IDŹ DO

PRZYKŁADOWY ROZDZIAŁ

SPIS TREŚCI

KATALOG KSIĄŻEK

KATALOG ONLINE

ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK

DODAJ DO KOSZYKA

CENNIK I INFORMACJE

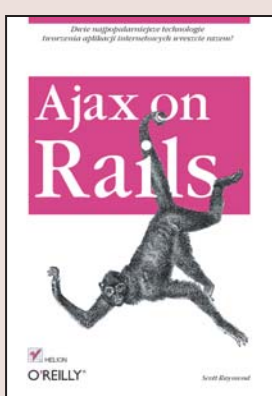
ZAMÓW INFORMACJE O NOWOŚCIACH

ZAMÓW CENNIK

CZYTELNIA

FRAGMENTY KSIĄŻEK ONLINE

Ajax on Rails



Autor: Scott Raymond
Tłumaczenie: Adrian Elczewski
ISBN: 978-83-246-1048-8
Tytuł oryginału: Ajax on Rails
Format: B5, stron: 336

Zobacz, jak wykorzystać potencjał technologii Ajax i Rails w tworzeniu zaawansowanych aplikacji internetowych!

- Jak używać platformy Rails do budowy dynamicznych aplikacji internetowych?
- Jak szybko tworzyć witryny ajaksowe, wykorzystując wydajne biblioteki?
- Jak zwiększyć komfort pracy użytkowników Twoich aplikacji internetowych?

Ajax to olbrzymie możliwości w zakresie tworzenia dynamicznych i interaktywnych aplikacji internetowych, działających niemal tak szybko, jak tradycyjne programy. Jednak lepsza jakość witryn wymaga zwykle pisania bardziej skomplikowanego kodu i, co za tym idzie, większych nakładów pracy i czasu. Tak też było do niedawna w przypadku Ajaksa, ale obecnie, gdy wzrosła popularność tej technologii, a ona sama dojrzała, programiści mogą korzystać z wielu bibliotek i platform, dzięki którym tworzenie efektownych aplikacji internetowych stało się niezwykle proste.

„Ajax on Rails" to podręcznik dla programistów, którzy chcą szybko i łatwo budować wydajne aplikacje internetowe na bazie dwóch popularnych mechanizmów – technologii Ajax oraz platformy Rails. Czytając go, dowiesz się, w jaki sposób Ajax umożliwia kreowanie funkcjonalnych i wygodnych w obsłudze witryn, a także nauczysz się błyskawicznie stosować tę technologię w oparciu o biblioteki Prototype i scipt.aculo.us oraz kompletną platformę do tworzenia aplikacji internetowych, czyli Rails. Poznasz też sposoby sprawnego diagnozowania aplikacji ajaksowych oraz zapewnisz im bezpieczeństwo i wydajność, aby udostępniać swym klientom produkty najwyższej klasy.

- Przegląd mechanizmów technologii Ajax
- Działanie platformy Rails
- Ajaksowe przesyłanie danych za pomocą biblioteki Prototype
- Dodawanie efektów do witryn przy użyciu biblioteki scipt.aculo.us
- Generowanie kodu JavaScript za pomocą szablonów RJS
- Zwiększanie użyteczności aplikacji
- Diagnozowanie aplikacji na platformie Rails
- Zapewnianie bezpieczeństwa programu
- Zwiększanie wydajności aplikacji

Wydawnictwo Helion
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

Przedmowa 7

1. Wprowadzenie 11
Dla kogo jest ta książka 11
Czym jest Ajax? 12
Czym jest Rails 18
„Twój Ajax w moim Rails" 21
Nabieranie prędkości 21
Podsumowanie 27

2. Pierwsze kroki 29
Staromodny sposób 29
Prototype oraz inne biblioteki JavaScript 33
Rails pojawia się na horyzoncie 35
Podsumowanie 40

3. Wprowadzenie do Prototype 41
Ustawianie sceny 41
Łącza w Ajaksie 44
Formularze 48
Formularze w Ajaksie 51
Przyciski 52
Obserwatory formularza 54
Podsumowanie 55

4. Wprowadzenie do script.aculo.us 57
Efekty wizualne 57
Przeciągnij i upuść 62
Podsumowanie 70

5. **RJS** 71
Instrukcje zamiast danych 71
Umieszczenie R w skrócie RJS 72
Przykłady z życia wzięte 85
Podsumowanie 87

6. **Użyteczność Ajaksa** 89
Zasady użyteczności 90
Kontekst Internetu 94
Użyteczność w Internecie 97
Programowanie uwzględniające różne przeglądarki 103
Podsumowanie 107

7. **Testowanie i usuwanie błędów** 109
Usuwanie błędów 110
Testowanie 122
Podsumowanie 134

8. **Bezpieczeństwo** 135
Zdrowy sceptycyzm: nie ufać danym wejściowym użytkownika 135
Hashowanie haseł 144
Uciszanie logów 145
Polityka tej samej domeny 146
Używanie i nadużywanie metod HTTP 148
Szyfrowanie i certyfikaty bezpieczeństwa 151
Lista mailingowa o bezpieczeństwie w Rails 152
Podsumowanie 152

9. **Wydajność** 153
Środowiska projektowe i produkcyjne 153
Przechowywanie sesji 154
Buforowanie wyjścia 155
Pakowanie zasobów 160
Postępowanie z długo działającymi zadaniami 162
Podsumowanie 164

10. **Informator o Prototype** 165
Wsparcie Ajaksa 166
Manipulacja DOM 172
Wbudowane rozszerzenia 185

11. **Informator o script.aculo.us** **199**
Efekty wizualne 199
Przeciągnij i upuść 209
Kontrolki 218
Rozszerzenia klasy element 226
Konstruktor DOM 228
Testowanie jednostkowe JavaScript 229
Metody narzędziowe 232

Przykład A Quiz **233**

Przykład B Galeria zdjęć **249**

Przykład C Aplikacja współpracy w grupie **267**

Skorowidz **315**

ROZDZIAŁ 2.

