

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

Rails. Przepisy

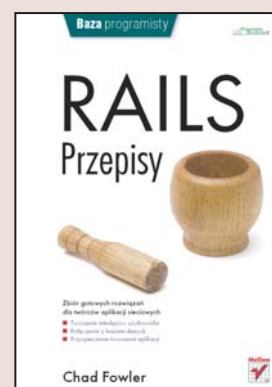
Autor: Chad Fowler

Tłumaczenie: Tomasz Bąk

ISBN: 83-246-0618-1

Tytuł oryginału: [Rails Recipes](#)

Format: B5, stron: 400



Zbiór gotowych rozwiązań dla twórców aplikacji sieciowych

- Tworzenie interfejsów użytkownika
- Połączenia z bazami danych
- Przyspieszanie tworzenia aplikacji

Rosnąca popularność aplikacji sieciowych, zastępujących w wielu zastosowaniach tradycyjne programy „biurowe”, sprawia, że środowiska służące do szybkiego tworzenia takich programów stają się niezbędnymi narzędziami pracy programistów. Dostępne w sieci struktury – frameworki – pozwalają im skoncentrować się wyłącznie na implementacji zadań, jakie ma spełniać aplikacja, co zdecydowanie usprawnia pracę. Wśród takich środowisk coraz większe uznanie zyskuje Ruby on Rails – bazujący na języku Ruby framework – który sprowadza do minimum nakład pracy konieczny do zbudowania aplikacji sieciowej.

Książka „Rails. Przepisy” to zbiór rozwiązań najczęściej wykonywanych zadań programistycznych przeznaczony dla tych twórców aplikacji, którzy w swojej pracy wykorzystują Ruby on Rails. Znajdziesz w niej porady, dzięki którym Twoja praca zyska na efektywności, a tworzone aplikacje będą szybsze i bezpieczniejsze. Przeczytasz o tworzeniu elementów interfejsu użytkownika, pobieraniu danych z baz, autoryzowaniu użytkowników, zarządzaniu sesjami i automatycznym generowaniu dokumentacji. Dowiesz się także, w jaki sposób przyspieszyć proces tworzenia aplikacji i jak zaimplementować w nich obsługę poczty elektronicznej.

- Sortowalne listy rozwijane
- Tworzenie wykresów
- Łączenie aplikacji z bazą danych
- Uwierzytelnianie użytkowników
- Automatyczne testowanie aplikacji
- Generatory kodu
- Przetwarzanie grafik
- Wysyłanie poczty elektronicznej

Przekonaj się, jak wiele możesz zyskać, korzystając z nowoczesnych narzędzi



Spis treści

Wprowadzenie	7
Część I Przepisy na interfejs użytkownika	11
1. Edycja formularza w miejscu	13
2. Tworzenie własnej metody pomocniczej JavaScript	21
3. Podgląd dynamiczny	29
4. Autouzupelnianie pola tekstowego	33
5. Tworzenie sortowalnych list typu przeciągnij i upuść	37
6. Aktualizacja wielu elementów w jednym żądaniu Ajax	45
7. Szybkie autouzupelnianie przy wykorzystaniu JavaScript	51
8. Tania i prosta obsługa motywów	57
9. Skracanie stron statycznych za pomocą Ajax	59
10. Inteligentna odmiana	61
11. Debugowanie Ajax	63
12. Tworzenie własnego konstruktora formularzy	65
13. Tworzenie ładnych wykresów	71
Część II Przepisy baz danych	77
14. Rails bez bazy danych	79
15. Łączenie się z wieloma bazami danych	85
16. Integracja z bazami innych aplikacji	95
17. Upraszczenie konfiguracji bazy danych	99

18. Powiązania modelu wiele-do-wielu z samym sobą	103
19. Znakowanie zawartości	107
20. Wersjonowanie modeli	115
21. Przejście na schematy oparte na migracjach	121
22. Powiązania wiele-do-wielu z dodatkowymi danymi	129
23. Powiązania polimorficzne — has_many :whatevers	135
24. Dodanie zachowań do powiązań Active Record	141
25. Dynamiczna konfiguracja bazy danych	147
26. Używanie Active Record poza Rails	149
27. Wykonywanie obliczeń na danych modelu	151
28. Upraszczanie kodu Active Record poprzez zawężanie	155
29. Inteligentne struktury danych z composed_of()	157
30. Bezpieczne używanie modeli w migracjach	161
Część III Przepisy kontrolera	163
31. Uwierzytelnianie użytkowników	165
32. Autoryzacja użytkowników za pomocą ról	173
33. Porządkowanie kontrolerów za pomocą akcji zwrotnych	179
34. Monitorowanie wygasających sesji	181
35. Renderowanie CSV w akcjach	185
36. Czytelne (i ładne) adresy URL	187
37. Podwaliny uwierzytelniania	193
38. Przejście na sesje Active Record	195
39. Pisanie kodu, który pisze kod	197
40. Zarządzanie stroną statyczną w Rails	205
Część IV Przepisy testów	207
41. Tworzenie dynamicznych obiektów fixture dla testów	209
42. Tworzenie obiektów fixture na podstawie rzeczywistych danych	215
43. Testowanie na przestrzeni wielu kontrolerów	221
44. Pisanie testów dla metod pomocniczych	229
Część V Przepisy ogólne	233
45. Automatyzacja tworzenia aplikacji poprzez własne generatory	235
46. Ciągła integracja kodu	243
47. Powiadamianie o nieobsłużonych wyjątkach	249

