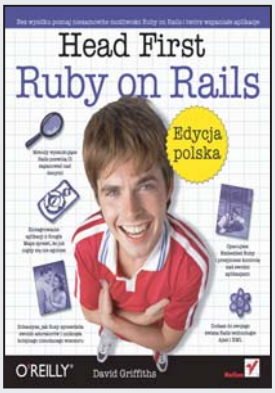
Head First Ruby on Rails. Edycja polska



Autor: David Griffiths
Tłumaczenie: Anna Trojan
ISBN: 978-83-246-2130-9
Tytuł oryginału: Head First Rails: A learner's companion to Ruby on Rails
Format: 200×234, stron: 470

Bez wysiłku poznaj niesamowite możliwości Ruby on Rails i twórz wspaniałe aplikacje

Jeśli chcesz szybko i sprawnie budować internetowe aplikacje bazodanowe, warto, abyś poznał niezwykłe możliwości Rails. Ta wyjątkowa platforma programowania pozwala tworzyć w pełni funkcjonalne aplikacje z wykorzystaniem języka Ruby. Jej wielką zaletą jest to, że wszystkie zmiany wprowadzane do aplikacji można zobaczyć natychmiast po ich zapisaniu i odświeżeniu strony w przeglądarce. Dzieje się tak dzięki zastosowaniu języka Ruby, ponieważ kod w tym języku nie musi być kompilowany.

Książkę „Head First Ruby on Rails. Edycja polska" napisano w oparciu o najnowsze, skuteczne techniki ułatwiające zrozumienie i przyswajanie wiedzy. Dzięki temu szybko i bez trudności nauczysz się tworzyć interaktywne aplikacje internetowe za pomocą tej platformy. Dowiesz się, na czym polega współpraca z bazą danych, integracja z Ajaksem i XML oraz dynamiczne wykreślanie danych. Ponieważ to obrazy najlepiej przemawiają do Twojego umysłu, książka ta została bogato zilustrowana – abyś jak najszybciej poznał możliwości Rails i natychmiast zaczął wykorzystywać je w praktyce.

- Język Ruby
- Tworzenie tabel
- Wykonywanie migracji bazy danych
- Kod modelu, widoku i kontrolera
- Tworzenie formularza
- Sprawdzanie poprawności danych
- Zapobieganie błędom
- Dołączanie bibliotek Ajaksa
- Udostępnianie aplikacji użytkownikom

Wykorzystaj najnowsze metody uczenia się i szybko opanuj Ruby on Rails!

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

Spis treści (skrócony)

Spis treści (z prawdziwego zdarzenia)

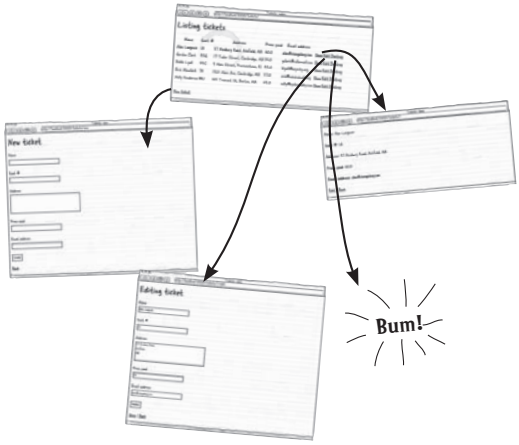
Wprowadzenie

Przestawienie swojego mózgu na Rails. A zatem tutaj *Ty* próbujesz się czegoś *nauczyć*, podczas gdy Twój *mózg* próbuje oddać Ci przysługę, starając się, by to, czego się nauczyłeś, nie zostało *zapamiętane*. Twój mózg myśli sobie: „Lepiej zostawić miejsce na ważniejsze rzeczy, takie jak to, których dzikich zwierząt należy unikać i czy jazda na snowboardzie nago jest złym pomysłem". *Jak* zatem możesz zmusić swój mózg do zaakceptowania przekonania, że Twoje życie uzależnione jest od poznania Rails?

Początki

1 Naprawdę szybkie Rails

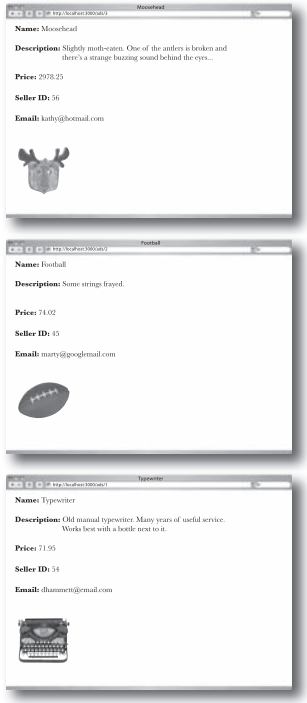
Chcesz szybko zacząć pisać aplikacje internetowe? Powinieneś zatem poznać **Rails**. Rails to **najfajniejsza** i **najszybsza platforma programowania**, jaka istnieje. Pozwala tworzyć **w pełni funkcjonalne aplikacje internetowe** szybciej, niż kiedykolwiek wydawało się to możliwe. Początki są łatwe — wystarczy **zainstalować Rails** i zacząć przewracać strony książki. Zanim się zorientujesz, **o lata świetlne wyprzedzisz swoich konkurentów**!

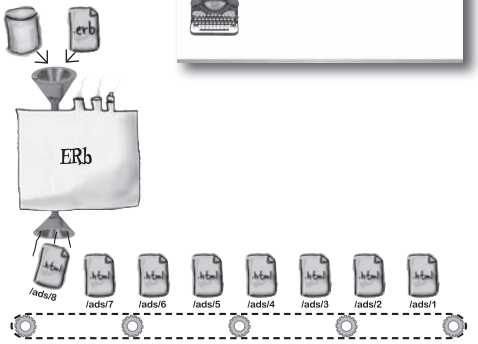


Poza rusztowaniem

2 Aplikacje Rails — stworzone, by nimi zarządzać

Co tak naprawdę dzieje się w Rails? Widziałeś już, jak **rusztowania** generują mnóstwo kodu i pomagają pisać aplikacje internetowe w sposób niesamowicie szybki, ale co, jeśli pragniesz czegoś innego? W tym rozdziale zobaczysz, jak można ***przejąć kontrolę*** nad programowaniem w Rails, i będziesz miał okazję zajrzeć pod maskę tej platformy. Przekonasz się, w jaki sposób Rails decyduje o tym, który **kod** należy wykonać, jak **dane** wczytywane są z bazy danych i jak generowane są **strony internetowe**. Pod koniec rozdziału będziesz w stanie publikować dane tak, jak ***sam*** zechcesz.

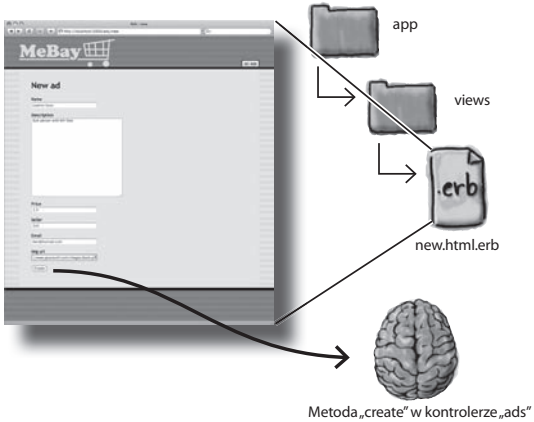


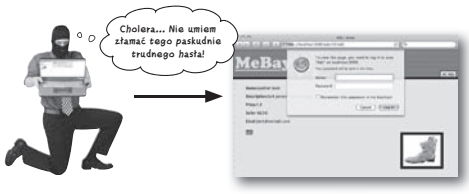


Wstawianie, uaktualnianie i usuwanie

3 Wszystko się zmienia

Zmiana to część życia — szczególnie w przypadku danych. Na razie widziałeś, jak można szybko wyczarować aplikację Rails dzięki rusztowaniu, a także jak napisać własny kod w celu publikacji danych z bazy. Ale co zrobić, kiedy chcemy, by użytkownicy mogli edytować dane w zaplanowany *przez nas* sposób? Co jeśli rusztowanie nie robi tego, co chcemy *my*? W tym rozdziale nauczysz się **wstawiać**, **uaktualniać** i **usuwać** dane dokładnie tak, jak tego chcesz. A przy okazji zobaczysz również, jak tak *naprawdę* działa Rails, i być może nauczysz się również czegoś o bezpieczeństwie.



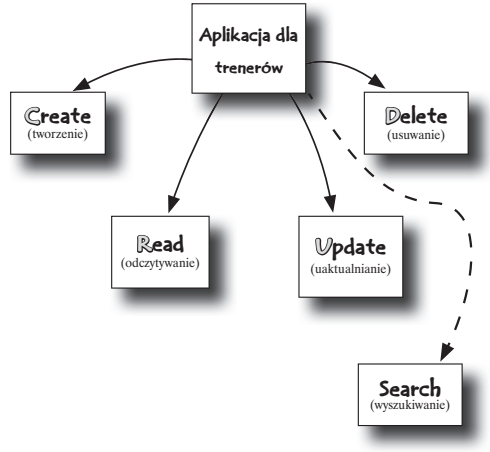


Wyszukiwanie w bazie danych

4

Prawda czy konsekwencje?

Każda decyzja ma swoje konsekwencje. W Rails wiedza o tym, jak podejmować **dobre decyzje**, może zaoszczędzić Ci zarówno czasu, jak i wysiłku. W tym rozdziale przyjrzymy się, jak **wymagania użytkownika** wpływają na wybory, jakich dokonujesz, już **od samego początku** tworzenia Twojej aplikacji. Czy powinieneś użyć rusztowania, czy lepiej zmodyfikować wygenerowany kod? Czy powinieneś tworzyć wszystko od nowa? Bez względu na wybór, kiedy nadejdzie pora dalszego dostosowania aplikacji do własnych potrzeb, będziesz musiał nauczyć się obsługi **wyszukiwania w bazie danych** — ***dostępu do danych*** w sposób, który ma sens zarówno z Twojego punktu widzenia, jak i z punktu widzenia **potrzeb Twoich użytkowników**.



Sprawdzanie poprawności danych

5 Zapobieganie błędom

Każdy popełnia błędy… ale wielu z nich można zapobiec! Nawet przy najlepszych chęciach użytkownicy nadal będą wprowadzać niepoprawne dane do Twojej aplikacji internetowej **i to Ty będziesz musiał poradzić sobie z konsekwencjami**. Wyobraź sobie, co by było, gdyby istniała jakaś metoda **zapobiegania występowaniu błędów**. Do tego właśnie służą **walidatory**. Czytaj dalej, a pokażemy Ci, jak można dodać **sprytne sprawdzanie błędów w Rails** do Twojej aplikacji internetowej, tak byś mógł **przejąć kontrolę** nad tym, jakie dane są dozwolone, a jakich należy się wystrzegać.

Tworzenie połączeń

6 Łączenie wszystkiego razem

Niektóre rzeczy lepsze są razem niż osobno. Posmakowałeś zatem niektórych **kluczowych składników Rails**. Tworzyłeś całe aplikacje internetowe, a także brałeś to, co wygenerowała platforma Rails, i **przystosowywałeś** do swoich potrzeb. W prawdziwym świecie **życie może jednak być bardziej skomplikowane**. Czytaj dalej... czas zacząć budować **wielofunkcyjne strony internetowe**! I nie tylko to — czas zacząć sobie radzić ze **skomplikowanymi powiązaniami między danymi**, a także przejąć kontrolę nad danymi, pisząc **własne walidatory**.