Pierwsze kroki

O, Ajaksie! Znowuż Cię przyzywam.

— Sofokles

W tym rozdziale głównym zamysłem jest zrobienie rundki, małymi kroczkami, po naprawdę prostych przykładach wykorzystania technologii Ajax. Rails dostarcza wiele możliwości tworzenia złożonych interakcji w technologii Ajax z użyciem bardzo małej ilości kodu. Ale żeby zrozumieć, co się dzieje „pod maską", każdy powinien być obeznany z najniższym poziomem działania technologii Ajax (np. obiektem `XMLHttpRequest`). Po przyswojeniu treści tej książki tworzenie obiektu `XMLHttpRequest` za pomocą biblioteki Prototype lub bez jej użycia nie będzie stanowiło problemu. Czytelnik będzie potrafił z pomocą Rails utworzyć proste interakcje w technologii Ajax bez pisania jakiegokolwiek kodu w JavaScripcie. Z tym założeniem zdobędziemy wiedzę na temat działania pomocników Rails oraz dowiemy się, jak wielu kłopotów one oszczędzają.

Dla czytelników, którzy mieli okazję zapoznać się z Rails i znają podstawy Ajaksa, ten rozdział będzie okazją do odświeżenia wiedzy, warto przynajmniej przyjrzeć się przykładom.

Staromodny sposób

Żeby rozpocząć, wykonajmy najprostszą rzecz do zrobienia z użyciem technologii Ajax: kliknijmy łącze i zaprezentujmy odpowiedź z serwera — używając bezpośrednio `XMLHttpRequest`, bez pomocy Prototype czy pomocników Rails dla JavaScript.

Używanie `XMLHttpRequest` jest często opisywane jako coś wyjątkowo trudnego. Łatwo zauważyć, że po zdobyciu odrobiny doświadczenia i poznaniu kilku nowych koncepcji nie jest to aż tak zawiłe, jak można by było się spodziewać na podstawie powszechnej opinii.

Rozpoczynanie projektu

Osoby, które nie stworzyły przykładu szkieletu Rails w poprzednim rozdziale, powinny zrobić to teraz, wpisując w wierszu poleceń systemowych:

```
rails ajaxonrails
cd ajaxonrails
script/server
```

Za pomocą przeglądarki należy otworzyć stronę *http://localhost:3000/* — powinien się pojawić ekran powitalny Rails (dla celów przyszłego projektowania warto pamiętać, że `script/server` uruchamia na porcie 3000 serwer HTTP). Teraz utwórzmy nowy kontroler, który nazwiemy `Chapter2Controller`, z akcją `myaction`. (Po uruchomieniu serwera w jednym terminalu warto otworzyć inny).

```
script/generate controller chapter2 myaction
```

Generator Rails jest używany do uzupełniania szkieletu — przeważnie przez tworzenie nowych kontrolerów i modeli. Oczywiście można by w prosty sposób utworzyć kontroler plików ręcznie, ale używanie generatora jest oszczędnością pisania — co zapobiega robieniu błędów.

Generator ma także inny skutek: za każdym razem, gdy generuje się kontroler, tworzony jest również współpracujący z nim plik testów funkcjonalnych. To sposób Rails na przypominanie, że testowanie jest ważną częścią tworzenia aplikacji. Aby dowiedzieć się więcej o dostępnych generatorach i ich opcjach, należy uruchomić `script/generate` bez żadnych argumentów.

Teraz trzeba przejść do *http://localhost:3000/chapter2/myaction*. Należy się spodziewać nowo utworzonego widoku jak na rysunku 2.1.

Rysunek 2.1. Nowo utworzony kontroler Rails i jego widok

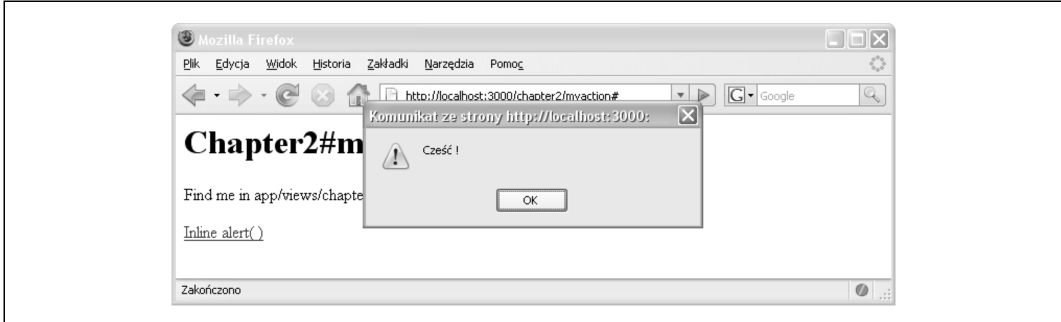
Proszę zauważyć, że domyślnie pierwsza część adresu URL determinuje kontroler, a druga akcję — metodę w ramach kontrolera. Teraz będziemy edytować szablon dla tej akcji, do którego prowadzi ścieżka *app/views/chapter2/myaction.rhtml*. Dodajemy ten fragment HTML na dole pliku.

```
<p><a href="#" onclick="alert('Cześć !');">Inline alert( )</a></p>
```

Jak można zauważyć, tworzymy akapit z prostym łączem — ale zamiast standardowego atrybutu `href` używamy `onclick`, do którego dostarczamy fragment kodu JavaScript do uruchomienia. Po odświeżeniu przeglądarki i kliknięciu łącza pojawi się to, co przedstawia rysunek 2.2.

