

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

RailsSpace. Tworzenie społecznościowych serwisów internetowych w Ruby on Rails

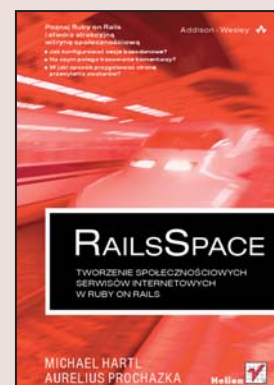
Autor: Michael Hartl, Aurelius Prochazka

Tłumaczenie: Marcin Rogóż

ISBN: 978-83-246-1633-6

Tytuł oryginału: [RailsSpace: Building a Social Networking Website with Ruby on Rails](#)
(Addison-Wesley Professional Ruby Series)

Format: 172x245, stron: 552



Poznaj Ruby on Rails i stwórz atrakcyjną witrynę społecznościową

- Jak konfigurować sesje bazodanowe?
- Na czym polega trasowanie komentarzy?
- W jaki sposób przygotować stronę przesyłania awatarów?

Serwisy społecznościowe, które gromadzą ludzi o podobnych zainteresowaniach i umożliwiają komunikację między znajomymi, cieszą się ogromną i wciąż rosnącą popularnością. Dzięki temu, że pozwalają na wymianę opinii i ułatwiają podtrzymywanie globalnych kontaktów, stają się elementami strategii biznesowych i marketingowych wielu firm. Do budowania takich serwisów doskonale nadaje się Rails, który oferuje klientom witryny w pełni dopasowane do potrzeb ich użytkowników. Rails został napisany w dynamicznym obiektowym języku Ruby z użyciem architektury MVC. Ten framework wyróżniają przede wszystkim dwie reguły: reguła DRY, polegająca na unikaniu wykonywania tej samej pracy w różnych miejscach, oraz reguła CoC., która pozwala na zminimalizowanie niezbędnej konfiguracji przez zastępowanie jej gotowymi, domyślnymi, zalecanymi wzorcami. Rails umożliwia także użycie wtyczek, rozszerzających aplikacje o rozmaite funkcjonalności np. logowanie, wrzucanie i skalowanie obrazków czy tagowanie.

Książka „RailsSpace. Tworzenie społecznościowych serwisów internetowych w Ruby on Rails” stanowi praktyczny kurs tworzenia interaktywnego serwisu społecznościowego. Za pomocą tego podręcznika nauczysz się budować taką witrynę, zaczynając od statycznej strony głównej, przez utworzenie mechanizmu rejestracji i uwierzytelnienia użytkowników, a kończąc na dynamicznej stronie WWW, z możliwością przesyłania obrazów i prowadzenia blogów, oraz systemie dodawania znajomych.

- Konfigurowanie środowiska programistycznego
- Modelowanie i rejestrowanie użytkowników
- Testowanie
- Ochrona stron
- Zaawansowane logowanie
- Aktualizacja informacji użytkownika
- Tworzenie sieci społecznej
- Awatary
- Model znajomości
- Blogi w technologii REST
- Komentarze do blogu w technologii AJAX

Samodzielnie zbuduj funkcjonalny serwis społecznościowy!!!



SPIS TREŚCI

	Spis rysunków	13
	Podziękowania	17
Rozdział 1.	Wprowadzenie	19
	1.1. Dlaczego Rails?	19
	1.1.1. Produktywność chce być wolna	20
	1.1.2. Ta produktywność nie jest wolna	20
	1.2. Dlaczego ta książka?	21
	1.3. Kto powinien przeczytać tę książkę?	22
	1.3.1. Jak czytać tę książkę?	22
	1.3.2. Jak oglądać tę książkę?	23
	1.4. Kilka historii związanych z Rails	23
	1.4.1. Aure	23
	1.4.2. Michael	25
Część I	Podstawy	29
Rozdział 2.	Zaczynamy	31
	2.1. Przygotowania	31
	2.1.1. Konfigurowanie środowiska programistycznego	33
	2.1.2. Praca z rails	33
	2.1.3. Serwer deweloperski	35
	2.2. Nasze pierwsze strony	37
	2.2.1. Generowanie kontrolera	38
	2.2.2. Kontroler Site	41
	2.2.3. URL w Rails	42
	2.2.4. Zmianianie trasy	44
	2.3. Widoki w Rails	44
	2.3.1. Osadzony Ruby (ERb)	45

2.4.	Układy	47
2.4.1.	ERb, akcje i zmienne egzemplarza	48
2.4.2.	Powtórka: podział strony	50
2.4.3.	Dodawanie nawigacji	50
2.4.4.	Tablice asocjacyjne	51
2.4.5.	Symbole	52
2.4.6.	Dopracowywanie link_to	53
2.4.7.	Kwestia stylu	54
2.4.8.	Dopracowywanie nawigacji	55
2.4.9.	Znajdź coś dla siebie	55
2.5.	Programowanie ze stylem	56
Rozdział 3.	Modelowanie użytkowników	61
3.1.	Tworzenie modelu User	61
3.1.1.	Konfigurowanie bazy danych	61
3.1.2.	Migracje i model User	64
3.1.3.	Pierwsza migracja użytkownika	65
3.1.4.	Rake migracji	66
3.2.	Walidacja modelu użytkownika	69
3.2.1.	Konsola	70
3.2.2.	Prosta walidacja	71
3.2.3.	Walidacje w akcji	75
3.2.4.	Poprawianie walidacji	75
3.2.5.	Porządne walidacje	77
3.2.6.	Magiczne kolumny	80
3.3.	Dalsze kroki w celu zapewnienia integralności danych (?)	82
Rozdział 4.	Rejestrowanie użytkowników	85
4.1.	Kontroler User	85
4.2.	Rejestracja użytkownika — widok	86
4.2.1.	Widok rejestracji — wygląd	87
4.2.2.	Omówienie widoku rejestracji	91
4.2.3.	Poprawianie formularza rejestracji	93
4.2.4.	Zabawa z formularzami i funkcją debug	95
4.3.	Rejestracja użytkownika — akcja	97
4.3.1.	Komunikaty o błędach formularza	103
4.3.2.	Flash	108
4.3.3.	Ukończona funkcja register	110
4.3.4.	Zarys głównej strony użytkownika	111
4.4.	Dołączanie rejestracji	113
4.4.1.	Pliki pomocnicze	115
4.5.	Przykładowy użytkownik	118
Rozdział 5.	Rozpoczynamy testowanie	119
5.1.	Nasza filozofia testowania	120
5.2.	Konfiguracja testowej bazy danych	120

5.3.	Testowanie kontrolera Site	121
5.3.1.	Niebanalny test	122
5.3.2.	Nadmiar testów?	125
5.4.	Testowanie rejestracji	126
5.4.1.	Uruchamianie testów funkcjonalnych	126
5.4.2.	Podstawowe testy rejestracji	126
5.4.3.	Testowanie rejestracji zakończonej powodzeniem	129
5.4.4.	Testowanie rejestracji zakończzonej niepowodzeniem	130
5.4.5.	Uruchamianie testów	133
5.4.6.	Więcej testów rejestracji?	133
5.5.	Podstawowe testy modelu User	133
5.5.1.	Podstawowe testy walidacji	135
5.6.	Szczegółowe testy modelu User	137
5.6.1.	Testowanie niepowtarzalności	138
5.6.2.	Testowanie długości pseudonimu	139
5.6.3.	Skorzystaj z konsoli	140
5.6.4.	Testowanie długości hasła	142
5.6.5.	Testowanie wyrażeń regularnych	144
5.6.6.	Uruchamianie wszystkich testów	151
Rozdział 6.	Logowanie i wylogowywanie	153
6.1.	Utrzymywanie stanu za pomocą sesji	154
6.1.1.	Konfigurowanie sesji bazodanowych	154
6.2.	Logowanie	156
6.2.1.	Rejestrowanie stanu zalogowania	156
6.2.2.	Logowanie po zarejestrowaniu	156
6.2.3.	Debugowanie ze zmienną sesji	157
6.2.4.	Widok i akcja logowania	162
6.2.5.	Testowanie poprawnego logowania	165
6.2.6.	Testowanie nieprawidłowego logowania	167
6.3.	Wylogowanie	168
6.3.1.	Testowanie wylogowania	169
6.3.2.	Testowanie nawigacji	170
6.4.	Ochrona stron	173
6.4.1.	Chronienie stron w głupi sposób	173
6.4.2.	Chronienie stron w mądry sposób	173
6.4.3.	Testowanie chronienia	176
6.5.	Przyjazne przekazywanie URL	178
6.5.1.	Zmienna request	178
6.5.2.	Przyjazne przekierowywanie po zalogowaniu	181
6.5.3.	Przyjazne przekierowywanie po rejestracji	183
6.5.4.	Przyjazne testowanie	184