Ajax

7 Ograniczanie ruchu

Każdy chce uzyskać z życia jak najwięcej… podobnie z aplikacji. Bez względu na to, jak jesteś dobry w obsłudze Rails, czasami tradycyjne aplikacje internetowe sobie nie radzą. Bywa, że użytkownicy pragną czegoś bardziej **dynamicznego**, czegoś, co odpowiada na wszystkie ich kaprysy. Ajax pozwala tworzyć **szybkie aplikacje internetowe z doskonałym czasem reakcji**, zaprojektowane tak, by użytkownik mógł **czerpać z Internetu jak najwięcej**. Rails ma wbudowany własny zestaw bibliotek Ajaksa, które tylko czekają na to, aż ich użyjesz! Pora **szybko i łatwo dodać do aplikacji fantastyczne możliwości oferowane przez technologię Ajax** i zachwycić jeszcze większą liczbę użytkowników.

XML i różne reprezentacje

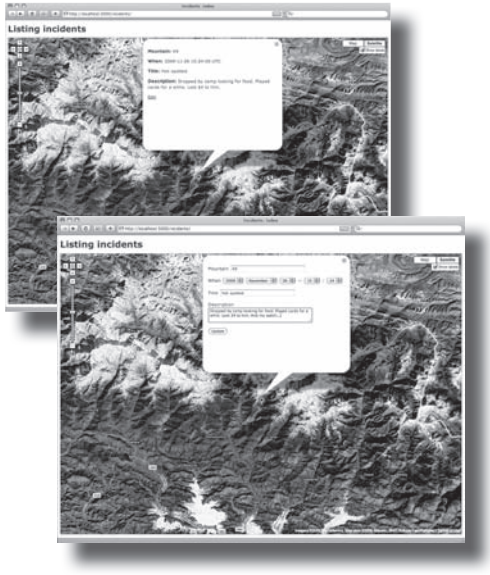
8 Wszystko wygląda teraz inaczej...

Nie da się zawsze wszystkich zadowolić. A może jednak? Dotychczas widzieliśmy, jak można wykorzystać Rails do szybkiego i łatwego tworzenia aplikacji internetowych, które **idealnie pasują do pewnego zbioru wymagań**. Co jednak zrobić, kiedy **pojawiają się inne wymagania**? Co powinniśmy zrobić, jeśli niektóre osoby chcą otrzymać **proste strony internetowe**, inne interesuje **mashup z aplikacji firmy Google**, a jeszcze inne chcą, by aplikacja była dostępna w czytniku **kanałów RSS**? W tym rozdziale będziemy tworzyć **różne reprezentacje** tych samych danych, co da nam **maksymalną elastyczność** przy **minimalnym wysiłku**.

Architektura REST i Ajax

9 Kolejne kroki

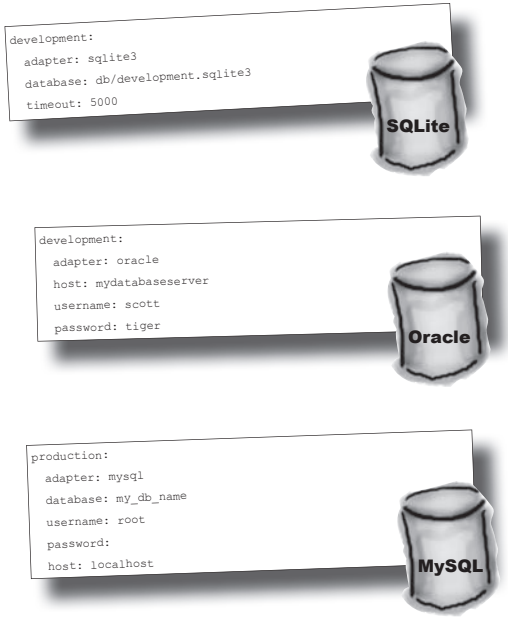
Czas skonsolidować umiejętności w zakresie korzystania z aplikacji typu mashup. Dotychczas widzieliśmy, jak w celu pokazania danych geograficznych można dodać do naszych aplikacji mapy z serwisu **Google Maps**. Co jednak, jeśli chcemy **rozszerzyć istniejącą już funkcjonalność**? Czytaj dalej, a przekonasz się, jak można wzbogacić aplikacje typu **mashup** o **bardziej zaawansowane cudeńka oparte na Ajaksie**. Co więcej, przy okazji nauczysz się też nieco o architekturze **REST**.



Prawdziwe aplikacje

10 Rails w świecie rzeczywistym

Nauczyłeś się już wiele o Ruby on Rails. By jednak zastosować tę wiedzę **w prawdziwym świecie**, będziesz musiał zastanowić się nad kilkoma sprawami. W jaki sposób połączyć aplikację **z inną bazą danych**? Jak **testuje** się aplikacje Rails? Jak można wydobyć maksimum możliwości z Rails oraz **języka Ruby**? I skąd można dowiedzieć się o **najświeższych nowościach** w świecie Rails? Czytaj dalej, a pokażemy Ci kierunek, dzięki któremu jeszcze bardziej rozwiniesz swoje umiejętności programistyczne.



Naprawdę szybkie Rails

Chcesz szybko zacząć pisać aplikacje internetowe? Powinieneś zatem poznać **Rails**. Rails to **najfajniejsza** i **najszybsza platforma programowania**, jaka istnieje. Pozwala tworzyć **w pełni funkcjonalne aplikacje internetowe** szybciej, niż kiedykolwiek wydawało się to możliwe. Początki są łatwe — wystarczy **zainstalować Rails** i zacząć przewracać strony książki. Zanim się zorientujesz, **o lata świetlne wyprzedzisz swoich konkurentów**!

Piątek, godzina 9 rano

Pierwszy e-mail, jaki otwierasz, pochodzi od przyjaciela, który jest w opałach:

Hej — jak się masz?

Potrzebna mi *wielka* przysługa! Pamiętasz tę aplikację do sprzedaży biletów, nad którą — jak mówiłem — pracowaliśmy? Nie wygląda to za dobrze. Siedzimy nad tym od tygodni! Nasz zespół naprawdę sobie nie radzi.

Czy myślisz, że mógłbyś utworzyć tę aplikację dla nas?

Potrzebna nam strona internetowa, która jest w stanie:

- wyświetlić wszystkie sprzedane bilety,
- utworzyć nową transakcję sprzedaży biletu,
- wczytać i wyświetlić pojedynczy bilet,
- uaktualnić szczegóły sprzedaży,
- usunąć transakcję sprzedaży biletu.

Wiem — wydaje się, że to gigantyczna liczba funkcji, ale szef mówi, że to minimum opcji, jakich potrzebują; dobrze wiesz, że z tym facetem trudno się kłócić! A oto struktura danych:

Ticket (bilet):

name — imię i nazwisko kupującego (łańcuch znaków)

seat_id_seq — numer miejsca, na przykład E14 (łańcuch znaków)

address — adres kupującego (długi łańcuch znaków)

price_paid — cena sprzedaży biletu (liczba dziesiętna)

email_address — adres e-mail kupującego (łańcuch znaków)

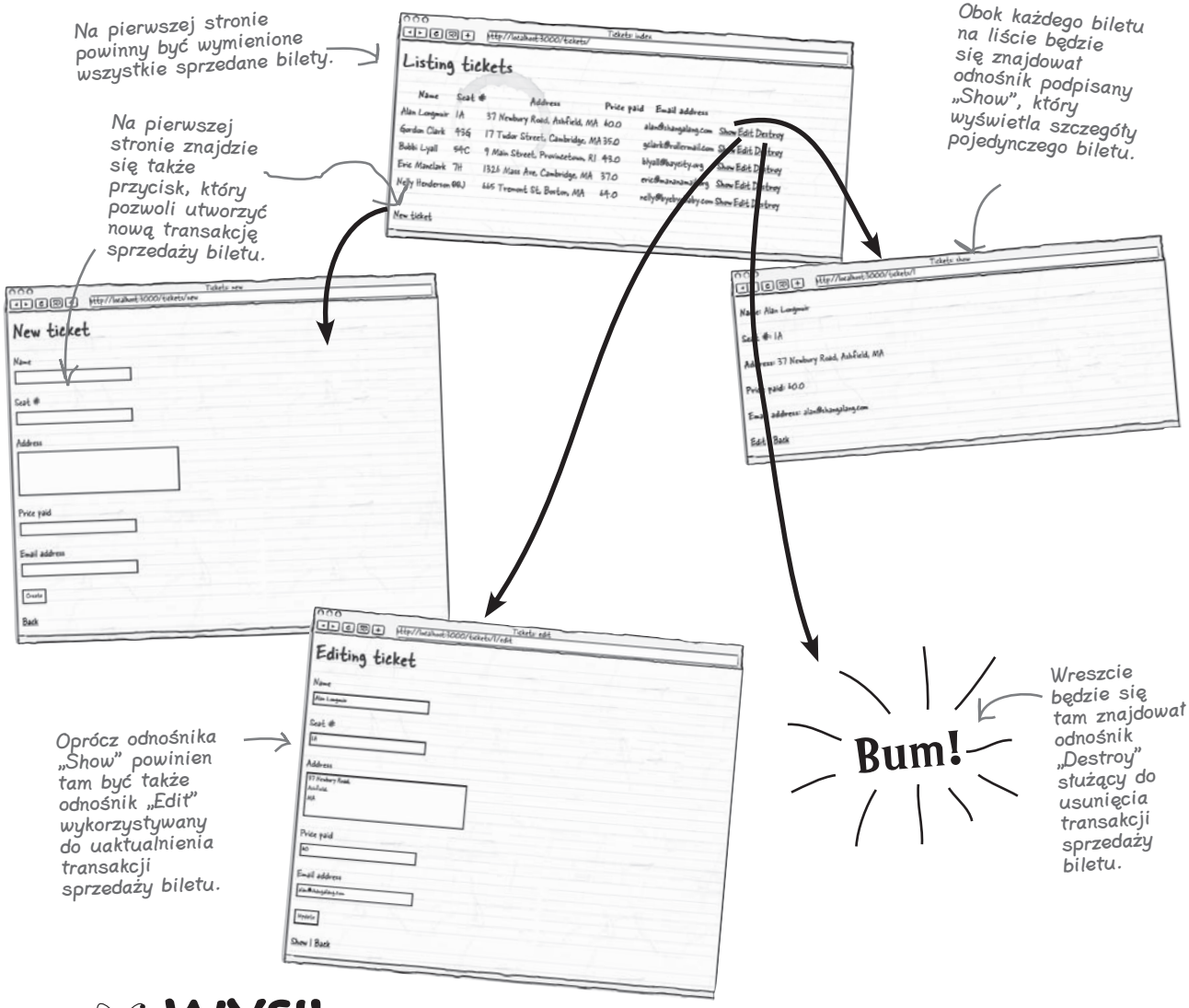
Załączam szkice stron internetowych, żebyś wiedział, do czego zmierzamy.

Eeee, potrzebujemy tego na poniedziałek, inaczej polecę ze stołka. Pomocy!

System jest zaprojektowany do użycia przez personel zatrudniony w hali koncertowej. Baza danych będzie resetowana dla każdego koncertu, więc wystarczające będzie zapisanie szczegółów jednego koncertu naraz. Myślisz, że dasz radę pomóc?

Aplikacja musi robić wiele rzeczy

Poniżej znajdują się szkice stron. Czy pasują one do wymagań tego systemu?