Więcej niż jedna czy dwie instrukcje wstawione do atrybutu `onclick` mogłyby szybko stać się niewygodne. Przenieśmy kod do osobnej funkcji JavaScript poprzez dodanie tego, co znajduje się poniżej:

```
<p><a href="#" onclick="customAlert( ); ">Wywołanie własnej funkcji</a></p>
<script type="text/javascript">
  function customAlert( ) {
    alert('Powitanie z własnej funkcji.');
  }
</script>
```



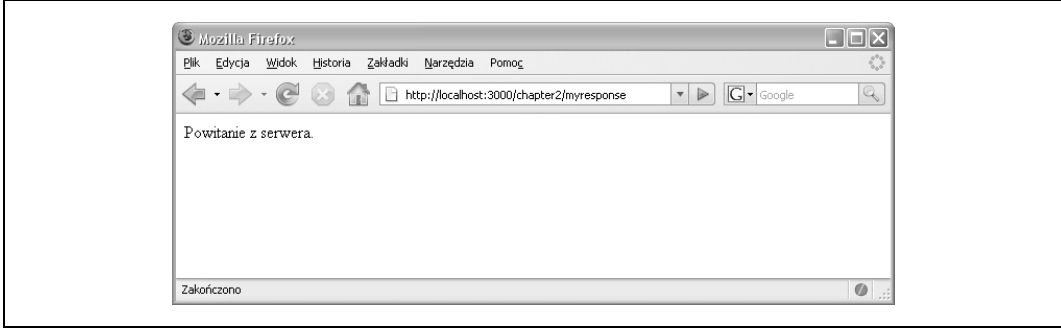
Rysunek 2.2. Prosta ramka ostrzegawcza

Proszę spróbować ponownie odświeżyć stronę i zobaczyć, co się stanie. Rezultat powinien być w zasadzie taki sam jak poprzednio.

Koniec rozgrzewki, teraz zajmiemy się Ajaksem. (Ale proszę pamiętać, że wciąż zaglądamy „pod maskę" — pod koniec tego rozdziału sporo złożoności Rails znacznie się uprości). Po pierwsze, musimy zdefiniować nową akcję w kontrolerze, *app/controllers/chapter2_controller.rb*. Teraz znajduje się tam akcja `myaction`, więc następną nazwijmy `myresponse`. Aby to zrobić, należy utworzyć nowy plik, *myresponse.rhtml* w katalogu *app/views/chapter2*. Do zawartości pliku wprowadźmy:

```
Powitanie z serwera.
```

Żeby mieć pewność, że wszystko działa, proszę odwiedzić tę akcję w swojej przeglądarce pod adresem *http://localhost:3000/chapter2/myresponse* — będzie widoczne to, co przedstawia rysunek 2.3.

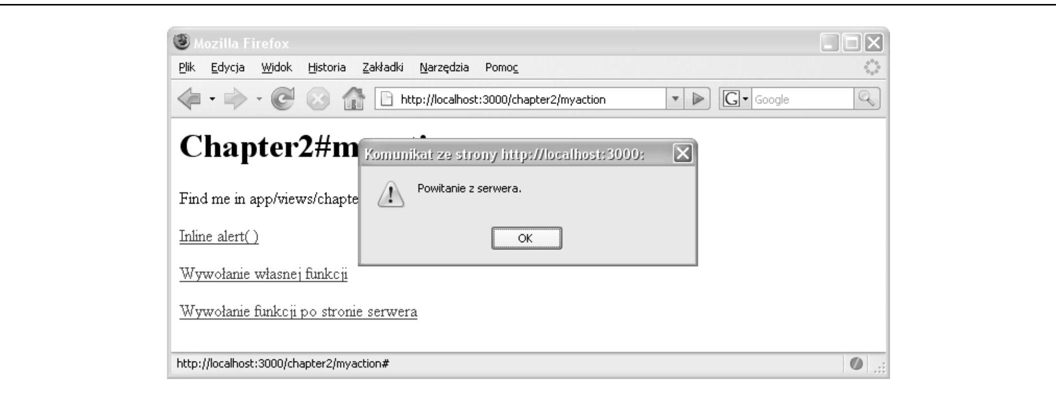


Rysunek 2.3. Wynik akcji myresponse

Teraz wróćmy do *myaction.rhtml* i dodajmy kolejny fragment kodu HTML i JavaScript.

```
<p><a href="#" onclick="serverSideAlert( );">Wywołanie funkcji po stronie serwera
</a></p>
<script type="text/javascript">
  function serverSideAlert( ) {
    var request = new XMLHttpRequest( );
    request.open('get', '/chapter2/myresponse', false);
    request.send(null);
    alert(request.responseText);
  }
</script>
```

Za pomocą przeglądarki przejdźmy z powrotem do *http://localhost:3000/chapter2/myaction* i kliknijmy nowe łącze. Jeśli wszystko poszło dobrze, powinna się pojawić wiadomość z serwera, taka jak na rysunku 2.4. Ostrzegamy, że ten przykład *nie będzie* działał we wcześniejszych niż siódma wersjach Internet Explorera (ten problem podejmiemy później).