6.6.	Refaktoryzacja podstawowego logowania	185
6.6.1.	Zalogowany?	186
6.6.2.	Zaloguj!	190
6.6.3.	Wyloguj!	193
6.6.4.	Wyczyść hasło!	194
6.6.5.	Obsługa formularza bez powtórzeń	197
6.6.6.	Przyjazne przekierowania bez powtórzeń	198
6.6.7.	Sprawdzamy poprawność	200
Rozdział 7.	Zaawansowane logowanie	201
7.1.	A więc chcesz być zapamiętany?	201
7.1.1.	Pole opcji „zapamiętaj mnie”	202
7.1.2.	Atrybut „pamiętaj mnie”	205
7.1.3.	Cookie „pamiętaj mnie”	206
7.2.	Faktyczne zapamiętywanie użytkownika	213
7.2.1.	Cookie uwierzytelniające	214
7.2.2.	Pamiętanie, że zapamiętaliśmy	216
7.2.3.	Aktualizacja logout	218
7.2.4.	Bardziej bezpieczny plik cookie	220
7.2.5.	Ukończone (?) funkcje	222
7.3.	Testy zapamiętywania użytkowników	224
7.3.1.	Poprawione testy logowania	224
7.3.2.	Poprawiona funkcja wylogowania	230
7.4.	Testy zaawansowane — testowanie integracji	230
7.4.1.	Testowanie pamiętania cookie — pierwsze cięcie	231
7.4.2.	Testowanie testu — opowieść ku przestrodze	233
7.4.3.	Kilka refleksji dotyczących testowania w Rails	235
7.5.	Ponowna refaktoryzacja	235
7.5.1.	Refaktoryzacja remember	236
7.5.2.	Refaktoryzacja forget	239
7.5.3.	Jeszcze dwie poprawki	240
7.5.4.	W pełni zrefaktoryzowana funkcja login	241
7.5.5.	Kilka końcowych przemyśleń	244
Rozdział 8.	Aktualizacja informacji użytkownika	245
8.1.	Sensowniejsza strona centrum użytkownika	246
8.2.	Aktualizacja adresu e-mail	246
8.3.	Aktualizacja hasła	248
8.3.1.	Obsługa przesyłania haseł	253
8.4.	Testowanie edycji informacji o użytkownikach	257
8.4.1.	Funkcje pomocnicze dla testów	259
8.4.2.	Testowanie strony edycji	262
8.4.3.	Zaawansowany test	263

8.5.	Części	266
8.5.1.	Dwie proste części	266
8.5.2.	Bardziej zaawansowany plik części	267
8.5.3.	Problem, a później koniec	270
8.5.4.	Aktualizacja akcji login i register	271

Część II Tworzenie sieci społecznościowej 275

Rozdział 9.	Profile osobiste	277
9.1.	Zaczątek profilu użytkownika	277
9.1.1.	Adresy URL profili	278
9.1.2.	Kontroler i akcje profilu	280
9.2.	Specyfikacja użytkownika	282
9.2.1.	Generowanie modelu Spec	282
9.2.2.	Model Spec	284
9.2.3.	Łączenie modeli	286
9.3.	Edycja specyfikacji użytkownika	288
9.3.1.	Kontroler Spec	288
9.3.2.	Narzędzie dla HTML	290
9.3.3.	Widok edycji specyfikacji	292
9.3.4.	Ochrona specyfikacji	293
9.3.5.	Testowanie specyfikacji	295
9.4.	Aktualizacja centrum użytkownika	299
9.4.1.	Nowy widok centrum użytkownika	300
9.4.2.	Pole specyfikacji	303
9.4.3.	Trasy nazwane i adres URL profilu	305
9.4.4.	Główna zawartość centrum użytkownika	307
9.5.	Osobisty FAQ — zainteresowania i osobowość	310
9.5.1.	Model FAQ	311
9.5.2.	Kontroler FAQ	314
9.5.3.	Edycja FAQ	315
9.5.4.	Dodawanie FAQ do centrum użytkownika	317
9.5.5.	Testy FAQ	320
9.6.	Upublicznianie profilu	322
Rozdział 10.	Społeczność	325
10.1.	Tworzenie społeczności (kontroler)	325
10.2.	Wprowadzanie przykładowych użytkowników	326
10.2.1.	Zbieranie danych	327
10.2.2.	Ładowanie danych	328
10.3.	Spis członków społeczności	330
10.3.1.	Nowy trik funkcji find	331
10.3.2.	Akcja index	334
10.3.3.	Spis alfabetyczny	336
10.3.4.	Wyświetlanie wyników indeksu	338

10.4.	Dopracowywanie wyników	343
10.4.1.	Dodawanie paginacji	343
10.4.2.	Podsumowanie wyników	347
Rozdział 11.	Wyszukiwanie i przeglądanie	349
11.1.	Wyszukiwanie	349
11.1.1.	Widoki wyszukiwania	350
11.1.2.	Ferret	353
11.1.3.	Wyszukiwanie za pomocą funkcji find_by_contents	355
11.1.4.	Dodawanie paginacji do wyszukiwania	357
11.1.5.	Wyjątek od reguły	361
11.2.	Testowanie wyszukiwania	363
11.3.	Rozpoczynamy przeglądanie	366
11.3.1.	Strona przeglądania	366
11.3.2.	Wyszukiwanie według wieku, płci i miejsca pobytu (na razie bez tego ostatniego)	368
11.4.	Miejsce pobytu	372
11.4.1.	Lokalna baza danych informacji geograficznych	373
11.4.2.	Używanie GeoData do wyszukiwania według miejsca pobytu	375
11.4.3.	Nazwy lokalizacji	378
11.4.4.	Dodawanie walidacji przeglądarki	381
11.4.5.	Ukończona strona główna społeczności	386
Rozdział 12.	Awatary	389
12.1.	Przygotowania do przesłania awataru	389
12.1.1.	Dostosowywanie modelu	390
12.1.2.	Strona przesyłania awatarów	392
12.1.3.	Plik części awataru	396
12.2.	Operowanie na awatarach	397
12.2.1.	ImageMagick i convert	398
12.2.2.	Metoda save	401
12.2.3.	Dodawanie walidacji	402
12.2.4.	Usuwanie awatarów	406
12.2.5.	Testowanie awatarów	409
Rozdział 13.	E-mail	413
13.1.	Action Mailer	413
13.1.1.	Konfiguracja	414
13.1.2.	Przypominanie hasła	415
13.1.3.	Tworzenie odnośnika i dostarczanie przypomnienia	416
13.1.4.	Testowanie przypominania	420
13.2.	Podwójnie ślepy system e-mail	423
13.2.1.	Odnośnik e-mail	424

	13.2.2.	Akcja correspond i formularz e-mail	425
	13.2.3.	Wiadomość e-mail	427
	13.2.4.	Testowanie podwójnie ślepego systemu poczty elektronicznej	430
Rozdział 14.	Znajomości		435
	14.1.	Tworzenie modelu znajomości	435
	14.1.1.	Znajomości w abstrakcji	436
	14.1.2.	Model znajomości	437
	14.1.3.	Tworzenie oczekujących znajomości	439
	14.1.4.	Propozycja zawarcia znajomości	440
	14.1.5.	Kończenie modelu Friendship	441
	14.1.6.	Testowanie modelu Friendship	443
	14.2.	Propozycje znajomości	444
	14.2.1.	Odnośnik do propozycji znajomości	444
	14.2.2.	Sterowanie propozycją	446
	14.3.	Zarządzanie znajomościami	449
	14.3.1.	has_many :through	449
	14.3.2.	Centrum znajomości	452
	14.3.3.	Akcje dla znajomości	455
	14.3.4.	Testowanie propozycji znajomości	458
Rozdział 15.	Blogi w technologii REST		461
	15.1.	Zasługujemy na REST	462
	15.1.1.	REST i CRUD	463
	15.1.2.	Modyfikatory URL	465
	15.1.3.	Śłoń;w pokoju	466
	15.1.4.	Odpowiadanie na formaty i darmowe API	468
	15.2.	Rusztowania dla blogów zgodnych z REST	469
	15.2.1.	Pierwszy zasób REST	469
	15.2.2.	Wpisy do blogu	471
	15.2.3.	Kontroler Post	474
	15.3.	Tworzenie prawdziwego blogu	478
	15.3.1.	Łączenie modeli	478
	15.3.2.	Trasowanie blogu i wpisu	479
	15.3.3.	Kontroler Posts	480
	15.3.4.	Zarządzanie blogiem	483
	15.3.5.	Tworzenie wpisów	485
	15.3.6.	Wyświetlanie wpisów	487
	15.3.7.	Edycja wpisów	490
	15.3.8.	Publikowanie wpisów	493
	15.3.9.	Ostatni, denerwujący szczegół	495
	15.4.	Testy REST	498
	15.4.1.	Domyślne testy funkcjonalne REST	498
	15.4.2.	Dwa niestandardowe testy	501

Rozdział 16.	Komentarze do blogu w technologii Ajax	503
16.1.	Komentarze zgodne z REST	503
16.1.1.	Zasób Comments	504
16.1.2.	Model i powiązania komentarza	505
16.1.3.	Kontroler Comments i zapobiegawczy widok części	507
16.1.4.	Trasowanie komentarzy	508
16.2.	Wprowadzamy Ajaksa	509
16.2.1.	Nowe komentarze	510
16.2.2.	Tworzenie komentarzy	514
16.2.3.	Niszczanie komentarzy	517
16.3.	Efekty wizualne	519
16.3.1.	Pliki RJS i pierwszy efekt	520
16.3.2.	Kolejne dwa efekty	521
16.3.3.	Przycisk anuluj	523
16.3.4.	Zgrabna degradacja	524
16.4.	Debugowanie i testowanie	526
16.4.1.	Inne spojrzenie na new	526
16.4.2.	Testowanie Ajax za pomocą xhr	527
Rozdział 17.	Co dalej?	529
17.1.	Co należy wziąć pod uwagę?	529
17.1.1.	Wybór sprzętu i oprogramowania	530
17.1.2.	Praca w trybie produkcyjnym	530
17.1.3.	Minimalny serwer produkcyjny	532
17.1.4.	Skalowanie	534
17.1.5.	Podstawy administrowania	536
17.2.	Więcej Ruby on Rails	539
	Skorowidz	541

ROZDZIAŁ 10.