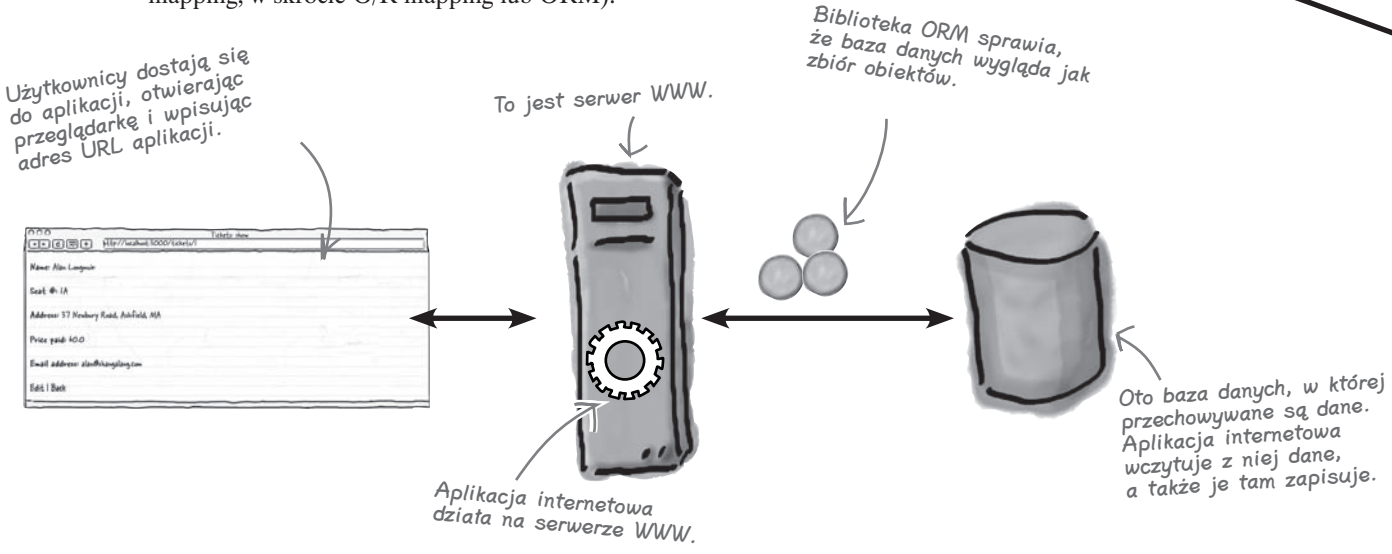
WYSIL SZARE KOMÓRKI

Jakiego rodzaju oprogramowania będziesz potrzebował do utworzenia i uruchomienia aplikacji?

Co jest potrzebne aplikacji?

By uruchomić aplikację na serwerze hali koncertowej, potrzebujemy kilku elementów. Potrzebne nam są:

1. **Platforma aplikacji.**
   Niezbędny nam będzie zbiór napisanego wcześniej kodu, który będzie stanowił podstawę aplikacji internetowej.
2. **System bazy danych.**
   Potrzebna nam będzie jakaś baza danych, w której przechowamy dane.
3. **Serwer WWW.**
   Musimy gdzieś uruchomić aplikację.
4. **Biblioteka mapowania relacyjno-obiektowego.**
   By ułatwić dostęp do bazy danych, większość aplikacji internetowych wykorzystuje obecnie do przekształcenia rekordów bazy danych w obiekty bibliotekę mapowania relacyjno-obiektowego (ang. object-relational mapping, w skrócie O/R mapping lub ORM).



W czym zatem pomoże nam Rails?

Bez względu na język, w jakim tworzysz aplikację, najprawdopodobniej będziesz potrzebował wszystkich trzech elementów. Jedną z najlepszych rzeczy w Rails jest to, że platforma ta zawiera *całe* oprogramowanie, jakiego będziesz potrzebował — ***dołączone za darmo***.

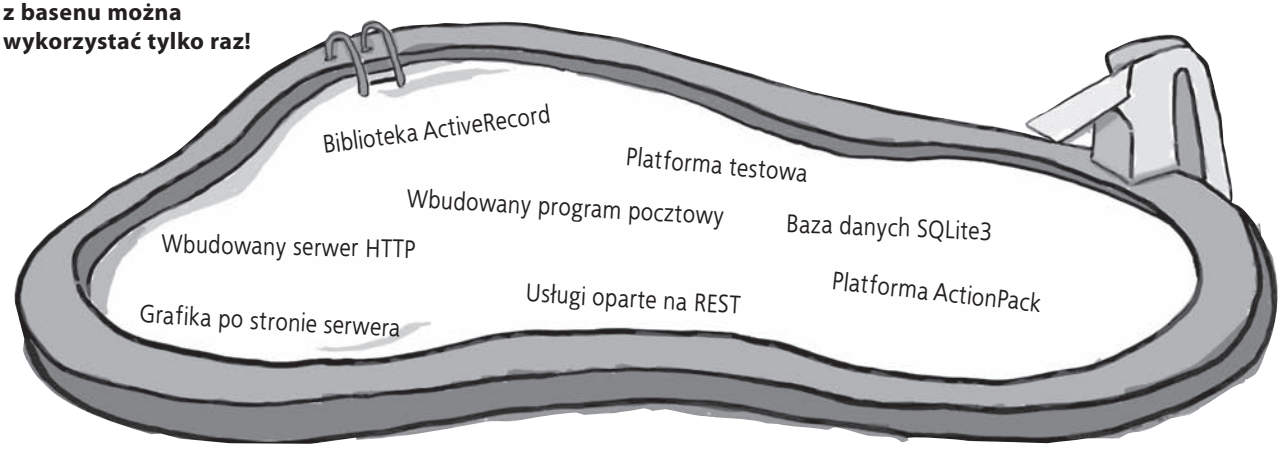
Zobaczmy, jak to działa.

Łamigłówka

Platforma Rails ma wiele wbudowanych możliwości. Twoje zadanie polega na odgadnięciu, które z elementów widocznych w basenie potrzebne nam będą w naszej aplikacji internetowej. Później elementy te należy umieścić w pustych wierszach poniżej. Nie wszystkie elementy będą nam potrzebne.

1. .....
2. .....
3. .....
4. .....

Uwaga: każdy element z basenu można wykorzystać tylko raz!



Rails służy do tworzenia aplikacji bazodanowych, takich jak system sprzedaży biletów

Sercem wielu aplikacji jest baza danych. Podstawowym celem istnienia tych aplikacji jest umożliwienie użytkownikom dostępu do zawartości bazy danych oraz edycji tych danych ***bez konieczności*** bezpośredniego korzystania z języka SQL.

Jakie problemy należy rozwiązać przy połączeniu bazy danych z aplikacją internetową?

Aplikacja internetowa musi zezwalać użytkownikowi na dostęp i modyfikację danych, dlatego Rails zawiera **platformę aplikacji** o nazwie **ActionPack**, która wspomaga generowanie interaktywnych stron internetowych współdziałających z bazami danych.

Po drugie, aplikacje internetowe muszą być uruchamiane na **serwerze WWW**, który będzie w stanie wyświetlać te strony, dlatego serwer taki wbudowany jest w Rails.

Po trzecie, niezbędna jest **baza danych**. Rails tworzy aplikacje, które skonfigurowane są do pracy ze zintegrowaną bazą danych **SQLite3**.

Po czwarte, niezbędna jest **biblioteka mapowania relacyjno-obiektowego**; Rails udostępnia takową pod nazwą **ActiveRecord**. Dzięki temu baza danych wygląda jak zbiór prostych *obiektów języka Ruby*.

Oprócz tych narzędzi Rails zawiera również wiele **skryptów**, które wspomagają zarządzanie aplikacją. Kiedy będziesz tworzył aplikację internetową opartą na bazie danych, wkrótce przekonasz się, że

Rails daje Ci wszystko, czego potrzebujesz.

Łamigłówka: Rozwiązanie

Platforma Rails ma wiele wbudowanych możliwości. Twoje zadanie polega na odgadnięciu, które z elementów widocznych w basenie potrzebne nam będą w naszej aplikacji internetowej. Później elementy te należy umieścić w pustych wierszach poniżej. Nie wszystkie elementy będą nam potrzebne.

Platforma ActionPack

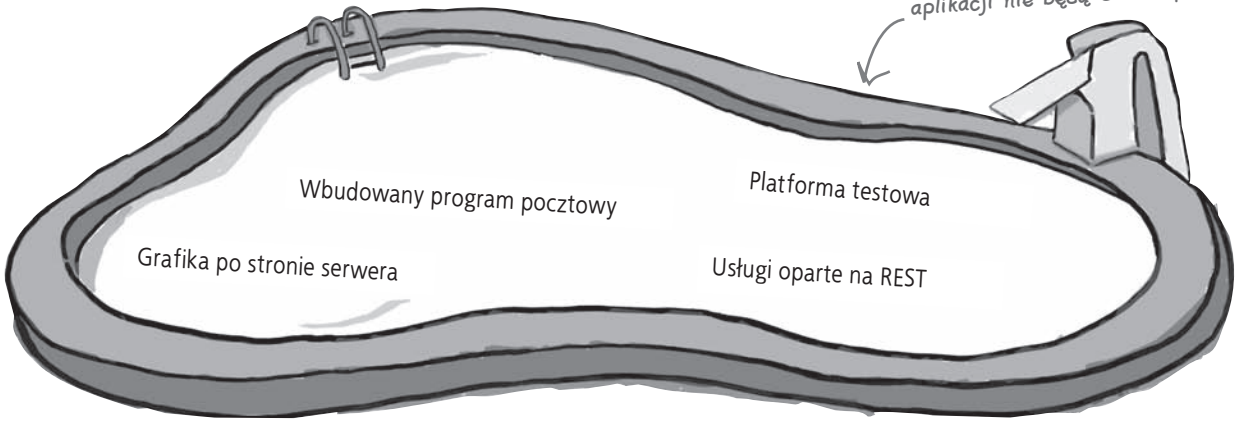
Baza danych SQLite3

Wbudowany serwer HTTP

Biblioteka ActiveRecord

W niektórych systemach operacyjnych będziesz musiał zainstalować ją niezależnie od Rails.

Rails daje Ci także wszystkie te elementy, jednak w przypadku tej aplikacji nie będą Ci one potrzebne.



Nową aplikację tworzy się za pomocą polecenia rails

Jak zatem zacząć pracę z Rails?

Utworzenie nowej aplikacji internetowej w Rails jest tak naprawdę bardzo proste. Wystarczy otworzyć okno wiersza polecenia lub terminala i wpisać do niego **`rails tickets`**, gdzie `tickets` to nazwa aplikacji, którą chcesz utworzyć.


```
Plik Edycja Okno Pomoc
> rails tickets
```

Wpisz po prostu „rails tickets" w wierszu poleceń.

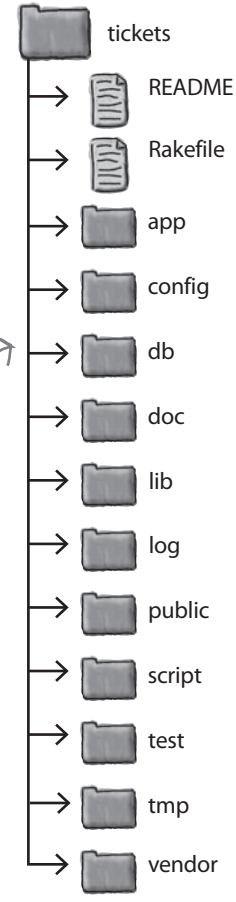
Co to robi?

Wpisanie `rails tickets` w sprytny sposób generuje aplikację internetową w nowym folderze o nazwie *tickets*. Co więcej, wewnątrz folderu *tickets* Rails generuje całe mnóstwo dalszych folderów i plików, które tworzą podstawową strukturę nowej aplikacji.

Oznacza to, że tak naprawdę utworzyłeś całą podstawową aplikację za pomocą jednego krótkiego polecenia.

Rails generuje mnóstwo plików i folderów, ale nie ma się tym co martwić.

Wszystkie mają swoje uzasadnienie i do końca książki zrozumiesz, do czego służą.



Rails generuje dla Ciebie cały zbiór plików i folderów za pomocą jednego polecenia. To jest struktura całej aplikacji internetowej.