Rysunek 2.4. Rezultat pierwszego wywołania w Ajaksie

Teraz do czegoś doszliśmy! Żeby się przekonać, warto zerknąć na terminal, gdzie uruchomiony jest `script/server`. Za każdym razem, gdy klika się „zajaksowane" łącze, rejestrowane będzie nowe kliknięcie:

```
Processing Chapter2Controller#myresponse [GET]
  Parameters: {"action"=>"myresponse", "controller"=>"chapter2"}
Completed in 0.00360 (278 reqs/sec) | Rendering: 0.00027 (7%) |
  200 OK [http://localhost/chapter2/myresponse]
```

Dużym problemem omawianego przykładu jest to, że nie działa on w jednej z najbardziej rozpowszechnionych przeglądarek, Internet Explorer 6. Przyczyną jest obiekt ActiveX w implementacji `XMLHttpRequest` Microsoftu (a właściwie dwa takie obiekty, co zależy od wersji IE), który musi być tworzony w inny sposób. Żeby zlikwidować ten problem i sprawić, aby nasz przykład działał poprawnie we wszystkich przeglądarkach, tworzymy małą funkcję. Oto wersja przyjazna dla IE:

```
<p><a href="#" onclick="IEAlert( );">Wywołanie serwera (działające pod IE)</a></p>
<script type="text/javascript">
  function IEAlert( ) {
    function getRequestObject( ) {
      try { return new XMLHttpRequest( ) } catch (e) {}
      try { return new ActiveXObject("Msxml2.XMLHTTP") } catch (e) {}
      try { return new ActiveXObject("Microsoft.XMLHTTP") } catch (e) {}
      return false
    }
    var request = getRequestObject( );
    request.open('get', '/chapter2/myresponse', false);
    request.send(null);
    alert(request.responseText);
  }
</script>
```

Ta wersja jest taka sama jak wcześniejsza, z wyjątkiem tego że zamiast tworzyć bezpośrednio obiekt `XMLHttpRequest`, wywoływana jest funkcja `getRequestObject()`, która wybiera możliwą opcję. Ta funkcja robi użytek z deklaracji `try` w JavaScripcie, która jest wykorzystywana

do wyłapywania wyjątków i tłumienia ich. (Ten przykład wprowadza także ideę deklarowania funkcji w funkcji, która może być nowością dla niektórych programistów).

Dotychczas odrobinę oszukiwaliśmy, ponieważ wywołanie Ajaksa nie jest asynchroniczne. Decyduje o tym trzeci parametr w metodzie `request.open()`. Do tej pory zakładaliśmy, że wywołanie nie było synchroniczne. W związku z tym `request.send()` było *blokujące* — interpreter JavaScript zatrzymywał wykonywanie w tym wierszu i nie przechodził do następnego, dopóki nie nadeszła odpowiedź z serwera. Żeby sprawić, aby wywołanie było asynchroniczne, musimy trochę zmienić kod. Proszę dodać ten fragment kodu do *myaction.rhtml*:

```
<p><a href="#" onclick="asyncAlert( )">Asynchroniczne wywołanie serwera</a></p>
<script type="text/javascript">
  function asyncAlert( ) {
    function getRequestObject( ) {
      try { return new XMLHttpRequest( ) } catch (e) {}
      try { return new ActiveXObject("Msxml2.XMLHTTP") } catch (e) {}
      try { return new ActiveXObject("Microsoft.XMLHTTP") } catch (e) {}
      return false
    }
    var request = getRequestObject( );
    request.open('get', '/chapter2/myresponse');
    request.onreadystatechange = function( ) {
      if(request.readyState == 4) alert(request.responseText);
    }
    request.send(null);
  }
</script>
```

We wszystkich poprzednich przykładach wywoływaliśmy `request.send()` i natychmiast potem odwoływaliśmy się do `request.responseText()`. Teraz, gdy wysyłamy asynchroniczne żądanie, odpowiedź niekoniecznie wraca po zakończeniu wywołania. Aby rozwiązać ten problem, obiekt `XMLHttpRequest` ma atrybut `readyState`, który zmienia się w czasie cyklu życia żądania. Ma także atrybut `onreadystatechange`, gdzie można zdefiniować funkcję, która będzie wywoływana za każdym razem, gdy status `readyState` będzie się zmieniał. W tym przykładzie definiujemy funkcję, która sprawdza, czy `readyState` jest równy `4` (co oznacza, że żądanie się zakończyło; kody `readyState` opisane są w pełni w rozdziale 3.), a jeśli tak, wyświetla okienko z komunikatem. Opanowanie asynchronicznych zdarzeń może wymagać trochę czasu, ale jest zasadniczą częścią ręcznego programowania w Ajaksie.

Prototype oraz inne biblioteki JavaScript

Osoby, które dopiero zaczynają swoją przygodę z Ajaksem, prawdopodobnie zaczęły zauważać, że pisanie w czystym Ajaksie, pozbawionym wsparcia dodatkowych bibliotek albo metod pomocniczych, nie jest powszechne. W ogóle pomysł pisania więcej niż tuzina wierszy kodu w celu stworzenia najprostszego możliwego zadania jest odpychający.

