu tego moglibyśmy także użyć ponownie do edycji rekordów.

Teraz potrzebujemy widoku do renderowania tego szablonu z modelu. W pliku *javascripts/app.js* utworzymy nowy widok o nazwie `FormView`, podobny do tego, który utworzyliśmy dla naszej listy. Tym razem jednak zmienną `el` ustawimy na obszar `form` strony, a funkcja `render()` będzie pobierać szablon formularza i renderować wynik w tym obszarze.

backbone/public/javascripts/app.js

```
FormView = Backbone.View.extend({
  el: $("#form"),
  template: $("#product_form_template"),
  initialize: function(){
    this.render();
  },
  render: function(){
    var html = Mustache.to_html(this.template.html(), this.model.toJSON() );
    this.el.html(html);
  }
});
```

Gdy użytkownik kliknie łącze *Nowy produkt*, widok powinien renderować na stronie formularz. Ponieważ w wyniku tego następuje zmiana w adresie URL polegająca na dodaniu do niego części `#new`, możemy zdarzenie to obsłużyć za pomocą routera. Najpierw musimy w sekcji tras dodać trasę `#new`, która odpowiada miejscu wskazywanemu przez łącze *Nowy produkt*.

backbone/public/javascripts/app.js

```
routes: {
  "new": "newProduct",
  "": "index"
},
```

Następnie musimy zdefiniować funkcję pobierającą nowy model i przekazującą go do nowego egzemplarza widoku formularza, aby widok ten mógł zostać wyrenderowany na stronie. Metodę tę umieścimy nad metodą `index()`, a ponieważ deklaracje tych metod są zdefiniowane jako własności obiektu `this`, musimy je rozdzielić przecinkiem.

backbone/public/javascripts/app.js

```
newProduct: function() {
  new FormView( {model: new Product()});
},
```

Gdy odświeżymy stronę i klikniemy łącze *Nowy produkt*, zostanie wyświetlony nasz formularz. Dzięki mechanizmowi śledzenia historii Backbone, gdy naciśniemy przycisk *Wstecz* przeglądarki, adres URL ulegnie zmianie. Nie możemy jednak jeszcze zapisywać nowych rekordów. Zajmiemy się tym teraz.

Reagowanie na zdarzenia w widoku

Użyliśmy naszego routera do wyświetlenia formularza, ale routery reagują tylko na zmiany adresu URL. A musimy jeszcze dodać obsługę zdarzeń kliknięcia przycisków *Zapisz* i *Anuluj*. Zrobimy to w utworzonym wcześniej widoku formularza.

Zaczniemy od zdefiniowania zdarzeń dla widoku do obserwacji. Dodamy je do widoku przed funkcją `initialize()`:

backbone/public/javascripts/app.js

```
events: {
    "click .delete": "destroy"
},
events: {
    "click #cancel": "close",
    "submit form": "save",
},
```

Użyta tu składnia nieco różni się od typowej składni monitorowania zdarzeń JavaScriptu. Klucz tablicy definiuje obserwowane zdarzenie. Po nim znajduje się selektor CSS elementu, który ma być obserwowany. Natomiast wartość określa funkcję widoku, którą chcemy wywoływać. W tym przypadku obserwujemy zdarzenie kliknięcia dla przycisku anulowania i zdarzenie zatwierdzenia całego formularza.

Kod obsługujący łącze „zamykania” jest prosty — po prostu usuwamy treść elementu HTML zawierającego ten widok:

backbone/public/javascripts/app.js

```
close: function(){
  this.el.unbind();
  this.el.empty();
},
```

Metoda `save()` jest nieco bardziej skomplikowana. Najpierw wyłączamy zatwierdzanie formularza, a następnie pobieramy wartości wszystkich pól i umieszczamy je w nowej tablicy. Później ustawiamy atrybuty modelu i wywołujemy jego metodę `save()`.

backbone/public/javascripts/app.js

```
save: function(e){
  e.preventDefault();
  data = {
    name: $("#product_name").val(),
    description: $("#product_description").val(),
    price: $("#product_price").val()
  };
  var self = this;
  this.model.save(data, {
    success: function(model, resp) {
      $("#notice").html("Produkt został zapisany.");
      window.products.add(self.model);
      window.router.navigate("#");
      self.close();
    },
    error: function(model, resp){
      $("#notice").html("Błędy uniemożliwiły utworzenie produktu.");
    }
  });
},
```

Sposób użycia metody `save()` jest podobny do sposobu użycia metody `fetch()`, tzn. należy zdefiniować zarówno zachowanie w przypadku powodzenia, jak i niepowodzenia. Ponieważ funkcje tych zachowań działają w różnych kontekstach, tworzymy tymczasową zmienną o nazwie `self`, do której przypisujemy bieżący kontekst, aby móc się do niego odwołać w metodzie powodzenia. W odróżnieniu od metody `each()`, której używaliśmy do renderowania listy produktów, Backbone nie umożliwia przekazywania parametru kontekstu do funkcji zwrotnych[22].