Jazda próbna

Ponieważ utworzona przed chwilą aplikacja jest aplikacją *internetową*, by zobaczyć, jak działa, będziesz musiał uruchomić wbudowany serwer WWW.

W wierszu poleceń czy terminalu wejdź do folderu *tickets* i wpisz **`ruby script/server`**.

To jest konsola. Wchodzisz do niej za pomocą wiersza poleceń w systemie Windows lub terminala w systemach Linux lub Mac.

Wejdź do folderu aplikacji...

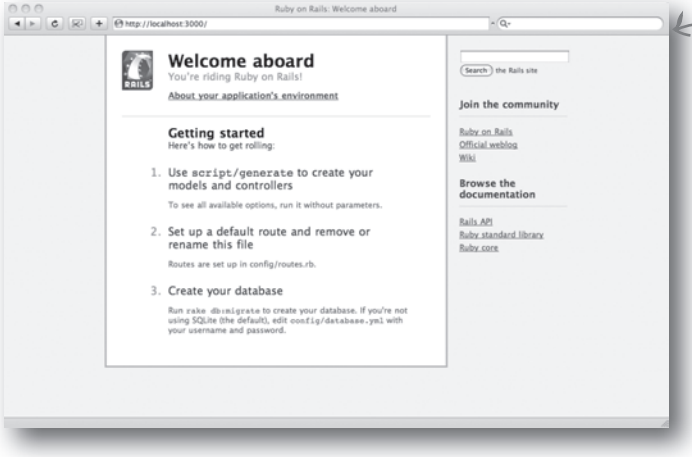
...i uruchom serwer WWW.

```
Plik Edycja Okno Pomoc
> cd tickets
> ruby script/server
```

Na ekranie pojawi się kilka poleceń, które potwierdzają, że serwer działa. Teraz możesz zobaczyć domyślną stronę główną, otwierając w przeglądarce adres:

`http://localhost:3000/`

To jest domyślna strona główna serwera WWW.



Ciekawostki

Rails domyślnie uruchamia serwer WWW na porcie 3000. Jeśli chcesz użyć innego portu, takiego jak 8000, wykonaj polecenie:

`ruby script/server -p 8000`

Teraz do domyślnej aplikacji trzeba dodać własny kod

Rails od razu tworzy podstawową strukturę aplikacji, jednak będziesz musiał dodać kod robiący to, czego chcesz **Ty**. Każda aplikacja jest inna, ale czy Rails ma jakieś narzędzia czy sztuczki, które ułatwiają tworzenie własnego kodu?

Faktycznie, tak właśnie jest. Czy zauważyłeś, że platforma Rails utworzyła dla Ciebie całą strukturę plików, prawie jakby wiedziała, czego będziesz potrzebować? Dzieje się tak, ponieważ aplikacje Rails mają bardzo silne konwencje nazewnictwa.

Aplikacje Rails zawsze przestrzegają konwencji

Wszystkie aplikacje Rails mają tę samą podstawową strukturę plików i wykorzystują spójne nazewnictwo poszczególnych elementów. Dzięki temu aplikacje są łatwiejsze do zrozumienia, jednak równocześnie oznacza to, że wbudowane narzędzia Rails będą rozumieć, jak działa Twoja aplikacja.

Dlaczego jest to tak istotne? Skoro narzędzia wiedzą, jaką strukturę ma aplikacja, można ich użyć do zautomatyzowania wielu zadań programistycznych. W ten sposób Rails może wykorzystać konwencje do wygenerowania dla Ciebie kodu bez konieczności konfiguracji aplikacji. Innymi słowy, Rails przedkłada ***konwencję nad konfigurację***.

Przyjrzyjmy się jednemu z narzędzi Rails o największych możliwościach — rusztowaniu.

Zasada Rails:

Konwencja ważniejsza od konfiguracji

Nie istnieją głupie pytania

P: Ciągle piszecie o „Ruby” albo o „Rails”. Jaka jest między nimi różnica?

O: Ruby to język programowania. Rails to zbiór skryptów języka Ruby. Dlatego serwer WWW, platforma aplikacji ActionPack, a także wbudowane skrypty narzędzi to wszystko skrypty języka Ruby, które są jednocześnie częścią Rails.

P: W jaki sposób mogę zmodyfikować stronę główną mojej nowej witryny?

O: Zerknij do kodu HTML znajdującego się w pliku `index.html` w katalogu `public` aplikacji. Katalog `public` zawiera całą statyczną zawartość aplikacji.

P: A co jeśli chcę użyć innego serwera WWW? Czy mogę to zrobić?

O: W czasie tworzenia aplikacji używanie wbudowanego serwera WWW ma sens. Jeśli jednak chcesz wdrożyć gotową wersję aplikacji na innym serwerze WWW, możesz to zrobić.

P: Czy ma znaczenie, w którym folderze jestem, kiedy wykonuję polecenie `ruby script/server`?

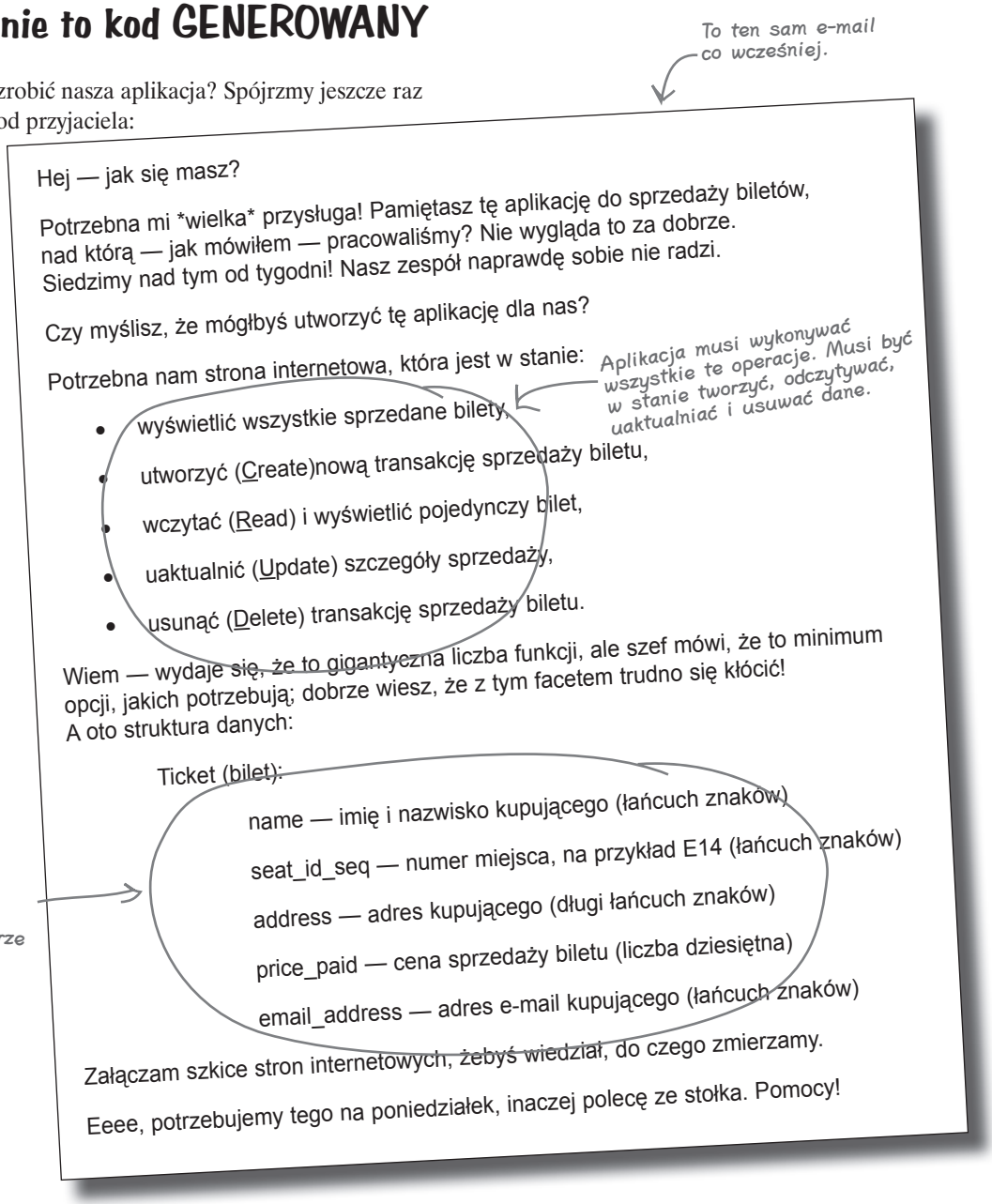
O: Oczywiście, że tak. Musisz znajdować się w folderze zawierającym aplikację.

P: Co kompiluje mój kod?

O: Ruby jest językiem interpretowanym, jak JavaScript. Oznacza to, że kompilacja nie jest potrzebna. Możesz po prostu zmienić kod i natychmiast go wykonać.

Rusztowanie to kod GENEROWANY

Co zatem musi zrobić nasza aplikacja? Spójrzmy jeszcze raz do wiadomości od przyjaciela:



Musimy zatem utworzyć strony internetowe, które pozwolą nam ***tworzyć*** (*Create*), ***odczytywać*** (*Read*), ***uaktualniać*** (*Update*) i ***usuwać*** (*Delete*) bilety. Ponieważ pierwsze litery tych operacji w języku angielskim to **C**, **R**, **U** i **D**, są one znane jako ***operacje CRUD***. W aplikacjach opartych na bazach danych operacje te wykonywane są stosunkowo często — tak często, że Rails udostępnia sposoby szybkiego generowania całego kodu i wszystkich stron, jakie będą potrzebne. Wszystko to wykonywane jest za pomocą ***rusztowania*** (ang. *scaffolding*).

Magnesiki z kodem

Istnieje proste polecenie, które można wydać w konsoli w celu wygenerowania kodu rusztowania. Zobacz, czy będziesz w stanie ułożyć magnesiki tak, by uzupełnić to polecenie.

ruby script/generate ..... **ticket name:**.....

.....:..... .....:.....

.....:..... .....:.....

seat_id_seq

string

scaffold

price_paid

decimal

string

text

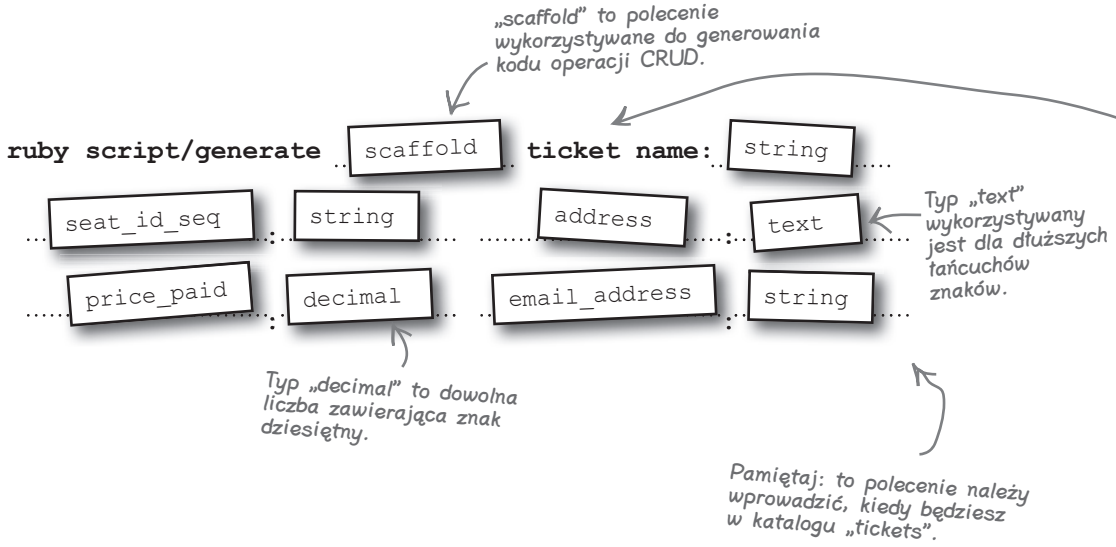
address

email_address

string

Magnesiki z kodem: Rozwiązanie

Istnieje proste polecenie, które można wydać w konsoli w celu wygenerowania kodu rusztowania. Zobacz, czy będziesz w stanie ułożyć magnesiki tak, by uzupełnić to polecenie.