Dziesiątki bibliotek JavaScript wychodzą z siebie, żeby sprawić, by Ajax był łatwiejszy w obsłudze. Jedną z najbardziej popularnych jest Prototype, która stanowi część Rails. Będziemy omawiać Prototype gruntownie w rozdziale 10., ale teraz przyjrzyjmy się pewnym przykładom. Zanim zaczniemy coś innego, przeróbmy ponownie ostatni przykład, tym razem używając Prototype. Oto nowy fragment do dodania:

```
<script src="/javascripts/prototype.js" type="text/javascript">
</script>
<p><a href="#" onclick="prototypeAlert( );">Wywołanie funkcji z Prototype</a></p>
```

```
<script type="text/javascript">
 function prototypeAlert( ) {
  new Ajax.Request('/chapter2/myresponse', { onSuccess: function(request) {
   alert(request.responseText);
  }})
 }
</script>
```

Proszę zwrócić uwagę na pierwszy wiersz, gdzie włączamy ładowanie źródła pliku *prototype.js*, by móc z niego korzystać na naszej stronie. Przy pierwszym tworzeniu szkieletu aplikacji Rails kopia Prototype była umieszczona w katalogu *public/javascripts*. Wewnątrz funkcji `prototype-Alert()` pierwszy wiersz tworzy nową instancję `Ajax.Request`, jednej z klas Prototype. Pierwszy wywoływany argument jest adresem URL, drugi — jest obiektem JavaScript — kolekcją par kluczy – wartości, które zachowują się podobnie do map albo tablic asocjacyjnych w innych językach programowania. W tym przypadku jedyną wartością jest `onSuccess` określająca funkcję wywoływaną jako funkcja zwrotna.

Proszę zwrócić uwagę, iż w tym przykładzie nie ma żadnego kodu specyficznego dla obsługi wersji `XMLHttpRequest` dla przeglądarki IE i żadnej obsługi kodów `readyState`. Prototype obsługuje te szczegóły, udostępniając programiście dużo czystsze API.

Dotychczas wszystkie nasze przykłady tworzyły okno komunikatu `alert()` — które, w rzeczywistych aplikacjach, prawdopodobnie nie jest najczęściej używane. Znacznie częściej dodawana jest nowa zawartość strony albo modyfikowana dotychczasowa. Oto nowy fragment do dodania:

```
<p><a href="#" onclick="updateElement( )">Uaktualnij element </a></p>
<p id="response"></p>
<script type="text/javascript">
  function updateElement( ) {
    new Ajax.Request('/chapter2/myresponse', { onSuccess: function(request) {
      $('response').update(request.responseText);
    }})
  }
</script>
```

Proszę zauważyć różnice między powyższym a wcześniejszym przykładem: dodany został nowy pusty element akapitu z atrybutem `id="response"`, który będzie przechowywał odpowiedź otrzymaną z serwera. Funkcja `onSuccess` została zmieniona, zamiast wywołania `alert()` funkcja ta umieszcza tekst odpowiedzi w elemencie `response` (używając metody `update()` z biblioteki Prototype, która ustawia właściwość elementu `innerHTML`). Symbol dolara jest faktycznie nazwą funkcji definiowanej przez Prototype, która pobiera ciąg znaków i zwraca element HTML na podstawie tego ID. Ponieważ aktualizacja elementów HTML będzie bardzo często wykonywanym zadaniem, Prototype ułatwia to poprzez `Ajax.Updater`. Proszę to sprawdzić:

```
<p><a href="#" onclick="updater( )">Modernizuj za pomocą Ajax.Updater</a></p>
<p id="response2"></p>
<script type="text/javascript">
  function updater( ) {
    new Ajax.Updater('response2', '/chapter2/myresponse');
  }
</script>
```

Funkcja `$()` w Prototype będzie używana bardzo często, z bliska wygląda niezwykle wartościowo. Na pierwszy rzut oka jest prostym opakowaniem dla standardowej metody DOM `document.getElementById` z nazwą dużo prostszą do zapamiętania i sprawiającym wrażenie składni JavaScript. Ale to więcej niż tylko opakowanie.

Przede wszystkim może przyjąć dowolną liczbę argumentów, więc można otrzymać kilka elementów jednocześnie. Ponadto każdy zwracany element jest automatycznie rozszerzany o potężny zestaw metod omówionych w rozdziale 10.

Prawdopodobnie najbardziej istotne jest, że jeśli przekaże się do metody `$()` ciąg znaków, zwróci ona element DOM z tym właśnie ID. Ale jeśli przekaże się obiekt jakiegokolwiek innego typu — powiedzmy element DOM — w prosty sposób zwróci ten obiekt bez zmian. Wynikiem jest to, że można używać `$()` z wartościami, nawet jeśli nie jest się pewnym, czy wartości te są ciągiem znaków czy elementem DOM, co sprawia, że API JavaScript jest mniej podatne na błędy.

Proszę zwrócić uwagę, że ten przykład nie ma w sobie funkcji `onSuccess`, tutaj `Ajax.Updater` pobiera tylko dwa argumenty: ID elementu HTML, który ma być zaktualizowany, i URL żądania. `Ajax.Updater` wywołuje URL i automatycznie tworzy funkcję `onComplete` służącą do zaktualizowania określonego elementu DOM za pomocą wartości `response.Text`. Tak jak w przypadku `Ajax.Request`, ostatni argument jest zestawem opcji. Jedna z nich jest nazwana `insertion`. Pozwala na pójście dużo dalej niż prosta zamiana zawartości elementu, zamiast tego umożliwia wstawienie zawartości w rozmaitych punktach. Istnieją cztery typy wstawiania: `Before`, `Top`, `Bottom` oraz `After`. Na przykład:

```
<p><a href="#" onclick="appendToElement( )">Dodaj do elementu</a></p>
<p id="response3"></p>
<script type="text/javascript">
  function appendToElement( ) {
    new Ajax.Updater('response3', '/chapter2/myresponse',
      { insertion:Insertion.Bottom });
  }
</script>
```

Kiedy kliknie się łącze za pierwszym razem, odpowiedź z serwera będzie dodana do tej strony tak jak poprzednio. Przy późniejszych kliknięciach, zamiast zastąpić wcześniejszą zawartość, kolejne odpowiedzi będą dołączane do poprzednich.