48. Tworzenie własnych zadań Rake	255
49. Radzenie sobie ze strefami czasowymi	263
50. Życie na krawędzi (rozwoju Rails)	271
51. Syndykowanie strony poprzez RSS	277
52. Tworzenie własnych wtyczek Rails	287
53. Sekretne URL-e	293
54. Szybki podgląd zawartości sesji	299
55. Współdzielenie modeli pomiędzy aplikacjami	301
56. Generowanie dokumentacji dla aplikacji	305
57. Przetwarzanie przesłanych zdjęć	307
58. Łatwe wyświetlanie list grupujących	313
59. Śledzenie tego, co kto zrobił	315
60. Dystrybucja aplikacji w postaci pojedynczego katalogu	321
61. Dodawanie obsługi lokalizacji	325
62. Konsola jest Twoim przyjacielem	333
63. Automatyczny zapis szkicu formularza	335
64. Walidacja obiektów spoza Active Record	339
65. Proste białe listy HTML	343
66. Dodawanie prostych usług sieciowych do naszych akcji	347
Część VI Przepisy poczty elektronicznej	353
67. Wysyłanie poczty elektronicznej o bogatej treści	355
68. Testowanie poczty przychodzącej	361
69. Wysyłanie wiadomości z załącznikami	371
70. Obsługa zwrotów wiadomości	375
Dodatki	383
A Zasoby	385
Skorowidz	387

Przepis 1.

Edycja formularza w miejscu

Problem

Nasza aplikacja ma jeden lub więcej fragmentów danych, które są często edytowane przez użytkowników — zwykle bardzo szybko. Chcemy umożliwić użytkownikom łatwą edycję danych **w miejscu**, bez otwierania osobnego formularza.

Rozwiązanie

Rails sprawia, że edycja w miejscu jest prosta dzięki kontrolce `script.aculo.us InPlaceEditor` i towarzyszących jej metodach pomocniczych. Przejdźmy od razu do rzeczy i wypróbujmy ją.

Najpierw stworzymy model i kontroler, na którym zademonstrujemy jej działanie. Założmy, że tworzymy prostą książkę kontaktów. Poniżej znajduje się kod migracji Active Record, którego użyjemy do definicji schematu:

```
InPlaceEditing/db/migrate/001_add_contacts_table.rb
```

```
class AddContactsTable < ActiveRecord::Migration
  def self.up
    create_table :contacts do |t|
      t.column :name, :string
      t.column :email, :string
      t.column :phone, :string
      t.column :address_line1, :string
      t.column :address_line2, :string
      t.column :city, :string
      t.column :state, :string
      t.column :country, :string
      t.column :postal_code, :string
    end
  end

  def self.down
    drop_table :contacts
  end
end
```

Następnie użyjemy domyślnie wygenerowanego modelu jako naszej klasy Contact. Aby szybko uruchomić nasz kod, wygenerujemy model, kontroler i widoki, wykorzystując generator rusztowania (*scaffolding*) Rails:

```
chad> ruby script/generate scaffold Contact
      exists app/controllers/
      :      :
      create app/views/layouts/contacts.rhtml
      create public/stylesheets/scaffold.css
```

Teraz możemy już uruchomić `script/server`, otworzyć w przeglądarce `http://localhost:3000/contacts/` i dodać kilka wpisów. Po kliknięciu łącza *Show* przy dopiero co utworzonym wpisie powinna ukazać się prosta, pozbawiona dekoracji strona prezentująca szczegóły danego wpisu w książce kontaktów. Na tej stronie dodamy naszą kontrolkę edycji w miejscu.