Co zatem robi polecenie scaffold?

Polecenie `scaffold` tworzy kod, który pozwala użytkownikowi tworzyć, odczytywać, uaktualniać oraz usuwać dane z bazy.

Kiedy masz aplikację opartą na bazie danych, która potrzebuje tworzyć, odczytywać, uaktualniać i usuwać dane z bazy, rusztowanie może oszczędzić Ci sporo czasu i wysiłku.

Wpisz polecenie `scaffold` dla tabeli `ticket` do konsoli i zobacz, co się stanie:

Zrób tak!

```
Plik Edycja Okno Pomoc
> ruby script/generate scaffold ticket name:string
seat_id_seq:string address:text price_paid:decimal
email_address:string
```

Jazda próbna

Teraz pora sprawdzić, czy aplikacja naprawdę działa. By zobaczyć nową stronę z biletami, wpisz w przeglądarce adres:

```
http://localhost:3000/tickets
```

Odpowiada to nazwie podanej w poleceniu „scaffold". Widzisz, jak platforma Rails sama zrobiła z tego liczbę mnogą?

Hm... to zdecydowanie nie wygląda dobrze.

Co zatem poszło nie tak? Pomimo że kod rusztowania został wygenerowany poprawnie, serwer WWW wyświetla błąd. Wszystko, co otrzymujemy, to komunikaty o błędach.

WYSIL SZARE KOMÓRKI

Zastanów się nad komunikatem o błędzie widocznym w przeglądarce. Jak myślisz, dlaczego aplikacja nie działa?

W bazie danych nie ma jeszcze tabel!

Aplikacja powinna wyświetlić pustą listę sprzedanych biletów, ale tak nie zrobiła. **Dlaczego nie?** Powinna była wczytać listę z tabeli bazy danych o nazwie **`tickets`**, jednak nie utworzyliśmy jeszcze żadnych tabel.

Czy powinniśmy połączyć się z bazą danych i utworzyć tabelę? W końcu baza danych znajduje się tu, w aplikacji. Ale właściwie dlaczego mamy to robić? Przecież przekazaliśmy Rails wystarczająco dużo informacji, by platforma ta utworzyła tabelę za nas. Spójrzmy raz jeszcze na polecenie `scaffold`:

Uwaga: rusztowanie nosi nazwę „ticket" (liczba pojedyncza), natomiast tabela będzie się nazywała „tickets" (liczba mnoga).

```
Plik Edycja Okno Pomoc
> ruby script/generate scaffold ticket name:string
seat_id_seq:string address:text price_paid:decimal
email_address:string
```

tickets	
name	string
seat_id_seq	string
address	text
price_paid	decimal
email_address	string

Kiedy wykonywaliśmy polecenie `scaffold`, przekazaliśmy Rails wystarczającą ilość informacji dotyczących struktury danych, a w Rails istnieje ważna zasada: ***Nie powtarzaj się***. Jeśli powiesz coś Rails raz, nie powinieneś musieć tego powtarzać.

Jak zatem zmusić Rails do utworzenia tabeli?

Ciekawostki

Rails ma wbudowaną bazę danych SQLite3. Gdzie ona zatem jest?

Baza danych znajduje się w folderze db, w pliku development.sqlite3.

Zasada Rails:

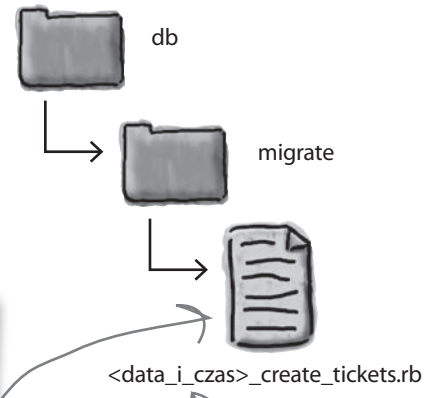
Nie powtarzaj się

Znajomi programiści mogą nazywać tę zasadę „DRY", od pierwszych liter jej angielskiego odpowiednika („Don't Repeat Yourself").

Tabelę tworzy się dzięki wykonaniu migracji

Kiedy platforma Rails wygenerowała rusztowanie, utworzyła również niewielki skrypt w języku Ruby, noszący nazwę **migracja** (ang. *migration*) i służący do tworzenia tabeli. Migracja to skrypt zmieniający strukturę dołączonej bazy danych.

Zajrzyj do folderu *db/migrate*. Powinieneś znaleźć tam plik o nazwie *<data_i_czas>_create_tickets.rb*, gdzie *<data_i_czas>* to data i czas UTC utworzenia pliku. Jeśli otworzysz ten plik w edytorze tekstu, powinien on wyglądać mniej więcej następująco:



Nazwa pliku tworzonego przez Rails zawiera datę i czas UTC, kiedy plik był wygenerowany.

```
class CreateTickets < ActiveRecord::Migration
  def self.up
    create_table :tickets do |t|
      t.string :name
      t.string :seat_id_seq
      t.text :address
      t.decimal :price_paid
      t.string :email_address
      t.timestamps
    end
  end
  def self.down
    drop_table :tickets
  end
end
```

Oto zawartość pliku migracji.

Migracja to niewielki skrypt języka Ruby. Zamiast wykonywać ten skrypt bezpośrednio, należy go wykonać za pomocą innego narzędzia Rails, o nazwie **rake**. By wykonać migrację, należy wpisać **`rake db:migrate`** w wierszu poleceń. Poniższe polecenie wykonuje kod migracji i tworzy tabelę:

```
Plik Edycja Okno Pomoc
> rake db:migrate
```

Uwaga!

Nie martw się, niebawem powiemy więcej na ten temat.

Rails wykorzystuje notację CamelCase w przypadku klas, a notację ze znakiem „_" dla nazw plików.

Dlatego właśnie migracja nosi nazwę „CreateTickets" i znajduje się w pliku o nazwie `„..._create_tickets.rb"`.

WYTĘŻ UMYSŁ

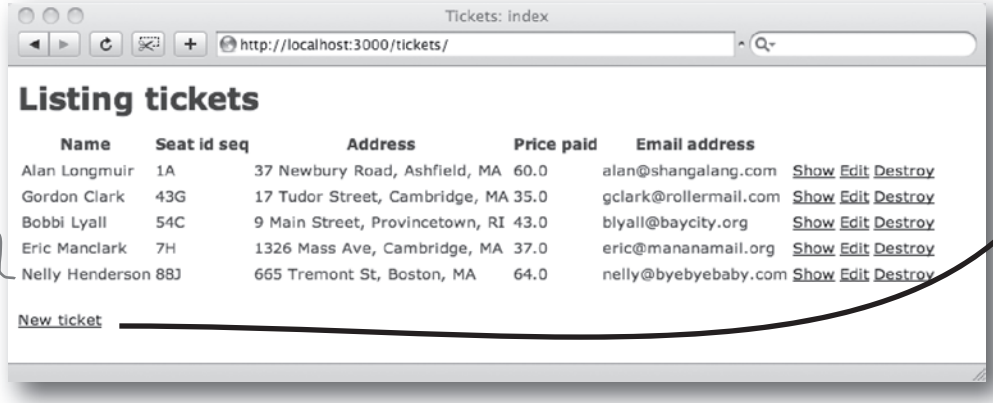
Dlaczego migracja zawiera w nazwie datę i czas?

Jazda próbna

Upewnij się, że utworzyłeś tabelę `tickets` za pomocą polecenia `rake`. Teraz wróć do przeglądarki i odśwież stronę:

```
http://localhost:3000/tickets
```

Twoja aplikacja internetowa działa! W kilka minut możesz utworzyć kilka testowych rekordów:

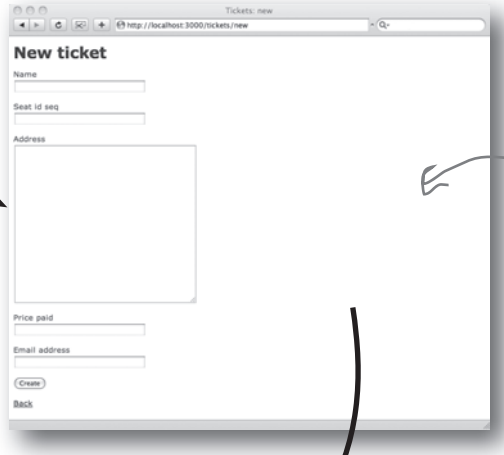


Oto kilka rekordów, jakie dodaliśmy. Ty też dodaj kilka!

Czekaj! To niemożliwe! Wpisaliśmy kilka poleceń do konsoli i to spowodowało utworzenie całej aplikacji?

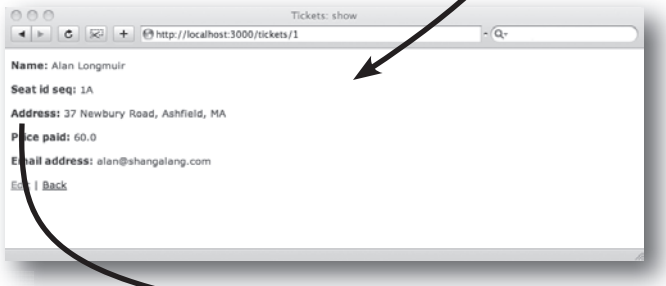
Tak — zbudowaliśmy o wiele więcej niż tylko stronę główną. Zbudowaliśmy cały system.

Rusztowanie wygenerowało cały zbiór stron, które pozwalają użytkownikom tworzyć, modyfikować i usuwać informacje dotyczące biletów. By zobaczyć, jak działa cała aplikacja, utworzymy i zmodyfikujemy kolejny rekord.



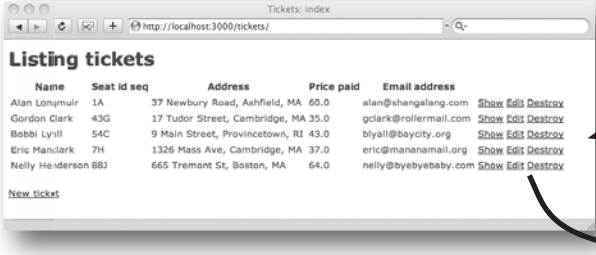
Kliknięcie odnośnika „New ticket" na stronie internetowej przeniesie Cię do formularza, w którym możesz utworzyć nowy bilet.

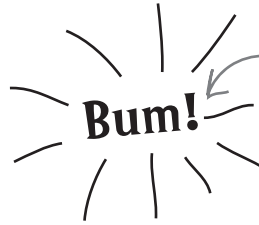
Po przesłaniu formularza będziesz w stanie wczytać nowy bilet z bazy danych i wyświetlić go.



Przycisk „Edit" na stronie wyświetlającej bilet pozwala uaktualnić dowolne informacje.

Kliknięcie tącza „Back" przeglądarki przenosi Cię z powrotem na stronę główną...





...gdzie możesz wybrać bilet do usunięcia.