Proszę zauważyć, że zdołaliśmy zredukować dość złożone zachowanie do postaci funkcji z zaledwie jedną instrukcją. Aby zatoczyć pełne koło, możemy zredukować kod do postaci pojedynczego atrybutu `onclick`:

```
<p><a href="#" onclick="new Ajax.Updater('response4',
'/chapter2/myresponse', { insertion:Insertion.Bottom });">
Dodaj do elementu</a></p>
<p id="response4"></p>
```

Jak będzie można się wkrótce przekonać, jest to dokładnie ten sam kod, który generują pomocniki JavaScript w Rails.

Rails pojawia się na horyzoncie

Rails dostarcza dogodną integrację z Prototype w formie metod pomocników, które generują wywołania funkcji udostępnianych przez Prototype. Odkryjemy, jak tworzyć Ajaksa bez pisania jakiegokolwiek kodu w JavaScripcie, używając metody pomocnika `link_to_remote()`.

Po pierwsze, musimy cofnąć się odrobinę i dowiedzieć się, jak Rails obsługuje widoki.

Podstawy ERb

Osoby, które kiedykolwiek korzystały z PHP, ColdFusion, ASP, JSP albo czegoś podobnego, uznają, że jest to znajoma koncepcja. Wbudowany Ruby (Erb, ang. *Embedded Ruby*) pozwala na łączenie fragmentów Ruby z HTML-em. ERb definiuje zestaw specjalnych znaczników, które są interpretowane jako Ruby; wszystko inne jest traktowane jako czysty HTML i zwracane w nienaruszonej postaci. Oto te specjalne znaczniki:

- <%= %> — Najczęściej używany, zawiera *wyrażenie* Ruby — którego wynik zwracany jest w miejscu znacznika.
- <%= -%> — Działa tak jak powyższy, ale usuwa znaki nowego wiersza znajdujące się za tym znacznikiem, co pozwala na czystsze zorganizowanie plików szablonów bez zbędnych pustych miejsc w wynikowych dokumentach HTML.
- <% %> — Przechowuje fragment kodu Ruby, ale nie zwraca niczego.
- <% -%> — Działa tak jak powyższy, ale usuwa znaki nowego wiersza znajdujące się za tym znacznikiem.
- <%# %> — To jest komentarz Ruby, który jest ignorowany i niczego nie zwraca.

Teraz spójrzmy na przykład.

Czy pamiętasz dyskusję o MVC z rozdziału 1.? Tutaj MVC zaczyna odgrywać swoją rolę. Zwykle kontroler będzie otrzymywać żądanie wyświetlenia strony i przygotowywać dane potrzebne dla widoku. W Rails dane te są umieszczane w *zmiennych instancji* (które są rozpoznawane dzięki brzydkiemu znakowi @, od którego się zaczynają ich nazwy). Proszę sobie zatem wyobrazić, że mamy taką akcję kontrolera:

```
def myaction
  @foo = "Witaj, świecie!"
end
```

Akcja definiuje zmienną nazwaną @foo i przypisuje jej łańcuch znaków Witaj, świecie!. Nasz szablon mógłby więc zawierać coś takiego:

```
<%= @foo %>
```

I, gdy szablon jest wywoływany, <%= @foo %> będzie zastąpione przez Witaj, świecie!. Całkiem oczywista sprawa. W praktyce przeważnie chce się wykorzystać zmienną w strukturze HTML, np.:

```
<h1><%= @foo %></h1>
```

Ponieważ znacznik <% %> nie produkuje żadnego wyjścia, najczęstsze jego użycie związane jest ze strukturami kontrolnymi, takimi jak instrukcja if i iteracje each. W odróżnieniu od innych systemów szablonowych nie istnieje składnia specyficzna dla ERb dla tych konstrukcji; ERb używa zwyczajnych wyrażeń języka Ruby. Kilka przykładów:

```
<% if @page_title %><h1><%= @page_title %></h1><% end %>
<% unless @names.empty? %>
  <ul>
    <% @names.each do |name| %><li><%= name %></li><% end %>
  </ul>
<% end %>
```

Proszę spojrzeć na drugi wiersz. Zaczyna się od wyrażenia warunkowego unless — odpowiednika Ruby dla if not. Proszę zwrócić też uwagę na @names.empty?. Wszystkie tablice Ruby korzystają z metody nazwanej empty? — zazwyczaj nazwy metod Ruby zwracających prawdę lub fałsz kończą się znakiem zapytania. Ostatnią sprawą wartą podkreślenia jest czwarty wiersz. Wywołanie metody each dla @names iteruje przez każdy element tablicy, zatem kod ten przejdzie całą tablicę @names i zwróci listę elementów w HTML dla każdego imienia.