Pierwszym krokiem do wykorzystania Ajax jest upewnienie się, że w naszych widokach dołączone są niezbędne pliki JavaScript. Gdzieś w sekcji `<head>` naszego dokumentu HTML możemy wywołać:

```
<%= javascript_include_tag :defaults %>
```

Zwykle umieszczamy tę deklarację w domyślnym szablonie aplikacji (`app/views/layouts/application.rhtml`), aby nie martwić się o dołączenie jej (podobnie jak innych globalnych dla aplikacji ustawień, znaczników itd.) do każdego widoku, który tworzymy. Jeśli będziemy potrzebować efektów Ajax tylko w konkretnych sekcjach aplikacji, możemy zdecydować się na lokalne

dołączanie tych plików JavaScript. W naszym przypadku generator rusztowania utworzył dla nas szablon `contacts.rhtml` w katalogu `app/views/layouts`. Możemy załączyć JavaScript pod wywołaniem `stylesheet_link_tag()` w tym szablonie.

Po otwarciu w edytorze `app/views/contacts/show.rhtml` domyślnie powinniśmy zobaczyć coś takiego:

```
InPlaceEditing/app/views/contacts/show.rhtml.default
```

```
<% for column in Contact.content_columns %>
<p>
  <b><%= column.human_name %></b> <%=h @contact.send(column.name) %>
</p>
<% end %>

<%= link_to 'Edit', :action => 'edit', :id => @contact %> |
<%= link_to 'Back', :action => 'list' %>
```

Domyślnie widok `show()` iteruje w pętli po kolumnach modelu i wyświetla każdą z nich dynamicznie, wraz z etykietą i wartością. W przeglądarce widzimy coś podobnego jak na poniższym rysunku (prosty widok rusztowania):

Name: Chad Fowler
Email: chadfowler.com
Phone: 303-555-1212
Address line1: 321 Main St.
Address line2:
City: Gotham
State: CA
Country: USA
Postal code: 12345

[Edit](#) | [Back](#)

Zacznijmy od tego pliku i dodajmy kontrolki edycji w miejscu do naszych pól. Najpierw usuniemy łącze `Edit`, ponieważ nie będziemy już go potrzebować. Następnie opakujemy wyświetlaną wartość w wywołanie metody

pomocniczej edytora w miejscu. Nasz *show.rhtml* powinien wyglądać następująco:

InPlaceEditing/app/views/contacts/show.rhtml

```
<% for column in Contact.content_columns %>
<p>
  <b><%= column.human_name %></b>
  <%= in_place_editor_field :contact, column.name, {},
    { :rows => 1, :cancel_text => 'anuluj' } %>
</p>
<% end %>

<%= link_to 'Back', :action => 'list' %>
```

Mówimy metodzie pomocniczej `in_place_editor_field()`, że chcemy utworzyć kontrolkę edycji dla zmiennej egzemplarza o nazwie `@contact` o atrybucie, który właśnie przetwarza pętlę iterująca po nazwach kolumn. Aby być bardziej konkretnym, gdyby nie dynamiczne działanie rusztowania, musielibyśmy stworzyć kontrolkę edycji dla nazwy kontaktu w następujący sposób:

```
<%= in_place_editor_field :contact, :name %>
```

Zwróćmy uwagę, że `in_place_editor_field()` oczekuje **nazwy** zmiennej egzemplarza jako swojego parametru, a nie samej zmiennej (dlatego używamy `:contact`, a nie `@contact`).

Po odświeżeniu strony `show()` i kliknięciu wartości kontaktu kontrolka edycji powinna automatycznie otworzyć w bieżącym widoku:

Name:

Kliknięcie przycisku *ok* spowoduje zgłoszenie dużego, brzydkiego błędu JavaScript. Jest to zachowanie poprawne. Kontrolka edycji na miejscu utworzyła formularz do edycji danych kontaktu, ale formularz ten nie zawiera odpowiadającej mu akcji. Spoglądając szybko w pliki dzienników aplikacji, zobaczymy linie:

```
127.0.0.1 . . . "POST /contacts/set_contact_name/1 HTTP/1.1" 404 581
```

Aplikacja próbowała wywołać metodą POST akcję o nazwie `set_contact_name()` (zwróćmy uwagę na konwencję nazewniczą) i otrzymała kod 404 (nie znaleziono) w odpowiedzi.

Moglibyśmy teraz przejść do naszego `ContactsController` i zdefiniować metodę `set_contact_name()`, ale ponieważ robimy coś tak **konwencjonalnego**, możemy zdać się na **konwencję Rails**, która wykona za nas całą robotę! Otwórzmy `app/controllers/contacts_controller.rb` i dodajmy następujący wiersz zaraz na początku definicji klasy (drugi wiersz będzie dobrym miejscem):

```
in_place_edit_for :contact, :name
```