Społeczność

Umożliwienie użytkownikom tworzenia i edycji profili jest dobrym początkiem, ale jeżeli RailsSpace ma być przydatne swoim członkom, musimy im dać możliwość odnalezienia siebie nawzajem. W tym i kolejnym rozdziale utworzymy trzy metody wyszukiwania użytkowników:

1. Prosty spis imion i nazwisk.
2. Przeglądanie według wieku, płci i miejsca pobytu.
3. Pełnotekstowe wyszukiwanie we wszystkich informacjach o użytkowniku, łącznie ze specyfikacją i FAQ.

W tym rozdziale umieścimy w deweloperskiej bazie danych RailsSpace przykładowych użytkowników, aby nasze próby odnalezienia użytkownika nie były bezcelowe. Następnie utworzymy alfabetyczny spis społeczności, aby utworzyć najprostszą listę członków RailsSpace. Choć prosty, spis członków RailsSpace pozwoli nam poznać kilka nowych aspektów Rails, takich jak paginacja wyników oraz niesamowita elastyczność funkcji `find`.

10.1. TWORZENIE SPOŁECZNOŚCI (KONTROLER)

Do wyszukania użytkownika będzie można użyć spisu członków społeczności, a także przeglądarki i wyszukiwarki użytkowników. „Przeglądać” i „wyszukiwać” są czasownikami, co sugeruje, że powinny być akcjami wewnątrz kontrolera. Chcemy kontynuować konwencję używania rzeczowników dla nazw kontrolerów, więc potrzebujemy odpowiedniego rzeczownika zbiorowego do opisanego zbioru użytkowników, który może być przeglądany i przeszukiwany. Ponieważ wyszukiwanie będzie odbywało się w społeczności użytkowników, utworzymy kontroler `Community` (społeczność):

```

> ruby script/generate controller Community index browse search
exists app/controllers/
exists app/helpers/
create app/views/community
exists test/functional/
create app/controllers/community_controller.rb
create test/functional/community_controller_test.rb
create app/helpers/community_helper.rb
create app/views/community/index.rhtml
create app/views/community/browse.rhtml
create app/views/community/search.rhtml

```

Dzięki temu użytkownicy będą mogli (na przykład) wyszukiwać innych użytkowników z użyciem URL

http://localhost:3000/community/search

i podobnie dla przeglądania (akcja browse). Zaktualizujmy teraz pasek nawigacji:

LISTING 10.1. app/views/layouts/application.rhtml

```

.
.
.
<%= nav_link "Pomoc",          "site", "help" %> |
<%= nav_link "Społeczność",    "community", "index" %>
.
.
.

```

Dalszą część tego rozdziału oraz rozdział następny poświęcimy wypełnianiu kontrolera Community. Jednak najpierw musimy rozwiązać problem podstawowy. W obecnej postaci wszystkie wysiłki podejmowane, by odnaleźć użytkowników RailsSpace, spełzną na niczym.

10.2. WPROWADZANIE PRZYKŁADOWYCH UŻYTKOWNIKÓW

Ponieważ niszowa witryna społecznościowa, jaką jest RailsSpace, może mieć setki, a nawet tysiące użytkowników, powinniśmy ją rozwijać, stosując bazę danych zawierającą wiele przykładowych wpisów. Dzięki temu różne sposoby przeglądania i wyszukiwania będą zwracały realistyczną liczbę wyników. Jednakże w tej chwili dysponujemy tylko jednym użytkownikiem — naszym starym przyjacielem Foo Barem — a dodawanie użytkowników samodzielnie, tak jak to robiliśmy z Foo, byłoby niezwykle praco- i czasochłonne. Co więcej, nasza deweloperska baza danych jest podatna na zniszczenie przez migracje i inne katastrofy. Nawet gdybyśmy wprowadzili samodzielnie dane wielu użytkowników, ryzykowalibyśmy utracenie tych danych.

Naszym rozwiązaniem jest wykorzystanie komputera do ciężkiej pracy. Utworzymy pliki YAML zawierające przykładową bazę danych użytkowników (a także odpowiadające im specyfikacje i FAQ). Następnie zautomatyzujemy ładowanie tych danych za pomocą własnego zadania Rake.

10.2.1. ZBIERANIE DANYCH

W tym podrozdziale utworzymy przykładowe dane użytkowników, specyfikacje i FAQ w formacie YAML. Naszym źródłem będą informacje o wyróżnionych absolwentach (Distinguished Alumni) Caltechu, dostępne publicznie pod adresem:

http://alumni/clatech.edu/distinguished_alumni

Jeżeli wolisz wypełnić pliki z danymi w inny sposób — nawet pisząc je samodzielnie — możesz to zrobić. Chodzi o to, aby mieć dane w formacie, który może być łatwo załadowany na żądanie, dzięki czemu w sytuacji, gdy coś się stanie bazie danych, będziemy mogli łatwo przywrócić jej poprzedni stan.

Gdybyś był zwykłym śmiertelnikiem, musiałbyś samodzielnie przepisywać informacje ze strony Distinguished Alumni, ale ponieważ to Aure tworzył witrynę Caltech Alumni, przykładowe dane są dostępne do pobrania w formacie YAML:

http://alumni.caltech.edu/distinguished_alumni/users.yml

http://alumni.caltech.edu/distinguished_alumni/specs.yml

http://alumni.caltech.edu/distinguished_alumni/faqs.yml

Te same pliki z danymi są dostępne pod adresem:

<ftp://ftp.helion.pl/przyklady/railsp>

Aby uzyskać wyniki przedstawiane w tym rozdziale, powinieneś pobrać te pliki YAML i umieścić je w katalogu

`lib/tasks/sample_data`

(będzie to wymagało utworzenia katalogu *sample_data*).

Przy okazji, dane o wyróżnionych absolwentach są mieszanką informacji prawdziwych i fałszywych. Użyliśmy prawdziwych imion i nazwisk oraz oficjalnych biografii (które wykorzystaliśmy w polu FAQ dla życiorysu), ale zmyśliliśmy daty urodzenia, miejsca pobytu i wiek. W przypadku miejsc pobytu umieściliśmy kody pocztowe w zakresie od 92101 (San Diego) do 98687 (Vancouver). W przypadku dat urodzenia tworzymy wrażenie, że absolwenci dostali nagrodę Distinguished Alumni w wieku 50 lat i zapisujemy im datę urodzenia 1 stycznia 50 lat przed otrzymaniem nagrody.

10.2.2. ŁADOWANIE DANYCH

Mając przykładowe dane użytkowników, musimy je skopiować z plików YAML do bazy danych. W zasadzie nadaje się do tego każda technika — możemy parsować plik, korzystając z Ruby (a nawet powiedzmy Perla, czy Pythona), ustanowić jakiegoś rodzaju połączenie bazy danych albo też jawnie wykonać wszystkie wstawienia. Jeżeli jednak się zastanowisz, Rails musi mieć już jakiś sposób, by to zrobić, ponieważ testy w Rails umieszczają w testowej bazie danych informacje z plików YAML, korzystając z plików *fixture*. Nasz sposób będzie polegał na zastosowaniu tego mechanizmu do wstawienia przykładowych danych do deweloperskiej bazy danych.

Moglibyśmy napisać skrypt w czystym Ruby, aby wykonać wstawienie danych, ale rozwiązaniem bardziej zgodnym z duchem Rails jest utworzenie w tym celu własnego zadania Rake. Wiąże się to z napisaniem własnego *Rakefile*. Nie powinno zaskoczyć Cię, że na takie pliki *Rakefile* znajduje się specjalne miejsce w drzewie katalogów Rails — katalog *lib/tasks* (teraz już wiesz, czemu umieściliśmy dane w katalogu *lib/tasks/sample_data*).

Ponieważ nasze zadania Rake wiążą się z ładowaniem przykładowych danych, nazwiemy nasz plik *sample_data.rake*. Pliki *Rakefile* zawierają serie *zadań* napisanych w Ruby. W naszym przypadku zdefiniujemy zadania *load* i *delete*:

LISTING 10.2. lib/tasks/sample_data.rake

```
# Zawiera zadania umożliwiające wczytanie i usunięcie przykładowych danych użytkowników
require 'active_record'
require 'active_record/fixtures'

namespace :db do
  DATA_DIRECTORY = "#{RAILS_ROOT}/lib/tasks/sample_data"
  namespace :sample_data do
    TABLES = %w(users specs faqs)
    MIN_USER_ID = 1000 # Początkowy identyfikator użytkownika w danych przykładowych

    desc "Ładowanie przykładowych danych."
    task :load => :environment do |t|
      class_name = nil # Używamy nil, aby Rails sam wybrał klasę
      TABLES.each do |table_name|
        fixture = Fixtures.new(ActiveRecord::Base.connection,
                               table_name, class_name,
                               File.join(DATA_DIRECTORY,
                                         ↳table_name.to_s))

        fixture.insert_fixtures
        puts "Załadowano dane z #{table_name}.yaml"
      end
    end

    desc "Usuwa przykładowe dane"
    task :delete => :environment do |t|
```

```

    User.delete_all("id >= #{MIN_USER_ID}")
    Spec.delete_all("user_id >= #{MIN_USER_ID}")
    Faq.delete_all("user_id >= #{MIN_USER_ID}")
  end
end
end

```

Nasza metoda wczytywania danych wykorzystuje pliki *fixture*, więc u góry pliku *Rakefile* umieściliśmy deklarację `require` dla biblioteki `fixtures Active Record`. Zgodnie ze standardową praktyką w plikach *Rakefile* poprzedzamy każde zadanie opisem (`desc`). Dzięki temu, gdy zapytamy `rake` o dostępne zadania, opisy `load` i `delete` zostaną wyświetlone na liście:

```

> rake --tasks
.
.
.
rake db:sample_data:delete # Usuwa przykładowe dane
rake db:sample_data:load  # Ładowanie przykładowych danych
.
.
.