Układ graficzny

Układ graficzny tworzą specjalne szablony, które przechowują powszechnie używane znaczniki dla wielokrotnie wykorzystywanych widoków. W innych systemach szablonowych jest to często osiągane poprzez tworzenie plików z szablonami nagłówka i stopki, które są włączane do szablonu strony. Rails działa odwrotnie — nagłówki i stopki są zdefiniowane w jednym pliku wystroju graficznego, a stamtąd dołączana jest treść strony. Pliki układu graficznego są przechowywane w *app/views/layouts* i domyślnie Rails najpierw poszuka tego, którego nazwa jest taka sama jak aktualnego kontrolera, np. *chapter2.rhtml*. Jeśli Rails takiego pliku układu graficznego nie znajdzie, poszuka pliku nazwanego *application.rhtml*. Zawartość pliku wystroju graficznego może wyglądać następująco:

```
<html>
  <head>
    <title>Moja Aplikacja Rails </title>
    <%= javascript_include_tag "prototype" %>
  </head>
  <body>
    <%= yield %>
  </body>
</html>
```

Najważniejszą częścią, o której należy wspomnieć, jest `<%= yield %>`. Jej zadaniem jest dołączenie kodu z szablonu widoku. Innymi słowy, spowoduje wstawienie kodu szablonu widoku do pliku układu graficznego. Proszę nie zapominać o dołączeniu tego wywołania w pliku układu graficznego, bo w przeciwnym razie strony mogą się wydawać puste.

Części

Części są podszablonami zaprojektowanymi dla fragmentów złożonych ze znaczników, które wykorzystywane są ponownie — albo np. chce się je trzymać w osobnym pliku, żeby pliki szablonów pozostały przejrzyste. Części są łatwe do zidentyfikowania, ponieważ ich nazwy zawsze zaczynają się od znaku podkreślenia. Na przykład, można stworzyć plik *app/views/chapter2/_person.rhtml* zawierający:

```
<p><%= person.name %></p>
```

Z głównego szablonu można by było załączyć taką część:

```
<%= render :partial => "person" %>
```

Jest trochę magii wplecionej w przekazywanie zmiennych do części. Ponieważ ta część jest nazwana „person”, główny szablon będzie szukał *zmiennej instancji* `@person` i przekazywał ją do części jako zmienną *lokalną* `person`. Co jeśli przykładowa zmienna nie pasowałaby do nazwy części? Wtedy trzeba ją przekazać jawnie jak tu:

```
<%= render :partial => "person", :locals => { :person => @adrian } %>
```

Wszystkie pary klucz – wartość w tablicy asocjacyjnej `:locals` będą dostępne jako zmienne lokalne części.

Dość częstym zastosowaniem części jest przeglądanie tablicy obiektów i generowanie części dla każdego obiektu. Metoda `render` sprawia, że jest to proste dzięki opcji `:collection`. Na przykład:

```
<%= render :partial => "person", :collection => @people %>
```

W tym przykładzie główny szablon zawiera tablicę `@people`, która będzie przeglądana, a każdy element tablicy — zmienna lokalna `person` — zostanie przekazany do części.

Domyślnie szablony części powinny znajdować się w tym samym katalogu co szablon główny. Aby wykorzystać części z poziomu innych kontrolerów, wystarczy dodać nazwę katalogu jako przedrostek. Na przykład:

```
<%= render :partial => "chapter1/person" %>
```

Pomimo że głównym szablonem jest *chapter2/index.rhtml*, część będzie generowana na podstawie pliku *chapter1/_person.rhtml*.

Pomocniki

Pomocniki są prostymi metodami Ruby dostępnymi w szablonach, dostarczającymi innego sposobu na to, by szablon pozostał czysty i czytelny. Dla każdego kontrolera tworzony jest jeden plik pomocnika, zatem `Chapter2Controller` będzie powiązany z plikiem *app/helpers/chapter2_helper.rb*. Jeśli chce się mieć pomocnika dostępnego dla wszystkich kontrolerów, należy zdefiniować go w *application_helper.rb*.

Rails dostarcza szereg wbudowanych pomocników, które są powszechnie używane — właściwie już widzieliśmy kilka z nich. W części „Układ graficzny" powyżej czwarty wiersz jest wywołaniem pomocnika:

```
<%= javascript_include_tag "prototype" %>
```

`javascript_include_tag()` jest metodą Ruby zdefiniowaną przez Rails, która pobiera jako argument łańcuch znaków (albo tablicę łańcuchów znaków) i zwraca fragment HTML, jak np.:

```
<script src="/javascripts/prototype.js" type="text/javascript"></script>
```

Innym użytecznym pomocnikiem jest `h`, który zamienia HTML na czysty tekst. Na przykład, `<%= h @foo %>` zapobiegnie zwróceniu znaków specjalnych HTML w wyjściu, zamieniając je na encje, co jest ważnym posunięciem ze względów bezpieczeństwa przy wyświetlaniu danych wprowadzonych przez użytkownika. Implikację tę będziemy rozważać dokładniej w rozdziale 8.

Być może najczęściej używanym pomocnikiem jest `link_to`, który w prosty sposób generuje łącze. Na przykład:

```
<%= link_to "Kliknij tutaj", :url => "/chapter2/myresponse" %>
```

Ten pomocnik zwraca: `<a href="/chapter2/myresponse">Kliknij tutaj</a>`.