Wróćmy teraz do przeglądarki, wyedytujmy nazwę kontaktu i ponownie kliknijmy *ok*. Dane zostaną zmienione, zapisane i wyświetlone ponownie. Wywołanie `in_place_edit_for()` dynamicznie definiuje akcję `set_contact_name()`, która zaktualizuje za nas nazwę kontaktu. Inne atrybuty na stronie nadal nie będą działać, ponieważ nie poleciliśmy kontrolerowi wygenerować niezbędnych akcji. Moglibyśmy skopiować i wkleić wiersz, który właśnie dodaliśmy, zmieniając nazwy atrybutów. Ale ponieważ potrzebujemy kontrolki edycji dla wszystkich atrybutów modelu `Contact`, a rusztowanie już nam pokazało, w jaki sposób uzyskać nazwy kolumn modelu, zastosujmy się do zasady DRY (Don't Repeat Yourself — Nie powtarzaj się) i zamieńmy istniejące wywołanie `in_place_edit_for()` w następujący sposób:

```
InPlaceEditing/app/controllers/contacts_controller.rb
```

```
Contact.content_columns.each do |column|
  in_place_edit_for :contact, column.name
end
```

Teraz wszystkie atrybuty powinny być poprawnie zapisywane poprzez kontrolki edytora w miejscu. Jak już widzieliśmy, `in_place_edit_for` po prostu generuje odpowiednio nazwane akcje, które zajmują się aktualizacją danych. Gdybyśmy chcieli zaimplementować jakieś specjalne akcje obsługujące aktualizacje danych, moglibyśmy zdefiniować nasze własne akcje je obsługujące. Na przykład gdybyśmy potrzebowali specjalnego przetwarzania kodów pocztowych, moglibyśmy zdefiniować akcję `set_contact_postal_code()`.

raise() jest Twoim przyjacielem

Gdybym nie napisał, jak zaimplementować własne akcje edytora miejscowego, skąd mógłbyś wiedzieć, co zrobić?

Jak widzieliśmy w przepisie, możemy podejrzeć, jaką akcję próbuje wywołać kontrolka Ajax, przyglądając się logom serwera webowego. Ale ponieważ użyta jest metoda POST, nie widzimy parametrów w dziennikach. Jak się dowiedzieć, jakie parametry wykorzystuje automatycznie wygenerowany formularz bez czytania stosów kodów źródłowych?

Można stworzyć akcje o nazwie takiej, jaka pojawiła się w logach, która wygląda następująco:

```
def set_contact_name
  raise params.inspect
end
```

Po przyciśnięciu przycisku wysyłającego formularz zobaczymy komunikat o błędzie Rails zawierający na samej górze listę przekazanych parametrów.

Formularz kontrolki edytora w miejscu przekazuje dwa wartości parametry: identyfikator kontaktu, trafnie nazwany `id`, oraz nową wartość do zaktualizowania dla danego klucza — `value`.



Kontrolka edytora w miejscu używa metody `update_attribute()` Active Record do wykonania aktualizacji bazy danych. Metoda ta pomija walidację modelu Active Record. Jeśli musimy wykonywać walidację dla każdej aktualizacji, musimy napisać własne akcje obsługujące edytory w miejscu.

Pola edycyjne działają, ale są dość brzydkie. Jak, na przykład, zwiększyć długość pola tekstowego? Zwłaszcza długi adres email lub imię mogłoby nie zmieścić się w polu tekstowym o domyślnej długości. Wiele metod pomocniczych Rails akceptuje dodatkowe parametry, które będą przekazywane bezpośrednio do renderowanych przez nie elementów HTML, co pozwala na prostą kontrolę takich ich parametrów jak długość.

`InPlaceEditor` działa nieco inaczej (niektórzy powiedzieliby, że lepiej). Ustawia domyślną nazwę klasy dla generowanego formularza HTML, której możemy użyć do wyboru stylu CSS. Aby więc dostosować długość generowanych pól tekstowych, moglibyśmy użyć następującego wpisu CSS:

```
.inplaceeditor-form input[type="text"] {  
  width: 260px;  
}
```

Oczywiście, ponieważ używamy tu CSS, możemy wykonać wszystko, co jest możliwe w CSS.

Omówienie

Nasz przykład zakłada, że chcemy edytować wszystkie pola danych za pomocą pola tekstowego. W rzeczywistości możliwe jest wymuszenie na `InPlaceEditor` użycie albo pola tekstowego lub pola `<textarea>`, używając opcji `:rows` w czwartym parametrze metody `in_place_editor_field()`. Każda wartość większa od 1 spowoduje, że wygenerowane zostanie pole `<textarea>`.

A co, jeśli chcemy dokonywać edycji za pomocą innych kontroltek tekstowych? `InPlaceEditor` domyślnie nie zawiera nic odpowiedniego. W przepisie 2. powiemy, jak to zrobić.

Poza tym, jak można się domyślić, `InPlaceEditor` nie pozwoli nam na edycję pola, gdy nie zawiera ono jeszcze wartości. To ograniczenie może być ominięte poprzez umieszczanie pustych pól z domyślnymi wartościami, takimi jak „Kliknij, aby wyedytować”.