```

Zwróć uwagę, że poprzez umieszczenie definicji zadań w blokach namespace sprawiamy, że zadania `Rake` mogą być wywoływane za pomocą tej samej składni, którą widzieliśmy przy okazji innych zadań, takiej jak:

```
> rake db:test:prepare
```

Zadanie `load` tworzy *fixture* za pomocą metody `Fixture.new`, która przyjmuje połączenie do bazy danych, nazwę tabeli, nazwę klasy i pełną ścieżkę do danych *fixture*:

```
Fixtures.new(connection, table_name, class_name, fixture_path)
```

Ponieważ ustawiliśmy `class_name` na `nil`, Rails będzie próbował wywnioskować nazwę klasy z nazwy tablicy. Skonstruowaliśmy też różne ścieżki, korzystając z `File.join`, która tworzy ścieżkę do pliku odpowiednią dla danej platformy. Po utworzeniu *fixture* wstawiamy dane do bazy, korzystając z metody `insert_fixtures`. Możemy cofnąć działanie `load`, stosując zadanie `delete`, wykorzystujące funkcję `Active Record delete_all` do usunięcia wszystkich danych odpowiadających użytkownikom, których identyfikator ma wartość większą niż 1000 (tym samym pozostawia użytkowników takich jak `Foo Bar`, których identyfikator ma mniejszą wartość).

A skąd *fixture* wie o (na przykład) klasie `User`? I skąd wie, w jaki sposób połączyć się z bazą danych? Odpowiedź tkwi w magicznym wierszu:

```
task :load => :environment do |t|
```

(i podobnie dla zadania `delete`). Wiersz ten oznacza, że zadanie `load` zależy od środowiska Rails. `Rake` odpowiada poprzez wczytanie lokalnego (deweloperskiego)

środowiska Rails, łącznie z modelami i połączeniami do bazy danych (które pobiera z pliku *database.yml*). Korzystając Rails do obsługi wszystkich tych szczegółów, Rake redukuje system do wcześniej rozwiązanego problemu.

Jeżeli chcesz, aby Twoje wyniki były zgodne z naszymi, zanim przejdziesz dalej, uruchom zadanie Rake, aby wczytać przykładowe dane:

```
> rake db:sample_data:load
(in /rails/rails_space)
Załadowano dane z users.yml
Załadowano dane z specs.yml
Załadowano dane z faqs.yml
```

10.3. SPIS CZŁONKÓW SPOŁECZNOŚCI

Tak jak w przypadku wszystkich pozostałych kontrolerów, utworzyliśmy akcję *index* dla kontrolera *Community* — ale po raz pierwszy nazwa „*index*” ma tu sens, ponieważ możemy wykorzystać tę stronę jako alfabetyczny indeks (spis) członków społeczności RailsSpace. Projekt, który chodzi nam po głowie, jest prosty. Wystarczy połączyć każdą z liter alfabetu z użytkownikami RailsSpace, których nazwisko rozpoczyna się od tej litery.

Implementacja tego projektu wymaga kilku różnych warstw, włącznie z paroma plikami części i opanowaniem nowych właściwości Active Record. Podczas implementowania różnych fragmentów warto wiedzieć, dokąd zmierzamy (rysunek 10.1). Zwróć uwagę, że adres URL

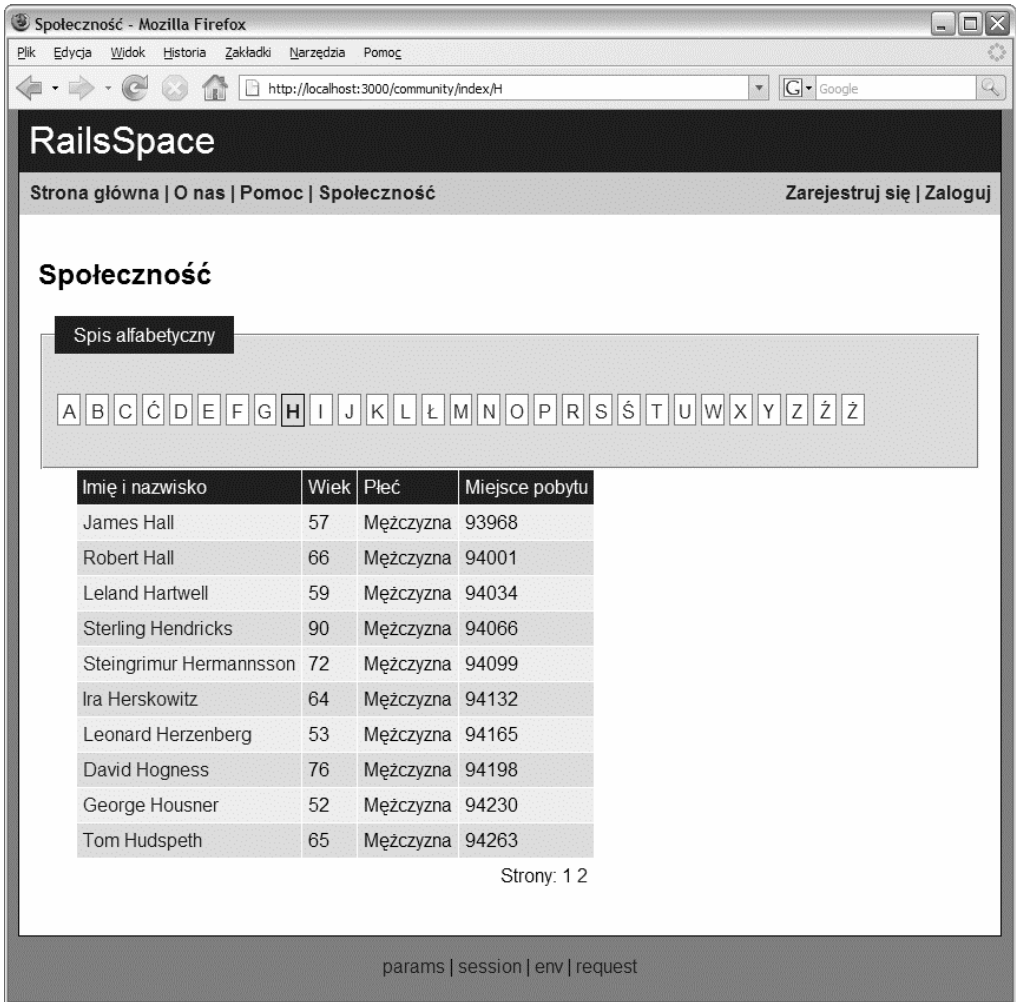
```
http://localhost:3000/community/index/H
```

zawiera cały zestaw parametrów — kontroler, akcję i identyfikator — obsługiwanych przez domyślną trasę w pliku *routes.rb* (punkt 2.2.4):

LISTING 10.3. *config/routes.rb*

```
ActionController::Routing::Routes.draw do |map|
  .
  .
  .
  # Install the default route as the lowest priority
  map.connect ':controller/:action/:id'
```

Warto wspomnieć, że każdy z tych trzech elementów jest dostępny w zmiennej *params*. Na przykład w tym przypadku *params[:id]* wynosi *H*.



RYSUNEK 10.1. Ukończony spis społeczności (pokazany dla litery H)

10.3.1. NOWY TRIK FUNKCJI FIND

Akcja `index` kontrolera `Community` będzie musiała wyszukać wszystkich użytkowników, których inicjał nazwiska to podana litera. Jak pamiętamy z podrozdziału 9.2, informacje o imieniu i nazwisku znajdują się w specyfikacji użytkownika. Musimy w jakiś sposób przeszukać specyfikacje, aby odnaleźć odpowiednie nazwiska.

Moglibyśmy dokonać tego typu wyszukiwania za pomocą surowego kodu SQL, stosując symbol zastępczy % do wyszukania wszystkich nazwisk rozpoczynających się (na przykład) literą N i wyświetleniu wyników w kolejności alfabetycznej według nazwiska¹:

```
SELECT * FROM specs WHERE last_name LIKE 'N%'
ORDER BY last_name, first_name
```

Co oczywiste, Active Record zapewnia warstwę abstrakcji dla tego typu zapytania. Bardziej zaskakujący jest fakt, że rozwiązanie wykorzystuje metodę `find`, którą wcześniej widzieliśmy przy wyszukiwaniu elementów według identyfikatora:

```
User.find(session[:user_id])
```

To nie jest kres możliwości metody `find`. Jest to funkcja całkiem elastyczna, która potrafi wykonać wiele różnych zapytań. Konkretnie, przesyłając do funkcji `find` opcje `:all`, `:conditions` i `:order`, możemy wyszukiwać wszystkich użytkowników, których nazwiska rozpoczynają się od litery N:

```
> ruby script/console
Loading development environment.
>> initial = "N"
>> Spec.find(:all, :conditions => "last_name LIKE '#{initial}%",
↳?> :order => "last_name, first_name")
=> [#<Spec:0x36390a4 @attributes={"city"=>"", "occupation"=>"",
↳"birthdate"=>"19
36-01-01", "zip_code"=>"96012", "gender"=>"Male", "id"=>"731",
↳"first_name"=>"Ro
ddam", "user_id"=>"1117", "last_name"=>"Narasimha", "state"=>""}>,
↳#<Spec:0x3638
f3c @attributes={"city"=>"", "occupation"=>"",
↳"birthdate"=>"1945-01-01", "zip_c
ode"=>"96045", "gender"=>"Male", "id"=>"655", "first_name"=>"Jerry",
↳"user_id"=>
"1118", "last_name"=>"Nelson", "state"=>""}>, #<Spec:0x3638dac
↳@attributes={"cit
y"=>"", "occupation"=>"", "birthdate"=>"1941-01-01",
↳"zip_code"=>"96079", "gende
r"=>"Male", "id"=>"713", "first_name"=>"Navin", "user_id"=>"1119",
↳"last_name"=>
"Nigam", "state"=>""}>, #<Spec:0x3638ba4 @attributes={"city"=>"",
↳"occupation"=>
"", "birthdate"=>"1939-01-01", "zip_code"=>"96112", "gender"=>"Male",
↳"id"=>"723
", "first_name"=>"Robert", "user_id"=>"1120", "last_name"=>"Noland",
↳"state"=>""
}
}
```

¹ Posortowanie wyników według `last_name`, `first_name` porządkuje je najpierw według nazwiska, a potem imienia, dzięki czemu na przykład Michelle Feynman znajdzie się przed Richardem Feynmanem, a oboje zostaną wyświetleni przed Murrayem Gellem-Manem.