CELNE SPOSTRZEŻENIA

- Polecenie

  ```
  rails <nazwa aplikacji>
  ```

  generuje dla Ciebie aplikację znajdującą się w folderze `<nazwa aplikacji>`. Rails tworzy również foldery oraz pliki stanowiące podstawową strukturę aplikacji.
- Rails zawiera wbudowany serwer WWW. By go uruchomić, należy użyć polecenia:

  ```
  ruby script/server
  ```

  Domyślna strona główna znajduje się pod adresem:

  ```
  http://localhost:3000/
  ```
- Aplikacje Rails są zgodne z zasadą: „Konwencja ważniejsza od konfiguracji".
- Operacje tworzenia, odczytywania, uaktualniania i usuwania wykonywane na bazie danych znane są pod nazwą operacji CRUD.
- Rusztowanie tworzy dla Ciebie kod CRUD. By utworzyć rusztowanie dla obiektu `thing`, należy wykonać:

  ```
  ruby script/generate scaffold thing
     <nazwa kolumny 1>:<typ kolumny 1>
     <nazwa kolumny 2>:<typ kolumny 2>
     ...
  ```
- By zobaczyć rusztowanie, należy wybrać w przeglądarce adres:

  ```
  http://localhost:3000/things
  ```
- Aplikacje Rails są zgodne z zasadą: „Nie powtarzaj się".
- Migracja to skrypt zmieniający strukturę dołączonej bazy danych. Migrację wykonuje się za pomocą polecenia:

  ```
  rake db:migrate
  ```

Nie istnieją głupie pytania

P: Niektóre polecenia rozpoczynają się od `rails`, inne od `ruby`, a jeszcze inne od `rake`. Jaka jest między nimi różnica?

O: Polecenie `rails` wykorzystywane jest do utworzenia nowej aplikacji. Z kolei `ruby` to interpreter języka Ruby, który wykorzystywany jest do wykonywania skryptów narzędzi przechowywanych w folderze *scripts*. Polecenia `ruby` i `rake` używane są w zasadzie do wszystkich zadań w Rails.

P: Czym zatem jest `rake`?

O: `rake` to polecenie użyte przez nas do wykonania migracji bazy danych. Oznacza „Ruby make" i wykorzystywane jest do niektórych z tych samych zadań, do jakich w językach takich, jak C i Java służą, odpowiednio, `make` oraz `ant`. Kiedy `rake` otrzymuje zadanie do wykonania (na przykład migrację), jest w stanie w sprytny sposób przeanalizować aplikację i zdecydować, które skrypty ma wykonać. Jest zatem nieco sprytniejsze od `ruby` i wykorzystywane jest do bardziej skomplikowanych zadań, takich jak modyfikowanie struktury bazy danych czy wykonywanie testów.

P: Nie rozumiem zasady: „Konwencja ważniejsza od konfiguracji". Co ona oznacza?

O: Wiele języków programowania daje Ci mnóstwo opcji, z których możesz wybierać, tak jak wybierasz wyposażenie nowego samochodu. W przypadku języka z dużą liczbą dostępnych opcji konieczne jest przechowanie wyborów programisty — zazwyczaj w dużych plikach XML. Rails ma inne podejście. W Rails wszystko nazywane jest w spójny sposób i przechowywane w ustandaryzowanym miejscu. Jest to nazywane podejściem „konwencjonalnym" — nie dlatego, że jest staromodne, ale dlatego, że jest zgodne z pewnymi „konwencjami" czy „standardami".

P: Nie mogę zatem zmienić sposobu działania Rails?

O: W Rails możesz zmienić właściwie wszystko, jeśli jednak będziesz przestrzegał konwencji, wkrótce zauważysz, że tworzenie aplikacji pójdzie Ci szybciej, a inne osoby uznają Twój kod za łatwiejszy do zrozumienia.

Pięknie! Uratowałeś pracę kumpla!

Twoje krótkie spotkanie z Rails uratowało stołek przyjaciela… przynajmniej na razie. Wygląda na to, że właśnie przyszedł kolejny e-mail:

Wielkie dzięki!

Niesamowite jest widzieć działającą aplikację — i jeszcze wszystko poszło Ci tak szybko! Rails wygląda na fantastyczną technologię. Samo to, jak wszystko pokazuje się od razu po edycji kodu… Żadnej kompilacji. Żadnego wdrażania. To musi być super!

Naprawdę uratowałeś mój stołek.

Tylko jedna uwaga — podpisy dla „seat_id_seq" powinny być bardziej czytelne dla człowieka, coś w stylu „Seat #". Myślisz, że dałoby się to zrobić?

Jak zatem możemy zmienić podpisy?

Platforma Rails szybko wygenerowała dla nas aplikację internetową, co oszczędziło nam wiele czasu i wysiłku. Co jednak zrobić, jeśli chcemy wprowadzić niewielkie zmiany do wyglądu wygenerowanych stron?

Jak łatwe jest modyfikowanie stron wygenerowanych dla nas przez Rails?

By zmodyfikować aplikację, musisz przyjrzeć się jej architekturze

Rusztowanie po prostu generuje dla nas kod. Po wygenerowaniu kodu dostosowanie go do własnych potrzeb należy do Ciebie. A jeśli spojrzysz do folderu aplikacji, zobaczysz, że jest w nim mnóstwo wygenerowanego kodu, który możesz chcieć dostosować do własnych wymagań.

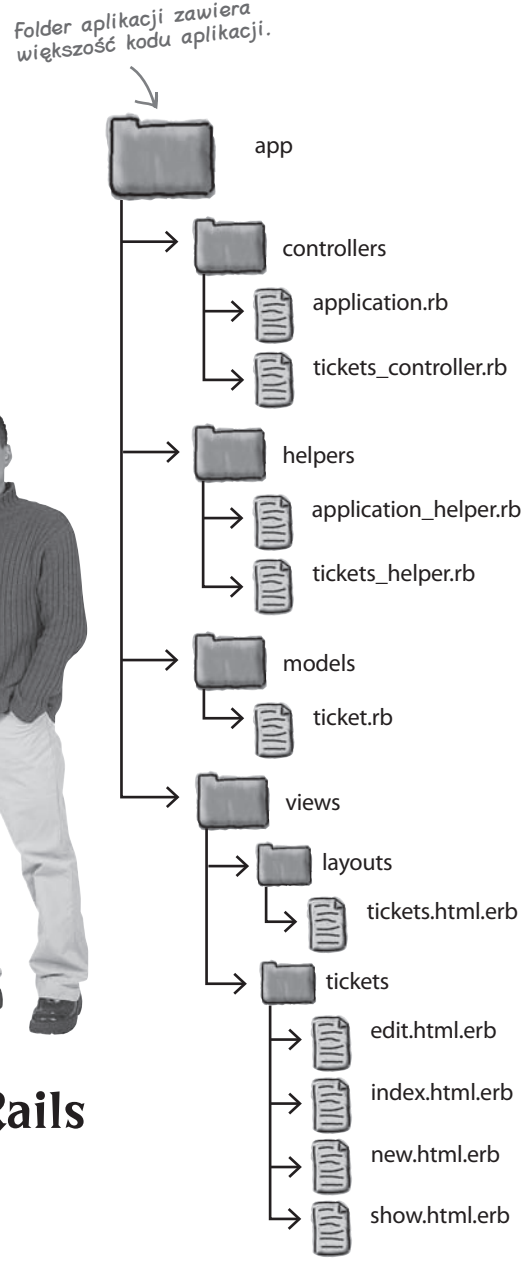
Jeśli zatem musisz wprowadzić zmiany do aplikacji — na przykład zmodyfikować podpisy stron — od czego możesz zacząć?

Polegaj na konwencjach Rails.

Pamiętasz, jak powiedzieliśmy, że aplikacje Rails zgodne są z zasadą: „Konwencja ważniejsza od konfiguracji”? To ułatwi nam modyfikację aplikacji. Dlaczego? Ponieważ kod podzielony jest zgodnie z jego **funkcją**. Oznacza to, że skrypty Ruby robiące *podobne rzeczy* znajdują się w *podobnych miejscach*.

Jeśli zatem potrzebujesz zmienić zachowanie aplikacji Rails, powinieneś być w stanie zidentyfikować, gdzie należy poprawić kod, a następnie go zmodyfikować.

Oczywiście, żeby to jednak zrobić, musisz zrozumieć…



Standardową architekturę Rails

Trzy części Twojej aplikacji: model, widok i kontroler

Prawie cały kod w aplikacjach Rails mieści się w jednej z trzech kategorii:

1 Kod modelu

Kod modelu (ang. *model*) zarządza zapisem danych do bazy oraz ich odczytem. **Obiekty** kodu modelu reprezentują rzeczy istniejące w *domenie systemu* — tak jak **bilety** w systemie biletów.

Oznacza to po prostu problemy biznesowe, jakie aplikacja stara się rozwiązać.

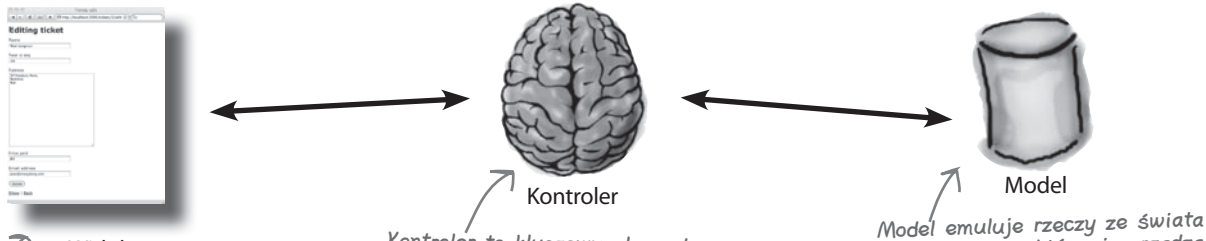
2 Kod widoku

Widok (ang. *view*) to część aplikacji **prezentowana** użytkownikowi. Z tego powodu czasami nazywana jest również **warstwą prezentacyjną**. W przypadku aplikacji internetowej widok przede wszystkim generuje strony internetowe.

3 Kod kontrolera

Kontroler (ang. *controller*) to prawdziwy *mózg* aplikacji. Decyduje o tym, w jaki sposób użytkownik wchodzi w **interakcję** z systemem, kontrolując, do których danych z modelu odbywa się dostęp i które części widoku je zaprezentują..

Różne typy kodu są w aplikacji Rails połączone w następujący sposób:



Widok składa się ze stron internetowych, które prezentują aplikację użytkownikowi.

Kontroler to kluczowy element podejmowania decyzji. Decyduje, do których danych z modelu odbywa się dostęp, a także które części widoku te dane wyświetlą.

Model emuluje rzeczy ze świata rzeczywistego, którymi zarządza aplikacja. W systemie sprzedaży biletów obiekty biletów żyją w modelu.

Cała prawda o Rails

Wywiad tygodnia:

Pytamy najbardziej pożądaną platformę internetową o to, co nią kieruje

Head First: Witaj Rails, ogromnie się cieszymy, że mogłaś do nas wpaść.

Rails: Cieszę się, że mogę tu być.

Head First: Musiało ci być ciężko znaleźć wolną chwilę w tak napiętym terminarzu.

Rails: Jasne, jestem niesamowicie zajęta. Z tymi wszystkimi połączeniami z bazą danych, logiką aplikacji i stronami, jakie muszę prezentować, nie zostaje mi zbyt wiele czasu dla siebie. Ale daję sobie jakoś radę — mam niezłych współpracowników.

Head First: Zastanawiałem się nad jedną rzeczą — mam nadzieję, że się nie obrazisz… Kiedy tworzysz nową aplikację, po co jest w niej od razu tyle katalogów?

Rails: Co mogę powiedzieć? Pomocna ze mnie dusza. Z czasem nauczyłam się, co ludzie chcą robić w swoich aplikacjach. Nie podoba mi się, gdy ludzie ręcznie muszą tworzyć ciągle te same rzeczy.