Gdy zapisywanie danych powiedzie się, dodajemy nowy model do kolekcji i za pomocą routera zmieniamy adres URL. Nie powoduje to jednak uruchomienia odpowiedniej funkcji w routerze, przez co nie zobaczymy naszego produktu na liście. Można to jednak łatwo naprawić dzięki wiązaniom zdarzeń w Backbone.

Gdy dodajemy model do kolekcji, kolekcja ta wywołuje zdarzenie `add`, które możemy obserwować. Pamiętasz metodę `renderProduct()` z widoku listy? Możemy sprawić, aby metoda ta była wywoływana za każdym razem, gdy dodajemy model do naszej kolekcji. Wystarczy w tym celu dodać poniższy wiersz kodu do metody `initialize()` widoku `ListView`:

[22] Przynajmniej w czasie pisania tej książki.

backbone/public/javascripts/app.js
```
this.collection.bind("add", this.renderProduct, this);
```

Metoda `bind()` umożliwia dokonywanie wiązań ze zdarzeniami i jako argumenty przyjmuje nazwę zdarzenia, funkcję oraz kontekst. Jako trzeci argument przekazaliśmy `this`, co oznacza, że chcemy, aby kontekstem był *widok*, a nie kolekcja. Podobnie zrobiliśmy w metodzie `render()` widoku listy z `collection.each`.

Ponieważ do dodania rekordu użyliśmy istniejącej metody `renderProduct()`, nowy rekord został dodany na dole listy. Aby rekordy były dodawane na początku listy, możemy napisać nową metodę `addProduct()`, która używałaby metody `prepend()` jQuery. Pozostawiamy to jednak jako zadanie do samodzielnego wykonania.

Możemy już tworzyć nowe produkty i wyświetlać ich listę na stronie bez odświeżania. Czas dodać mechanizm usuwania produktów. Teraz właśnie skorzystamy z dobrej organizacji naszego kodu.

Usuwanie produktu

Aby usunąć produkt, wykorzystamy umiejętności zdobyte podczas pracy nad widokiem `FormView` oraz zaimplementujemy funkcję `destroy()` w widoku `ProductView`, która będzie wywoływana w wyniku naciśnięcia przycisku *Usuń*.

Najpierw zdefiniujemy zdarzenie do obserwacji kliknięć przycisków należących do klasy `delete`.

backbone/public/javascripts/app.js
```
events: {
  "click .delete": "destroy"
},
events: {
  "click #cancel": "close",
  "submit form": "save",
},
```

Następnie definiujemy metodę `destroy()`, którą to zdarzenie będzie wywoływać. W metodzie tej wywołamy metodę `destroy()` modelu związanego z tym widokiem. Zastosowanie ma w niej ta sama strategia powodzenia i błędu, której używaliśmy wcześniej. Skorzystamy też ze sztuczki ze słowem kluczowym `self`, aby obejść problemy z kontekstem, podobnie jak to zrobiliśmy przy zapisywaniu rekordów w widoku formularza.

backbone/public/javascripts/app.js
```
destroy: function(){
  var self = this;
  this.model.destroy({
    success: function(){
      self.remove();
    },
```

```
    error: function(){
      $("#notice").html("Wystąpił problem podczas usuwania produktu.");
    }
  });
},
```

Gdy model zostanie usunięty przez serwer, następuje wywołanie metody zwrotnej `success` wywołującej metodę `remove()`widoku, co powoduje zniknięcie rekordu z listy. Jeśli coś pójdzie nie tak, wyświetlamy odpowiednią wiadomość.

To wszystko! Mamy prosty, dobrze zorganizowany prototyp, którym możemy się już chwalić albo który możemy dalej rozwijać.

Kontynuacja

Opisana aplikacja jest dobra na początek, ale jest parę rzeczy, które można poprawić.