Powyższy kod daje taki sam wynik jak czysty SQL przedstawiony wcześniej i w obecnej postaci działa dobrze. Oczekujemy jednak, że w RailsSpace inicjał będzie pochodził z internetu i będzie dostarczany za pośrednictwem `params[:id]`. Ponieważ użytkownik może wpisać dowolny „inicjał”, złośliwy haker mógłby umieścić w `params[:id]` łańcuch zdolny do wykonania dowolnych instrukcji SQL — łącznie (ale nie tylko) z usunięciem bazy danych². Aby zapobiec takiemu atakowi — zwanemu *wstrzyknięciem* kodu SQL — musimy zabezpieczyć wszystkie łańcuchy wstawiane do instrukcji SQL. W Active Record można to zrobić, używając znaku `?` jako symbolu zastępczego:

```
Spec.find(:all, :conditions => ["last_name LIKE ?", initial+"%"],
         :order => "last_name, first_name")
```

Dzięki temu, gdy użytkownik wpisze

```
http://RailsSpace.com/community/index/<niebezpieczny łańcuch>
```

aby uruchomić niebezpieczne zapytanie, niebezpieczny łańcuch zostanie przekształcony na coś niegroźnego przed wstawieniem do klauzuli warunków. Tak się składa, że nie możemy napisać

```
:conditions => ["last_name LIKE ?", initial]
```

ponieważ Rails próbowałby uruchomić zapytanie zawierające

```
last_name LIKE 'N'%
```

co jest nieprawidłowe.

Zwróć uwagę, że w przypadku bezpiecznej wersji wartością `:conditions` nie jest łańcuch, a tablica, której pierwszy element jest łańcuchem zawierającym warunki, a kolejnymi elementami są łańcuchy, które powinny być zabezpieczone i wstawiane. Możemy wymusić kilka warunków, stosując kilka znaków zapytania³:

```
Spec.find(:all, :conditions => ["first_name = ? AND last_name = ?",
                                "Foo", "Bar"])
```

Oczywiście w powyższym przypadku moglibyśmy również zapisać

```
Spec.find_by_first_name_and_last_name("Foo", "Bar")
```

A funkcja ta sama zastosuje zabezpieczenia. Jest to przykład tego, jak przy uruchamianiu zapytań SQL Active Record umożliwia przechodzenie na różne poziomy abstrakcji, dając użytkownikowi to, co najlepsze z dwóch światów — domyślnie wygodę, a w ramach potrzeb maksimum możliwości (patrz ramka „Przebijanie się przez abstrakcję”).

² Nawet jeżeli Rails będzie miał dostęp do bazy danych jako użytkownik MySQL z ograniczonymi prawami dostępu (a tak na pewno będzie w środowisku produkcyjnym), umożliwienie wydawania dowolnych poleceń wciąż jest złe.

³ Drugi sposób wstawiania wielu warunków znajdziesz w punkcie 11.3.2.

Przebijanie się przez abstrakcję

Jedną z głównych zasad projektowych Rails jest zapewnienie warstwy łatwych w użyciu funkcji wysokiego poziomu dla powszechnie wykonywanych zadań, ale również pozostawienie furtki do korzystania z warstw leżących poniżej. Na przykład widzieliśmy, że w celu odnalezienia użytkownika według pseudonimu i hasła Rails tworzy funkcję o nazwie

```
User.find_by_screen_name_and_password(screen_name, password)
```

Widzieliśmy również, jak zejść do niższej warstwy, korzystając z funkcji `find`:

```
spec = Spec.find(:all, :conditions => "last_name LIKE 'N%",
                 :order => "last_name, first_name"
```

Jeżeli chcesz, możesz zejść do kolejnej warstwy i użyć czystego kodu SQL:

```
spec = Spec.find_by_sql("SELECT * FROM specs
                        WHERE last_name LIKE 'N%'
                        ORDER BY last name, first name")
```

Jest to takie samo zapytanie jak powyższe, ale ponieważ `find_by_sql` stosuje czysty SQL, możemy w ten sposób dokonywać dowolnych zapytań⁴. A więc na przykład, jeżeli wąskim gardłem aplikacji jest jakieś nadmiernie rozbudowane zapytanie — co czasem można doskonale rozwiązać za pomocą czystego kodu SQL — zawsze możesz przejść do najniższej warstwy i utworzyć optymalne rozwiązanie.

10.3.2. AKCJA INDEX

Jak wspominaliśmy wcześniej, spis członków społeczności będzie stanowił katalog użytkowników witryny RailsSpace. Dzięki nowym umiejętnościom, które nabyliśmy w pracy z Active Record, możemy pobrać dane użytkowników, których nazwisko rozpoczyna się określoną literą. Oprócz tego musimy tylko utworzyć kilka zmiennych egzemplarza do wykorzystania w widokach:

LISTING 10.4. `app/controllers/community_controller.rb`

```
class CommunityController < ApplicationController
  helper :profile

  def index
    @title = "Społeczność"
    @letters = "ABCDEFHIJKLŁMNOPRSSTUWXYZZZ".split("")
    if params[:id]
      @initial = params[:id]
      specs = Spec.find(:all,
```

⁴ Dla naprawdę dowolnych zapytań możesz nawet użyć `Active::Record::Base.connection.execute` ↪ (query), gdzie query jest czystym poleceniem SQL, takim jak "DROP TABLE users".

```

        :conditions => ["last_name like ?",
          ↳@initial+'%'],
        :order => "last_name, first_name")
    @users = specs.collect { |spec| spec.user }
  end
end

def browse
end

def search
end
end

```

Zwróć uwagę, że dołączyliśmy plik pomocniczy Profile (stosując helper `:profile`), ponieważ w spisie członków społeczności użyjemy `profile_for` do utworzenia odnośników do profili użytkowników.

W tej akcji znajduje się kilka nowych elementów składni Rubi. Pierwszym i najprostszym jest

```
"ABCĆDEFGHIJKLŁMNOPRSŚTUWXYZŻ".split("")
```

Tworzona jest tablica łańcuchów, po jednym dla każdej litery alfabetu. Wykorzystujemy tutaj metodę `split`, którą możesz znać z Perla, Pythona lub jednego z wielu języków, w których istnieje podobna funkcja. Najczęściej funkcja `split` jest używana do dzielenia łańcucha na tablicę na podstawie białego znaku, ale może również dzielić na podstawie innych łańcuchów, co pokazuje ten przykład w `irb`:

```

> irb
irb(main):001:0> "foo bar baz".split
=> ["foo", "bar", "baz"]
irb(main):002:0> "1foo2fooredfoobbluefoo".split("foo")
=> ["1", "2", "red", "blue"]

```

W przypadku akcji `index` użycie pustego łańcucha `""` rozdziela podany łańcuch na jego znaki składowe:

```

irb(main):003:0> "ABCDEFGHIJKLMNOPQRSTUVWXYZ".split("")
=> ["A", "B", "C", "D", "E", "F", "G", "H", "I", "J", "K", "L", "M",
↳"N", "O", "P", "Q", "R", "S", "T", "U", "V", "W", "X", "Y", "Z"]

```

(Oczywiście moglibyśmy również napisać

```
%w(A Ą B C Ć D E Ę F G H I J K Ł M N Ń O Ó P R S Ś T U W X Y Z Ż Ż)
```

jednak byłoby to więcej wpisywania, niż byśmy chcieli, a poza tym już najwyższy czas, abyśmy przedstawili istotną funkcję `split`).

Drugim i bardziej istotnym fragmentem składni Rubi jest nasza metoda tworzenia zmiennej egzemplarza `@users`. W akcji `index` społeczności wiersz

```
users = specs.collect { |spec| spec.user }
```

kroczy przez specs i tworzy tablicę odpowiednich użytkowników⁵. Jak możesz domyśleć się z kontekstu, nawiasy klamrowe {...} są alternatywną składnią bloków Ruby. Działanie przedstawionego tu kodu jest w zasadzie identyczne⁶ jak składni, której używaliśmy poprzednio, czyli do...end:

```
users = specs.collect do |spec|
  spec.user
end
```

Jeżeli chcesz, możesz użyć składni z nawiasami w kilku wierszach:

```
users = specs.collect { |spec|
  spec.user
}
```

Wybór wersji jest po prostu kwestią konwencji. My przestrzegamy konwencji prezentowanej w dwóch naszych ulubionych książkach o Ruby — *Programowanie w języku Ruby* oraz *Ruby. Tao programowania w 400 przykładach*: używaj składni z nawiasami w blokach jednowierszowych, a składni do...end w blokach wielowierszowych.