Jest to całkiem trywialny przykład, ale interesującą sprawą jest to, że zamiast przekazywania zwykłego adresu URL jako parametru można przekazać nazwę kontrolera, nazwę akcji i inne parametry — a URL zostanie odpowiednio skonstruowany. Wspaniałe tutaj jest to, że gdy zmienia się ścieżki aplikacji, łącza automatycznie zostaną zmienione tak, aby pasowały do zmienionych ścieżek.

```
<%= link_to "Kliknij tutaj", :action => "myresponse" %>
```

Wyjście tej wersji jest takie samo jak powyżej. Proszę zauważyć, że nie określaliśmy nazwy kontrolera — została ona pominięta. Rails domyśla się, że chcemy użyć tego samego kontrolera, w którym właśnie się „znajdujemy".

Wewnętrznie `link_to` korzysta z innego pomocnika, `url_for` do stworzenia adresu URL łącza. Pomocnik `url_for` pobiera tablicę asocjacyjną elementów jako parametry i dopasowuje je na podstawie konfiguracji ścieżek aplikacji (pliku *routes.rb*), aby zwrócić URL. Inne klucze, które nie mają odpowiedniego obszaru w ścieżce, są dołączane jako łańcuch argumentów wejściowych. W dodatku istnieje kilka kluczy tablicy asocjacyjnej mających specjalne znaczenie:

- `:anchor` jest używany do dodawania kotwic (fragmentu URL po znaku #) do ścieżki.
- `:only_path` może być ustawiony na prawdę albo fałsz; jeśli zostanie użyta wartość `true`, protokół i fragment hostu URL zostaną pominięte.
- `:trailing_slash` może być ustawiony jako `true`, by do końca adresu URL dołączony został prawy ukośnik — co zwykle nie jest potrzebne i może powodować problemy z buforowaniem strony.
- `:host` może być określony, by wymusić inny adres hosta.
- `:protocol`, jeśli określony, zmienia aktualny protokół (np. HTTP, HTTPS, FTP).

Na przykład:

```
url_for :only_path => false, :protocol => 'gopher://',
  :host => 'example.com', :controller => 'chapter2',
  :action => 'myresponse', :trailing_slash => true, :foo => 'bar',
  :anchor => 'baz'
#=> 'gopher://example.com/chapter2/myresponse?foo=bar/#baz'
```

Pomysł oddzielania aktualnego URL od lokalizacji w obrębie aplikacji (kontrolera i akcji) jest centralnym założeniem Rails. To jest właściwie zawsze preferowane w celu wskazania lokalizacji aplikacji i pozwala się Rails tworzyć aktualną ścieżkę według zasad trasowania.

Wracając do Ajaksa

Omówiliśmy większość koncepcji systemu widoków Rails, które stanowią wszystko, co potrzebne do tego, by wrócić do Ajaksa. Do *myaction.rhtml* proszę dodać następujący fragment (zakładając, że *prototype.js* został już wcześniej załączony do tego dokumentu):

```
<p><%= link_to_remote "Powiadomienia z Pomocnika Javascript", :url =>
   "/chapter2/myresponse", :success => "alert(request.responseText)" %></p>
```

Ten przykład korzysta z pomocnika JavaScript `link_to_remote`, który jest ajaksowym wariantem pomocnika `link_to` omówionego wcześniej. Gdy spojrzy się na źródło wygenerowane przez pomocnika, można zobaczyć:

```
<p><a href="#" onclick="new Ajax.Request('/chapter2/myresponse',
{onSuccess:function(request){
  alert(request.responseText)
}}); return false;">Alert z pomocnika Javascript</a></p>
```

Ten kod robi dokładnie to samo, co nasz pierwszy ajaksowy przykład: tworzy łącze z atrybutem `onclick`, które tworzy obiekt `XMLHttpRequest` dla *chapter2/myresponse* i przekazuje wynik do `alert()`. Jeśli zamiast używać `alert()`, chce się wstawić tekst do strony, jest to jeszcze prostsze:

```
<p><%= link_to_remote "Aktualizacja z pomocnikiem Javascript", :url =>
  {:action => "myresponse"}, :update => "response5" %></p>
<p id="response5"></p>
```

Proszę zauważyć, że zamiast przekazywać opcję `:success`, przekazujemy opcję `:update`, która oczekuje ID elementu DOM. Kiedy `:update` jest określona, pomocnik korzysta z pochodzącego z Prototype wywołania `Ajax.Updater` zamiast `Ajax.Request`. Kolejna różnica: we wszystkich pozostałych dotychczasowych przykładach żądanie URL było określane jako ścieżka bezwzględna */chapter2/myresponse*. To działa, ale jest nieco ograniczone (wcześniej było to omawiane w części „Pomocniki"). Tym razem określamy już tylko nazwę akcji i pozwalamy, by właściwy URL został wygenerowany. Kod wygenerowany przez pomocnika wygląda następująco:

```
<p><a href="#" onclick="new Ajax.Updater('response5', '/chapter2/myresponse');
return false;"> Aktualizacja z pomocnikiem Javascript</a></p>
<p id="response5"></p>
```

Właśnie przekraczamy kamień milowy: po raz pierwszy utworzyliśmy wywołanie Ajaksa bez pisania czegokolwiek w JavaScripcie.

Podsumowanie

W tym rozdziale odkryliśmy mnóstwo podstaw przez budowanie prostych, działających wyłącznie po stronie klienta skryptów JavaScript, poprzez ręczne wywołania Ajaksa, później — dodawanie wsparcia z biblioteki Prototype i wreszcie pominięcie JavaScriptu dzięki pomocnikom Rails do JavaScript. Po przyswojeniu treści tego rozdziału czytelnik powinien mieć solidne podstawy do budowania aplikacji Ajax z pomocą Rails, a kilka kolejnych rozdziałów je umocni.