Head First: Ale czy to nie jest nieco… no, mylące?

Rails: Proszę cię. Jestem osobą przestrzegającą konwencji. Żadnych niespodzianek. Kiedy raz nauczysz się, jak ze mną współpracować, zobaczysz, że szybko się do mnie przyzwyczaisz.

Head First: Słyszałem, że nie lubisz, by cię konfigurowano.

Rails: Możesz mnie konfigurować w dowolny sposób, jednak większość osób woli pracować w ten sam sposób co ja. Konwencja ważniejsza od konfiguracji. Jasne?

Head First: No tak — to w końcu jedna z twoich zasad, prawda?

Rails: Właśnie — ta i jeszcze: „Nie powtarzaj się”.

Head First: I jeszcze co?

Rails: „Nie powtarzaj się”?

Head First: I jeszcze co?

Rails: Nie… Hej, zabawny z ciebie facet!

Nie istnieją głupie pytania

P: Gdzie w mojej aplikacji powinna się znajdować logika biznesowa?

O: Cóż, zależy, co rozumiesz pod tym pojęciem. Niektóre osoby definiują logikę biznesową jako reguły związane z zarządzaniem danymi. W tym przypadku logika biznesowa mieści się w modelu. Inne osoby definiują logikę biznesową jako reguły definiujące sposób działania systemu — na przykład jakie możliwości ma aplikacja i jaka jest kolejność dostępu do nich. W tym przypadku logika biznesowa mieści się w kontrolerze. W dalszej części książki będziemy używali pojęć „logika modelu” oraz „logika aplikacji”, by rozróżnić te dwie sytuacje.

P: Jaka jest różnica między widokiem a kontrolerem?

O: Widok decyduje o tym, jak *wygląda* aplikacja, a kontroler decyduje o tym, jak *działa*. Widok definiuje zatem kolor przycisku na stronie, a także widoczny na nim tekst, natomiast kontroler decyduje, co dzieje się po naciśnięciu przycisku.

P: Jaki kod będę pisał najczęściej?

O: To zależy od aplikacji i programisty. Jeśli zobaczysz, że najczęściej dodajesz kod do którejś z trzech części aplikacji, możesz się zastanowić, czy kolejny dodawany fragment przynależy do prezentacji, interakcji, czy też modelowania.

JAKI JEST MÓJ CEL?

Dopasuj opis kodu do części aplikacji, z którą łączy się ten kod.

Projekt kart w pasjansie internetowym.

W systemie bankowości elektronicznej ten kod decyduje o tym, czy chcesz przelać pieniądze na konto, czy z konta.

Obiekt „spotkanie” w aplikacji terminarza.

W systemie bloga kod ten decyduje, czy komentarze mają być wyświetlone jako tabela, czy jako lista.

Ten kod rejestruje ofertę w internetowym portalu aukcyjnym.

Kod decyduje, że musisz się zalogować do aplikacji udostępniającej pocztę elektroniczną.

Menu składające się z odnośników.

Model

Widok

Kontroler

JAKI JEST MÓJ CEL?

ROZWIĄZANIE

Dopasuj opis kodu do części aplikacji, z którą łączy się ten kod.

Projekt kart w pasjansie internetowym.

W systemie bankowości elektronicznej ten kod decyduje o tym, czy chcesz przelać pieniądze na konto, czy z konta.

Obiekt „spotkanie” w aplikacji terminarza.

W systemie bloga kod ten decyduje, czy komentarze mają być wyświetlone jako tabela, czy jako lista.

Ten kod rejestruje ofertę w internetowym portalu aukcyjnym.

Kod decyduje, że musisz się zalogować do aplikacji udostępniającej pocztę elektroniczną.

Menu składające się z odnośników.

Model

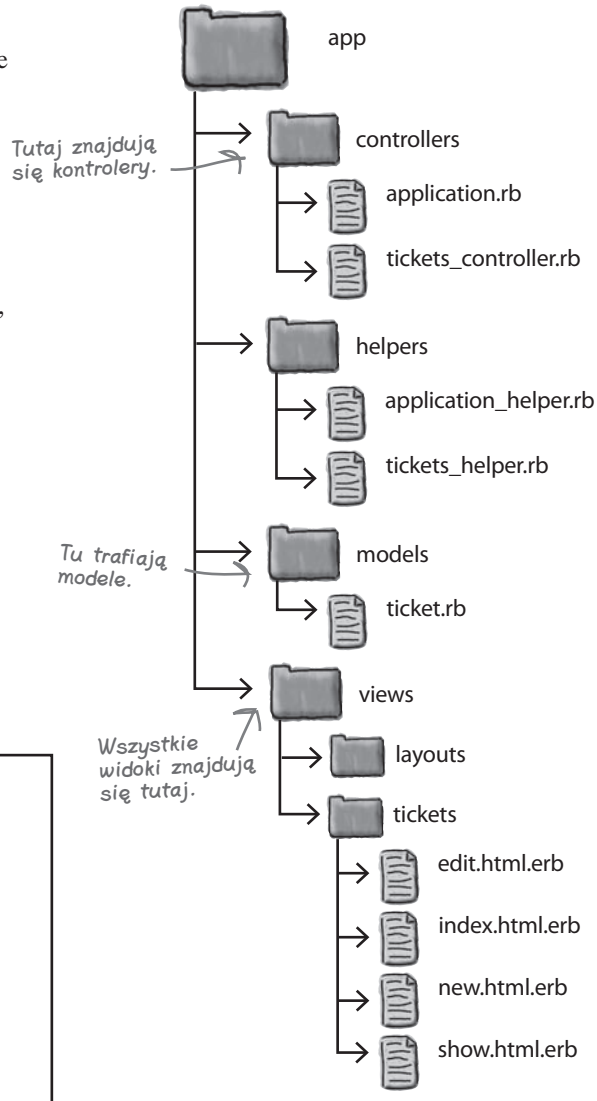
Widok

Kontroler

Trzy typy kodu przechowywane są w OSOBNYCH folderach

Rails woli zatem konwencję od konfiguracji i wykorzystuje architekturę MVC (*Model-View-Controller*, czyli Model-Widok-Kontroler). I co z tego wynika?

W jaki sposób architektura MVC pomoże nam zmienić podpisy na naszych stronach i tym samym poprawić aplikację? Przyjrzyjmy się raz jeszcze plikom wygenerowanym przez rusztowanie. Ponieważ kod jest w jasny sposób podzielony na trzy odrębne typy — model, widok oraz kontroler — Rails umieszcza każdy typ w osobnym folderze.



Zaostrz ołówek

Na diagramie folderów znajdującym się po prawej stronie zaznacz pliki, o których sądzisz, że trzeba je będzie zmodyfikować w celu edycji podpisów na stronach.

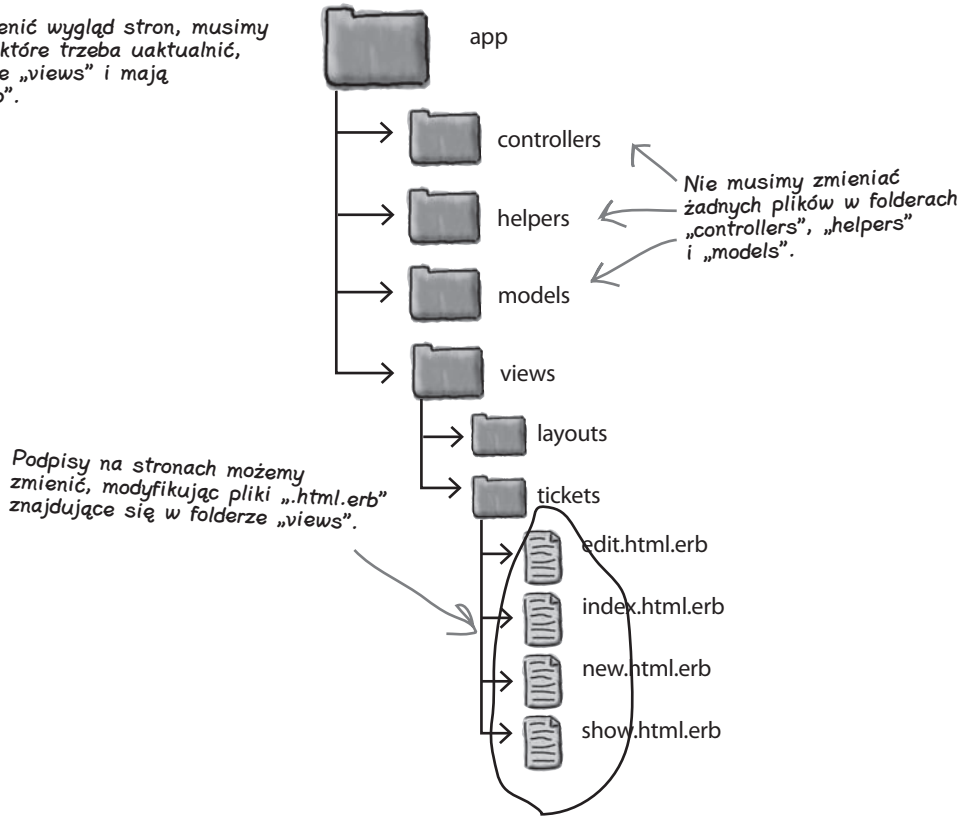
Napisz następnie, ***dlaczego*** wybrałeś te właśnie pliki.

Zaostrz ołówek

Rozwiązanie

Twoje zadanie polegało na zaznaczeniu plików, które trzeba zmodyfikować w celu zmiany podpisów na stronach.

Ponieważ musimy zmienić wygląd stron, musimy zmienić widoki. Pliki, które trzeba uaktualnić, znajdują się w folderze „views" i mają rozszerzenie „.html.erb".



Trzeba zmodyfikować pliki WIDOKU

Jeśli chcemy zmienić podpisy na stronach internetowych, musimy zmodyfikować kod widoku. Cały kod widoku znajduje się w folderze *app/views*.

Pliki widoku generują strony internetowe i nazywane są ***szablonami stron***. Czym zatem jest szablon strony i co zawierają te szablony?

Edycja kodu HTML w widoku

Jak zatem naprawdę wygląda szablon strony? Otwórz cztery pliki *.html.erb* z folderu *views/tickets* za pomocą edytora tekstu. Zawartość plików wygląda w dużej mierze jak kod HTML.

Chcemy zmienić podpisy dla pola `seat_id_seq` na `Seat #`. By to zrobić, odszukaj tekst „Seat id seq” we wszystkich czterech plikach, zmień go na „Seat #”, a następnie zapisz zmiany.

Zrób tak!

Przejdź do każdego z czterech plików **`.html.erb`** w folderze **`views/tickets`** i zmień tekst **`Seat id seq`** na **`Seat #`**. Spowoduje to zmianę podpisów na stronach na Seat #.

```
<p>
  <b>Name:</b>
  <%=h @ticket.name %>
</p>
<p>
  <b>Seat id seq:</b>
  <%=h @ticket.seat_id_seq %>
</p>
<p>
  <b>Address:</b>
  <%=h @ticket.address %>
</p>
```

To jest tekst, który musisz zmienić na „Seat #”.

#

```
<h1>Editing ticket</h1>

<% form_for(@ticket) do |f| %>
  <%= f.error_messages %>

  <p>
    <%= f.label :name %><br />
    <%= f.text_field :name %>
  </p>
  <p>
    <%= f.label :seat_id_seq %><br />
    <%= f.text_field :seat_id_seq %>
  </p>
```

Nie zapomnij dodać cudzysłowów — ponieważ teraz jest to łańcuch znaków.

Ten symbol trzeba będzie zmienić na łańcuch znaków „Seat #”.