10.3.3. SPIS ALFABETYCZNY

Czas zaprząć nasze zmienne egzemplarza do pracy w widoku spisu społeczności. Rozpoczniemy od samego wyświetlenia spisu, który będzie po prostu listą liter:

LISTING 10.5. app/views/community/index.rhtml

```
<h2><%= @title %></h2>
<fieldset>
  <legend>Spis alfabetyczny</legend>
  <% @letters.each do |letter| %>
  <% letter_class = (letter == @initial) ? "letter_current" :
  ↳"letter" %>
  <%= link_to letter, {:action => "index", :id => letter },
  :class => letter_class %>

  <% end %>
  <br clear="all" />
</fieldset>
```

Iterujemy przez wszystkie litery alfabetu, korzystając z metody each (inny sposób znajdziesz w ramce „for letter in @letters?”) i dla każdej litery definiujemy klasę CSS (za pomocą operatora trójkowego), aby określić, czy dana litera jest aktualnie wybrana. Następnie tworzymy odnośnik powrotny do strony index z bieżącą literą jako identyfikatorem (parametrem id).

⁵ Metodę collect widzieliśmy po raz pierwszy w punkcie 5.6.5 podczas tworzenia listy poprawnych adresów e-mail dla testowania walidacji.

⁶ Jedyną różnicą jest fakt, że nawiasy mają pierwszeństwo przed do...end, ale to rzadko ma znaczenie.

for letter in @letters?

Do skonstruowania listy alfabetycznej dla spisu członków społeczności używamy składni:

```
<% @letters.each do |letter| %>
.
.
.
<% end %>
```

Jest to w Ruby kanoniczny sposób iteracji przez tablicę, ale powinniśmy wiedzieć, że wewnątrz widoków niektórzy programiści Rails wykorzystują składnię alternatywną:

```
<% for letter in @letters %>
.
.
.
<% end %>
```

Jest tak prawdopodobnie dlatego, że ich zdaniem taka składnia będzie bardziej zrozumiała dla nie-programistów — na przykład projektantów stron — którzy mają szansę na nią natrafić.

Nic nam nie przeszkadza w składni alternatywnej — jest taka sama jak główny konstrukt pętli w Pythonie, który uwielbiamy — ale użycie `each` jest zdecydowanie bardziej „w stylu Ruby”: w tym języku zwykle do przesyłania instrukcji do obiektów używa się metod⁷ — w tym przypadku używamy `each`, aby „poinstruować” tablicę, by zwracała po kolei swoje elementy. Ponieważ nie widzimy przekonującego powodu, by rozdzielać style, pozostaniemy przy `each` nawet w widokach.

Należy podkreślić, że nawiasy okrągłe wokół `{ :action => "index", :id => letter }` są niezbędne do wywołania `link_to`. Argumenty funkcji `link_to` mają postać:

```
link_to(name, options = {}, html_options = nil)
```

Potrzebujemy nawiasów klamrowych, aby określić, gdzie kończy się tablica asocjacyjna z opcjami, a zaczyna tablica asocjacyjna z opcjami HTML. Gdybyśmy napisali

```
<%= link_to letter, :action = "index", :id => letter, :class =>
↳letter_class %>
```

cała tablica asocjacyjna

```
:action = "index", :id => letter, :class => letter_class
```

zostałaby przyjęta jako `options`. W wyniku tego zamiast odnośników w postaci

```
<a href="/community/index/A" class=letter">A</a>
```

⁷ Filozofia projektu, zwana „przesyłaniem komunikatów”, jest w dużej mierze inspirowana przez Smalltalk.

otrzymalibyśmy odnośniki w poniższej formie:

```
<a href="/community/index/A?class=letter">A</a>
```

a zupełnie nie o to nam chodzi.

Aby uzyskać żądany wygląd spisu społeczności, wykorzystamy niesamowite możliwości CSS w nadawaniu stylu znacznikom zakotwiczenia (a). Wystarczy, że dodamy poniższe reguły do pliku *site.css*:

LISTING 10.6. public/stylesheets/site.css

```
/* Style dla społeczności */
a, a#visited {
  color: maroon;
  text-decoration: none;
}

.letter, .letter_current {
  width: 0.9em;
  text-align: center;
  border: 1px solid gray;
  background: #fff;
  padding: 5px 2px 1px 2px;
  float: left;
  margin: 2px
}

.letter:hover {
  background: #fe4;
}

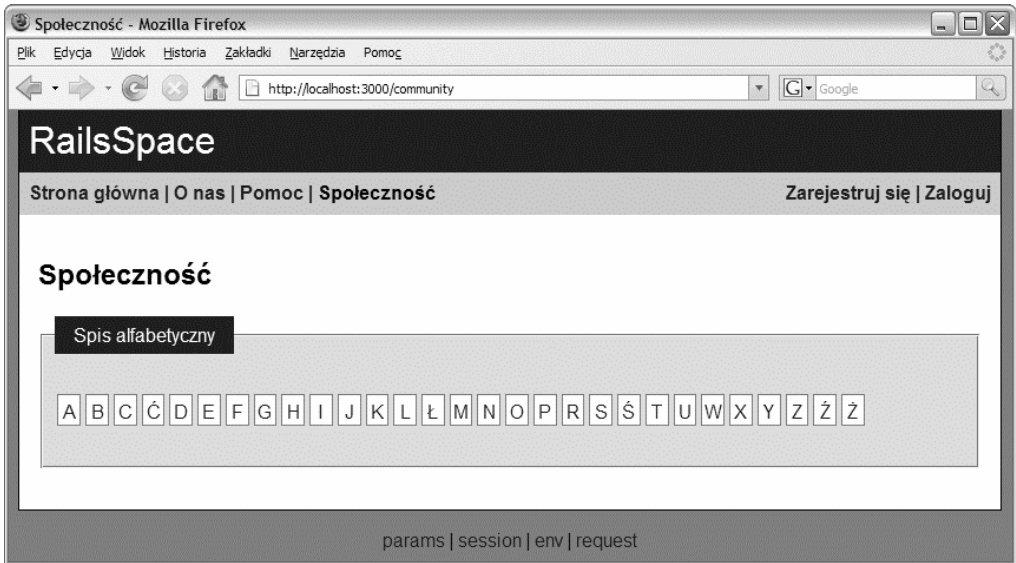
.letter_current {
  background: #fe4;

  font-weight: bold;
  border: 1px solid black;
}
```

Strona spisu społeczności wygląda już całkiem dobrze (rysunek 10.2), choć jeszcze tak naprawdę nic nie robi. Zajmijmy się teraz drugą częścią.

10.3.4. WYŚWIETLANIE WYNIKÓW INDEKSU

W punkcie 10.3.2 akcja `index` społeczności tworzyła zmienną egzemplarza `@users`, zawierającą użytkowników do wyświetlenia w widoku. Wykorzystamy tę zmienną w tabeli z wynikami, którą umieścimy w pliku części `/app/views/community/_user_table.rhtml`. Najpierw musimy wywołać ten plik części z pliku `index.rhtml`:



RYSUNEK 10.2. Strona społeczności RailsSpace z ładnie wystylizowanym indeksem alfabetycznym

LISTING 10.7. `app/views/community/index.rhtml`

```

.
.
.
<%= render :partial => "user_table" %>

```

Ten plik będzie tworzył tabelę wyników (jeżeli będą jakieś wyniki do wyświetlenia) poprzez iterację przez zawartość zmiennej `@users` w celu utworzenia wiersza tabeli dla każdego użytkownika:

LISTING 10.8. `app/views/community/_user_table.rhtml`

```

<% if @users and not @users.empty? %>
<table class="users" border="0" cellpadding="5" cellspacing="1">
  <tr class="header">
    <th>Imię i nazwisko</th> <th>Wiek</th> <th>Płeć</th> <th>Miejsce
      ↳ pobytu</th>
    </tr>

    <% @users.each do |user| %>
    <tr class="<%= cycle('odd', 'even') %>">
      <td><%= link_to user.name, profile_for(user) %></td>
      <td><%= user.spec.age %></td>
      <td><%= user.spec.gender %></td>
      <td><%= user.spec.location %></td>
    </tr>

```



```

    <% end %>
  </table>
<% end %>

```

Zwróć uwagę, że użycie funkcji pomocniczej `cycle`, która (domyślnie) zwraca raz jeden, raz drugi argument⁸, sprawiło, że przypisanie naprzemiennych stylów CSS jest banalne. Zwróć też uwagę, że w wywołaniu `link_to` użyliśmy funkcji `profile_url` wygenerowanej przez regułę trasowania, którą wprowadziliśmy w punkcie 9.1.1:

LISTING 10.9. `config/routes.rb`

```

map.connect 'profile/:screen_name', :controller = 'profile', :action
↳=> 'show'

```

Użyliśmy również nowej metody `name` z modelu `User`, która zwraca imię i nazwisko użytkownika, jeżeli informacje te są dostępne, a w przeciwnym przypadku zwraca pseudonim:

LISTING 10.10. `app/models/user.rb`

```

# Zwraca rozsądną nazwę użytkownika
def name
  spec.full_name.or_else(screen_name)
end

```

Tę funkcję można również wykorzystać w plikach `app/views/user/index.rhtml` (z punktu 9.4.4) oraz `app/views/profile/show.rhtml` (z podrozdziału 9.6). Jeżeli chcesz, zastosuj je w tych plikach.