Po pierwsze, za pomocą jQuery aktualizujemy uwagę `notice` w kilku miejscach:

```
$("#notice").html("Produkt został zapisany.");
```

Można by było utworzyć funkcję opakowującą do oddzielenia tego od kodu HTML albo nawet użyć innego widoku Backbone i szablonu Mustache, aby wyświetlić te wiadomości.

Gdy zapisujemy rekordy, wartości z formularza pobieramy przy użyciu selektorów jQuery. Dane można by było umieszczać od razu w egzemplarzu modelu przy użyciu zdarzeń `onchange` pól formularza.

W tej recepturze przedstawiliśmy rozwiązania dotyczące dodawania i usuwania rekordów, ale można jeszcze dodać możliwość ich edycji. Możemy użyć routera do wyświetlania formularza, a nawet wykorzystać ten sam widok formularza, którego używaliśmy do tworzenia produktów.

Backbone to doskonały system usprawniający pracę z danymi zapleczowymi, ale to dopiero początek. Nie musisz używać Ajaksa. Równie dobrze dane możesz zapisywać na kliencie przy użyciu HTML5.

W celu lepszej integracji z aplikacjami serwerowymi Backbone obsługuje metodę `pushState()` obiektu `History` HTML5, dzięki czemu można używać prawdziwych adresów URL zamiast fragmentów ze znakiem #. Dodatkowo można opracować mechanizmy awaryjne, które będą serwować strony, gdy wyłączona jest obsługa JavaScriptu.

Dzięki wielu opcjom i doskonałej obsłudze Ajaksa Backbone jest niezwykle elastycznym frameworkiem, który najlepiej sprawdza się w sytuacjach, gdy potrzebna jest dobrze zorganizowana struktura kodu.

Zobacz również

- Receptura 10.: „Tworzenie szablonów HTML przy użyciu systemu Mustache”
- Receptura 13.: „Tworzenie interaktywnych interfejsów użytkownika przy użyciu biblioteki Knockout.js”

Skorowidz

E

F

L

Ł

M

Q

R

S

Ś

T

Z

Ż

Tworzenie nowoczesnych stron internetowych to cenna umiejętność. Niezależnie od tego, czy jesteś początkującym programistą WWW, czy masz wieloletnie doświadczenie w tej branży, poznanie gotowych rozwiązań zawartych w tej książce sprawi, że staniesz się wszechstronnym specjalistą.
Znajdziesz tu ponad 40 zwięźle opisanych, wypróbowanych rozwiązań problemów towarzyszących tworzeniu stron internetowych oraz poznasz nowe sposoby pracy, które pozwolą Ci jeszcze bardziej rozwinąć umiejętności. Możesz spodziewać się rzetelnego przeglądu wszystkich najnowszych technik programistycznych, od metod tworzenia atrakcyjnych wizualnie przycisków, przez analizę danych i testowanie kodu, po hosting witryn na serwerze.

Z tą książką nauczysz się projektować przykuwające wzrok przyciski za pomocą prostych stylów działających we wszystkich przeglądarkach, tworzyć animacje dla urządzeń przenośnych bez żadnych wtyczek, budować i testować wiadomości e-mail oraz konstruować elastyczne układy stron, odpowiednie dla komputerów stacjonarnych i przenośnych. Opis możliwości bibliotek jQuery, Knockout i Backbone.js pozwoli Ci na jeszcze efektywniejsze wykorzystanie możliwości współczesnych przeglądarek. Obecnie nie można ignorować rynku urządzeń mobilnych, dlatego dużo uwagi poświęcono tu dostosowaniu stron do wymogów urządzeń przenośnych. Książka ta jest obowiązkową pozycją w bibliotece każdego programisty WWW podążającego za trendami!

Poznaj przepisy na:

- tworzenie animacji z wykorzystaniem transformacji CSS3
- dzielenie treści na strony
- wykorzystanie bibliotek jQuery, Knockout.js, Backbone.js
- obsługę urządzeń mobilnych
- testowanie strony WWW

Pragmatic Bookshelf

Nr katalogowy: 13285
Księgarnia internetowa: http://helion.pl
Zamówienia telefoniczne: 0 801 339900
0 601 339900

helion.pl
księgarnia internetowa
Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Helion
Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

sięgnij po WIĘCEJ
KOD KORZYŚCI

Cena: 49,00 zł
ISBN 978-83-246-5149-8
9 788324 651498

Informatyka w najlepszym wydaniu