Aby nasz plik części działał, musimy zrobić jeszcze jedną rzecz — dodać metodę `age` do modelu `Spec`, aby `@user.spec.age` istniało:

LISTING 10.11. `app/models/spec.rb`

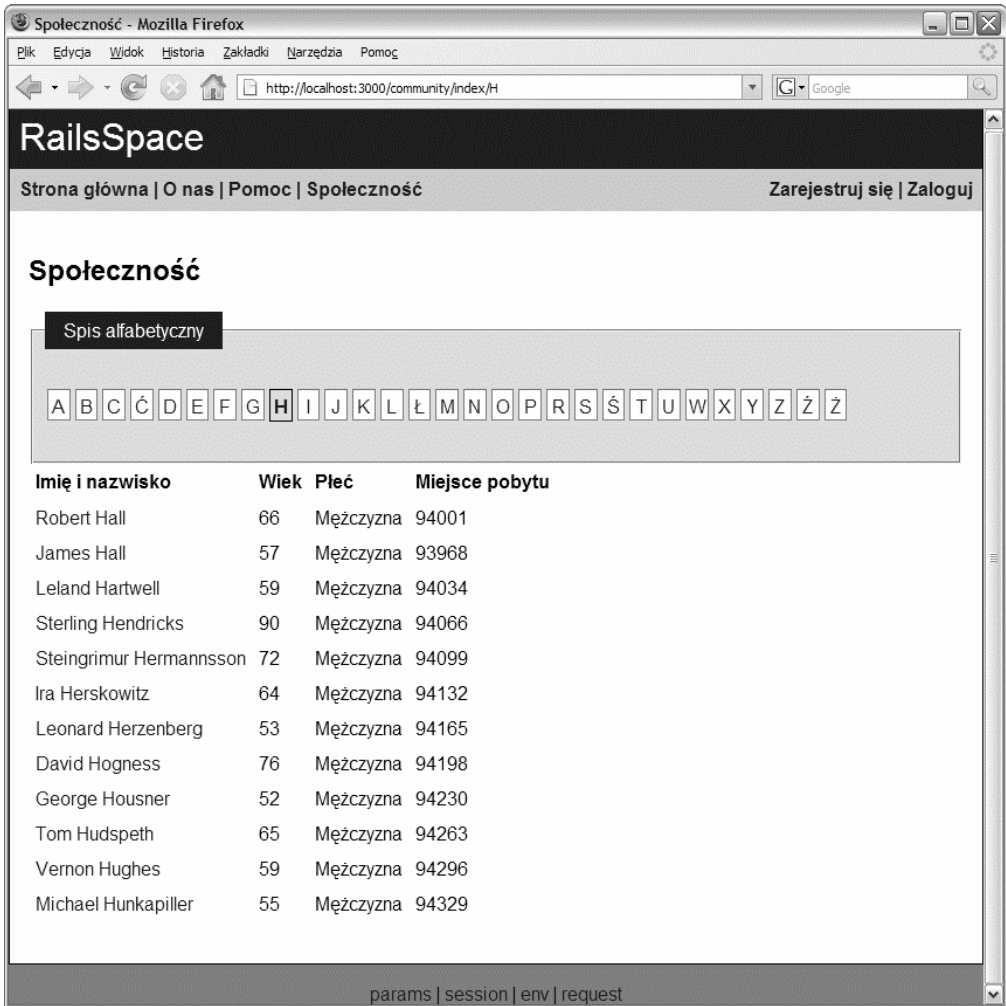
```

# Zwraca wiek obliczony na podstawie daty urodzenia
def age
  return if birthdate.nil?
  today = Date.today
  if (today.month > birthdate.month) or
    (today.month == birthdate.month and today.day >= birthdate.day)
    # Urodziny już były w tym roku
    today.year - birthdate.year
  else
    today.year - birthdate.year - 1
  end
end

```

⁸ Bardziej wyrafinowane przykłady zastosowania `cycle` znajdziesz w API Rails.

W zasadzie ukończyliśmy tworzenie funkcjonalności, co obrazuje rysunek 10.3, ale strona nie wygląda ładnie. Aby do wyników wyszukiwania dodać nieco stylu — na przykład naprzemiennie stosowanie stylów dla wierszy na podstawie `cycle` — dodaj poniższe reguły do sekcji *Style dla społeczności* w pliku `site.css`:



RYСУNEK 10.3. Ostateczna postać spisu społeczności

LISTING 10.12. `public/stylesheets/site.css`

```
/* Style dla społeczności */
.
.
.
table.users {
```

```

    background: #fff;
    margin-left: 2em;
  }

table.users td.bottom {
  border-top: 1px solid #999;
  padding-top: 10px;
}

table.users th {
  color: white;
  background: maroon;
  font-weight: normal;
}

table.users th a {
  color: white;
  text-decoration: underline;
}

table.users tr.even {
  background: #ddd;
}

table.users tr.odd {
  background: #eee;
}

```

Musimy wprowadzić jeszcze jedną drobną zmianę, aby wszystko działało jak należy. Trzeba zmienić funkcję tworzenia odnośnika w pasku nawigacji w pliku pomocniczym `Application`:

LISTING 10.13. `app/helpers/application_helper.rb`

```

# Zwraca odnośnik do wykorzystania w układzie nawigacji
def nav_link(text, controller, action="index")
  link_to_unless_current text, :id => nil,
                        :action => action,
                        :controller => controller
end

```

Powód, dla którego to niezbędne, jest dość subtelny. Bez jakiegokolwiek identyfikatora w wywołaniu `link_to_unless_current` Rails nie będzie widział różnicy między `/community/index a`, powiedzmy, `/community/index/A`. W wyniku tego odnośnik *Spółeczność* w pasku nawigacji nie będzie wyświetlany, dopóki nie dodamy opcji `:id => nil`.

Musimy również zmodyfikować trasę dla głównej strony naszej witryny, aby wziąć pod uwagę obecność identyfikatora nil:

LISTING 10.14. config/routes.rb

```

.
.
.
  # You can have the root of your site routed with map.root
  # -- just remember to delete public/index.html.
  map.connect '', :controller => 'site', :action => 'index', :id =>
    ↳nil
.
.
.

```

Dzięki temu / wciąż będzie automatycznie kierowało do */site/index*.

Po zajęciu się tym drobiazgiem ukończyliśmy w końcu spis członków społeczności (rysunek 10.4).

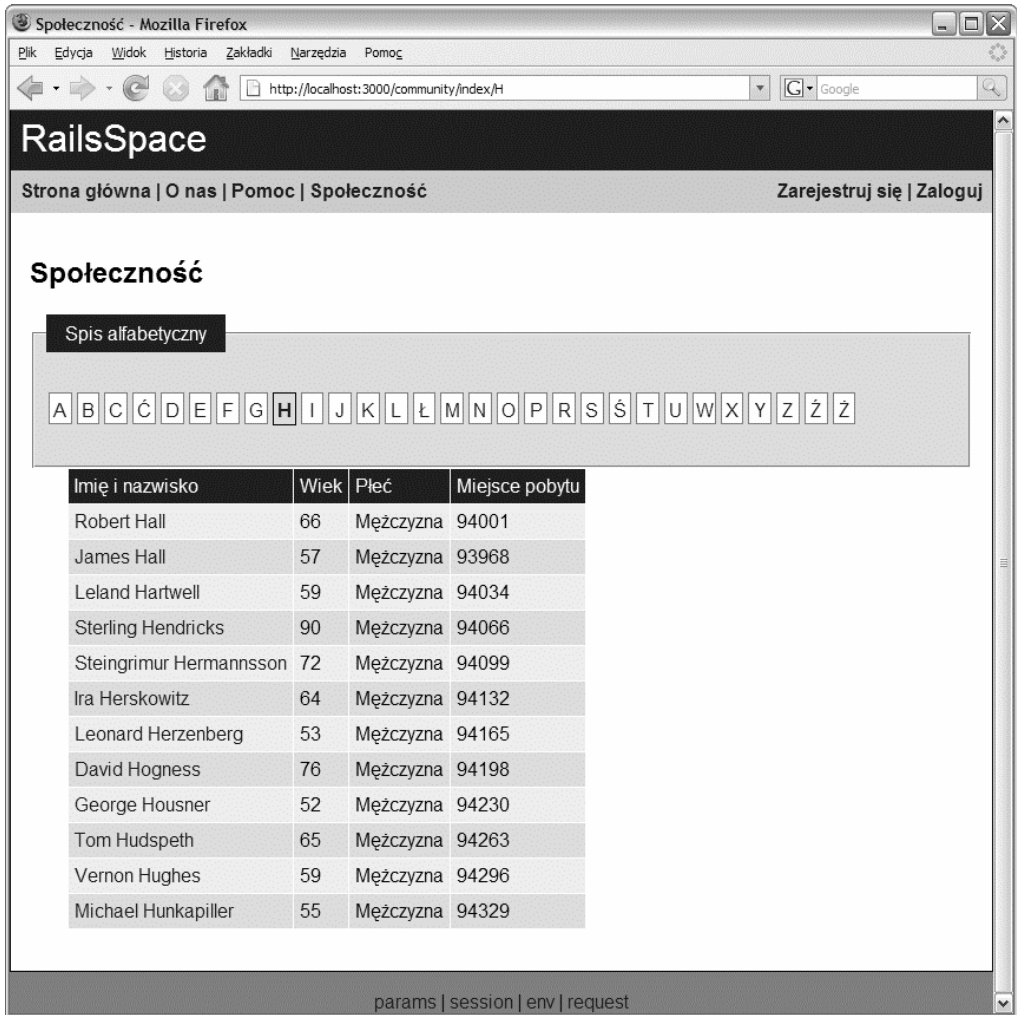
10.4. DOPRACOWYWANIE WYNIKÓW

W tej chwili tabela świetnie wyświetla wyniki. Jest jednak kilka powszechnie stosowanych usprawnień, które poprawiają wygląd wyników, gdy do wyświetlenia jest stosunkowo sporo użytkowników. W tym podrozdziale pokażemy, jak łatwo w Rails można utworzyć *paginację* wyników, dzięki czemu odnośniki do list użytkowników będą wygodnie podzielone na mniejsze części. Dodamy również pomocne podsumowanie wyników, wskazujące, jak wiele wyników zostało odnalezionych. Jak możesz się spodziewać, utworzony w tym podrozdziale kod wykorzystamy później podczas implementacji wyszukiwania i przeglądania.

10.4.1. DODAWANIE PAGINACJI⁹

Nasz spis członków społeczności powinien obsługiwać wiele stron wyników, dzięki czemu mimo powiększania się liczby użytkowników RailsSpace, będą one wciąż ładnie wyświetlane. Zamierzamy wyświetlać jedną stronę wyników na raz i umieszczać odnośniki do kolejnych stron. Jest to często stosowany wzorzec przy wyświetlaniu informacji w internecie, więc Rails dostarcza kilka funkcji pomocniczych ułatwiających implementację tej funkcjonalności. W kontrolerze musimy jedynie zastąpić wywołanie `find` wywołaniem funkcji `paginate`. Składnie tych funkcji są bardzo podobne — zmień tylko:

⁹ W Rails 2.0 wycofano funkcję `paginate`. Jest ona dostępna wyłącznie jako plugin — `classic_pagination`. W celu jej zainstalowania należy wpisać `ruby script/plugin install svn://errtheblog.com/svn/plugins/classic_pagination`. Po przeprowadzeniu instalacji konieczne jest ponowne uruchomienie serwera deweloperskiego — *przyjp. tłum.*



RYSUNEK 10.4. Strona po dodaniu stylów do tabeli wyników

LISTING 10.15. `app/controllers/community_controller.rb`

```

specs = Spec.find(:all,
                  :conditions => ["last_name like ?", @initial+'%'],
                  :order => "last_name, first_name")

```

na:

LISTING 10.16. `app/controllers/community_controller.rb`

```
@pages, specs = paginate(:specs,
                          :conditions => ["last_name like ?",
                                          ↳@initial+'%'],
                          :order => "last_name, first_name")
```

W miejsce `:all` funkcja `paginate` przyjmuje symbol reprezentujący nazwę tabeli, ale pozostałe dwie opcje są takie same. (Więcej opcji funkcji `paginate` znajdziesz w API Rails). Podobnie jak `Spec.find`, funkcja `paginate` zwraca listę specyfikacji, ale zwraca też w zmiennej `@pages` listę stron wyników. Zwróć uwagę, że `paginate` zwraca dwuelementową tablicę, więc możemy przypisać wartości obu zmiennym jednocześnie, korzystając ze składni Ruby dla wielokrotnych przypisań:

```
a, b = [1, 2] # a równe 1, b równe 2
```

Nie dręcz się zbytnio, czym jest `@pages`. Przede wszystkim jest ona przesyłana do funkcji `pagination_links` w widoku, co za chwilę uczynimy.

Będziemy paginować wyniki tylko wtedy, gdy zmienna `@pages` będzie istniała, a jej wartość będzie większa niż jeden, dlatego też utworzymy krótką funkcję pomocniczą testującą te warunki:

LISTING 10.17. `app/helpers/application_helper.rb`

```
module ApplicationHelper
  .
  .
  .
  # Zwraca true, jeżeli wyniki powinny być podzielone na strony
  def paginated?
    @pages and @pages.length > 1
  end
end
```

Ponieważ spodziewamy się, że funkcja `paginated?` będzie nam potrzebna w kilku miejscach, umieściliśmy ją w głównym pliku pomocniczym aplikacji.

Pozostało nam tylko umieścić paginowane wyniki na końcu tabeli użytkowników, korzystając ze wspomnianej wyżej funkcji pomocniczej `pagination_links`:

LISTING 10.18. `app/views/community/_user_table.rhtml`

```
<% if @users and not @users.empty? %>
<table class="users" border="0" cellpadding="5" cellspacing="1">
  .
  .
  .
```

```

<% end %>
<% if paginated? %>
<tr>
  <td colspan="4" align="right">
    Strony: <%= pagination_links(@pages, :params => params) %>
  </td>
</tr>
<% end %>
</table>
<% end %>

```

Wykorzystujemy tutaj funkcję `pagination_links`, która przyjmuje zmienną wygenerowaną przez funkcję `paginate` i tworzy odnośniki dla wielu stron, co obrazuje rysunek 10.5.

The screenshot shows a Mozilla Firefox browser window displaying the RailsSpace community page. The page has a dark header with the site name 'RailsSpace' and navigation links. Below the header is a section titled 'Społeczność' with an alphabetical index. The index shows letters A through Z, with 'H' highlighted. Below the index is a table of members with columns for name, age, gender, and location. The table lists 10 members, all male, with ages ranging from 52 to 90. The page is paginated, showing 'Strony: 1 2'. At the bottom, there are links for 'params', 'session', 'env', and 'request'.

Imię i nazwisko	Wiek	Płeć	Miejsce pobytu
James Hall	57	Mężczyzna	93968
Robert Hall	66	Mężczyzna	94001
Leland Hartwell	59	Mężczyzna	94034
Sterling Hendricks	90	Mężczyzna	94066
Steingrimur Hermannsson	72	Mężczyzna	94099
Ira Herskowitz	64	Mężczyzna	94132
Leonard Herzenberg	53	Mężczyzna	94165
David Hogness	76	Mężczyzna	94198
George Housner	52	Mężczyzna	94230
Tom Hudspeth	65	Mężczyzna	94263

RYСУNEK 10.5. Spis alfabetyczny dzielony na strony

Przy okazji — przekazaliśmy do `pagination_links` zmienną `params`, korzystając z `:params => params`, dzięki czemu funkcja będzie mogła wcielić przesłane parametry do tworzonych adresów URL. W tej chwili nie będziemy tego potrzebować, ale przyda się nam w rozdziale 11.

10.4.2. PODSUMOWANIE WYNIKÓW

Często przy zwracaniu wyników wyszukiwania umieszcza się informację o całkowitej liczbie wyników i jeżeli wyniki są paginowane, informacje o tym, które elementy są obecnie wyświetlane. Innymi słowy, chcemy, aby wyświetlana była informacja podobna do „Znaleziono 15 wyników. Wyświetlani są użytkownicy od 1 do 10”. Dodamy plik części implementujący tę funkcjonalność:

LISTING 10.19. `app/views/community/_result_summary.rhtml`

```
<% if @pages %>
<p>
  Znaleziono <%= pluralize(@pages.item_count, "wynik", "wyników") %>.

  <% if paginated? %>
  <% first = @pages.current_page.first_item %>
  <% last = @pages.current_page.last_item %>
  Wyświetlani są użytkownicy <%= first %>&ndash;<%= last %>.
  <% end %>
</p>
<% end %>
```

Następnie odwzorowujemy część w widoku `index`:

LISTING 10.20. `app/views/community/index.rhtml`

```
.
.
.
<%= render :partial => "result_summary" %>
<%= render :partial => "user_table" %>
```

Jak widzimy na podstawie powyższego kodu, zmienna `@pages` zwracana przez funkcję `paginate` ma kilka atrybutów ułatwiających utworzenie takiego podsumowania: `item_count`, który stanowi całkowitą liczbę wyników, oraz `current_page.first_item` i `current_page.last_item`, które stanowią numer pierwszego i ostatniego elementu na stronie. Wyniki wyglądają teraz tak, jak to zapowiadaliśmy — spójrz na rysunek 10.1.

Powinniśmy również zwrócić uwagę, że w pliku części z podsumowaniem wyników wykorzystujemy również wygodną funkcję pomocniczą Rails — `pluralize`¹⁰:

¹⁰ Funkcja `pluralize` nie jest domyślnie dostępna w sesji konsoli, więc musieliśmy ją zawrzeć w sposób jawny. Dowiedzieliśmy się, który moduł należy załadować, przeglądając API Rails


```
> ruby script/console
Loading development environment.
>> include ActionView::Helpers::TextHelper
=> Object
>> pluralize(0, "box")
=> "0 boxes"
>> pluralize(1, "box")
=> "1 box"
>> pluralize(2, "box")
=> "2 boxes"
>> pluralize(2, "box", "boxen")
=> "2 boxen"
```

Funkcja `pluralize` wykorzystuje *inflektor* Rails (wspomniany w punkcie 3.1.3) do określenia odpowiedniej formy liczby mnogiej danego łańcucha na podstawie pierwszego argumentu, który określa, ile jest obiektów. Jeżeli chcesz przesłonić inflektor, podaj trzeci argument. W związku z tym w Rails nie ma wymówki dla tekstów w rodzaju „Znaleziono 1 wynik(ów)” czy, nie daj Boże, „Znaleziono 1 wyników”¹¹.

¹¹ Nonsensowne komunikaty `1 tests, 1 assertions`, jakie mogłeś zauważyć w wyjściu testów, są winą frameworku `Ruby Test::Unit`, a nie Rails.