"Seat #"

Jeśli zmodyfikujesz podpisy w kodzie HTML, zmiany będą natychmiast widoczne w przeglądarce. Jeśli *teraz* wprowadzisz zmiany i odświeżysz stronę w przeglądarce, będą one widoczne *natychmiast*. Sprawdźmy to od razu…

Nie istnieją głupie pytania

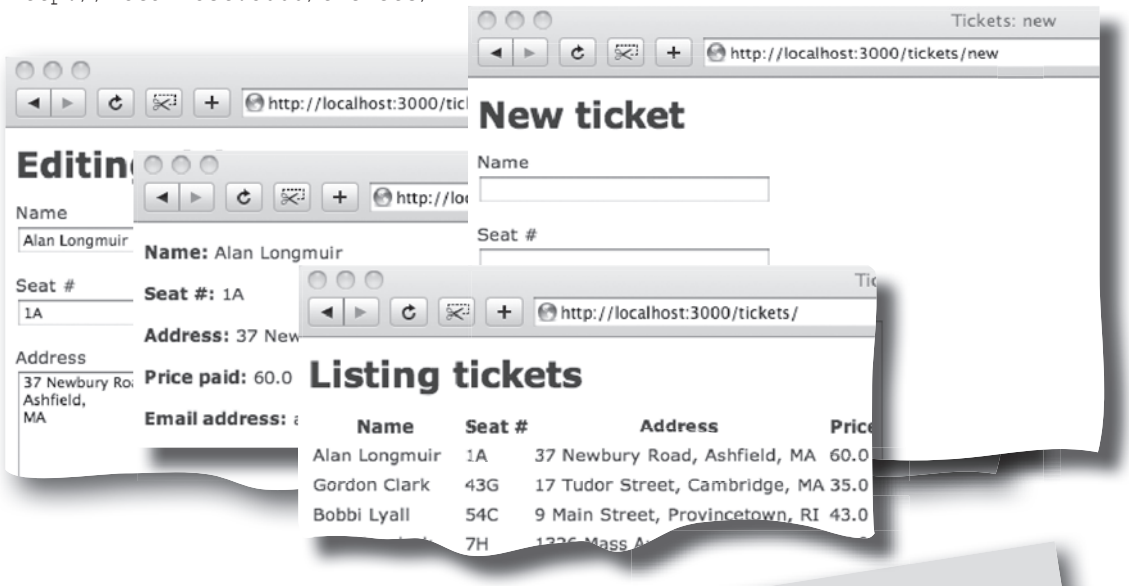
P: Nazywacie `:seat_id_seq` symbolem. Czym jest symbol?

O: Symbol nieco przypomina łańcuch znaków. Łańcuch znaków otoczony jest cudzysłowem, natomiast symbol zawsze rozpoczyna się od dwukropka (**`:`**). Symbole wykorzystywane są zazwyczaj w Rails do nazywania różnych rzeczy, ponieważ są nieco bardziej wydajne pod względem pamięci. W większości przypadków symbole i łańcuchy znaków mogą być wykorzystywane wymiennie.

Jazda próbna

Odśwież stronę znajdującą się pod adresem:

```
http://localhost:3000/tickets/
```

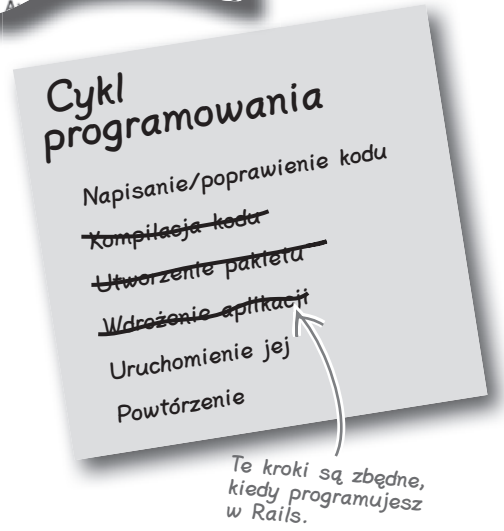


Teraz wszystkie podpisy brzmią „Seat #". Dokładnie tego chcieliśmy.

Czy zauważyłeś, jak szybko zmiana pokazała się w aplikacji?

To dzięki temu, że platforma Rails zbudowana jest w języku Ruby, a kod w tym języku nie musi być kompilowany. Serwer WWW Rails może po prostu wykonać uaktualniony kod źródłowy.
Czy ma to jednak jakieś znaczenie?

Owszem, by wypróbować zmodyfikowany kod, musisz wykonać o wiele mniej kroków. Nie trzeba na przykład kompilować kodu ani łączyć go w pakiety i gdzieś wdrażać. Wystarczy tylko napisać kod i wykonać go. Cykl programowania w Rails jest bardzo szybki, a wprowadzenie zmian do aplikacji odbywa się równie prędko.



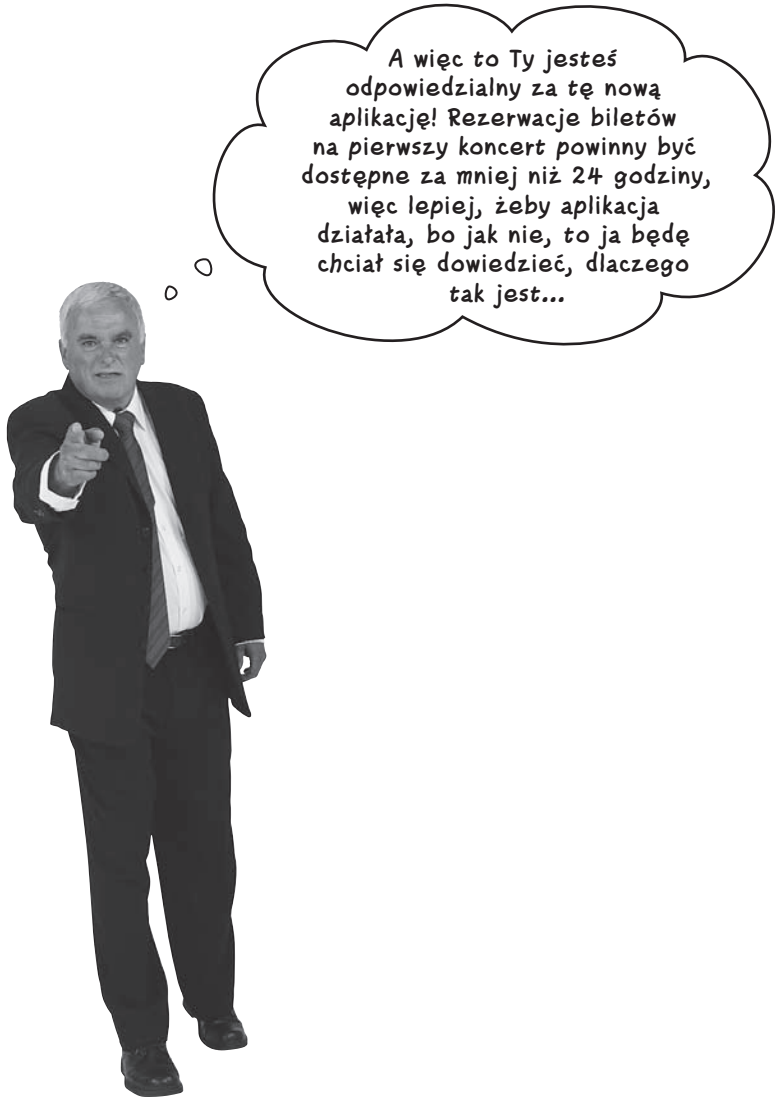
Niedziela, godzina 8 rano

Wprowadziłeś dwie poprawki, ale teraz dzwoni telefon…
Co u licha?

Hej, stary! Fantastycznie, że aplikacja działa tak świetnie. Ale słuchaj... Myślę, że powinieneś wiedzieć, iż zadzwonił do mnie szef i chciał się dowiedzieć, gdzie mieszkasz. Bardzo chciał zobaczyć całą tę robotę, jaką wykonałeś, ale ciągle się trochę martwi, że może ona nie być gotowa na jutro rano, więc jeszcze dziś chciał sprawdzić co i jak. Dziękuję za całą pracę, jaką wykonałeś. A przy okazji — mówiłem Ci już, że szef chciałby, by dla każdego zamówienia biletu obok adresu e-mail odnotowany był również numer telefonu? Przepraszam — musiało mi to umknąć.

PUK!

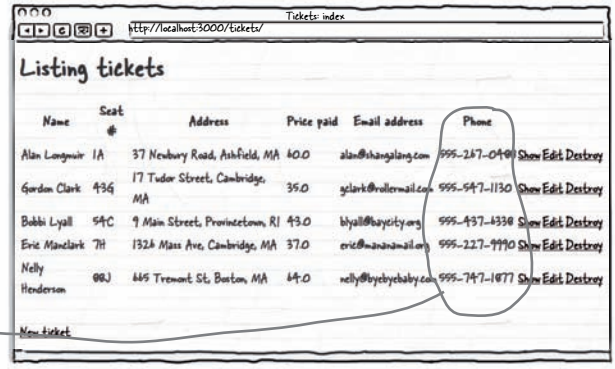
PUK!


A więc to Ty jesteś odpowiedzialny za tę nową aplikację! Rezerwacje biletów na pierwszy koncert powinny być dostępne za mniej niż 24 godziny, więc lepiej, żeby aplikacja działała, bo jak nie, to ja będę chciał się dowiedzieć, dlaczego tak jest...

Aplikacja musi teraz przechować większą liczbę informacji

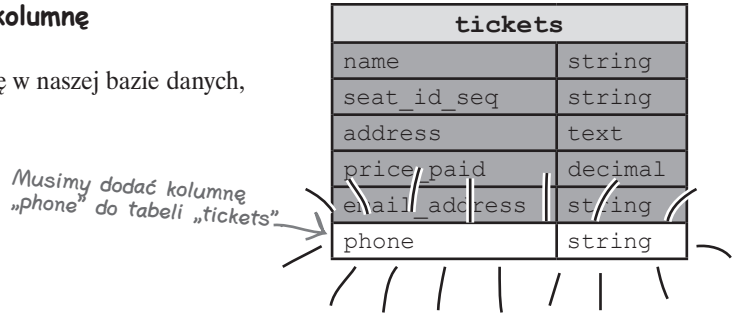
Wszystko było właściwie skończone do momentu, gdy Twój przyjaciel wspomniał, że należy zapisywać numery telefonów. Potrzebujemy więcej danych, a co to oznacza dla naszej aplikacji?

1 **Potrzebne nam dodatkowe pole wyświetlane na każdej stronie.**
Na szczęście wiemy, jak ulepszać szablony stron, więc nie powinno to stanowić problemu.



Musimy dodać takie dodatkowe pole do strony.

2 **Musimy przechować dodatkową kolumnę w bazie danych.**
Musimy przechować dodatkową kolumnę w naszej bazie danych, jednak jak to zrobić?



Musimy dodać kolumnę „phone" do tabeli „tickets"

Zaostrz ołówek

Potrzebujemy dodać kolumnę do tabeli bazy danych. Napisz poniżej, jaki **typ** skryptu wykorzystywaliśmy wcześniej w celu modyfikacji struktury bazy danych.

.....

Zaostrz ołówek

Rozwiązanie

Miałeś napisać, jaki typ skryptu wykorzystywaliśmy wcześniej w celu modyfikacji struktury bazy danych.

Migracja

Migracja to po prostu skrypt w języku Ruby

Aby dodać kolumnę do tabeli, potrzebna nam migracja. Ale *czym* tak naprawdę jest migracja? Przyjrzyjmy się tej, która utworzyła naszą tabelę z biletami.

```
class CreateTickets < ActiveRecord::Migration
  def self.up
    create_table :tickets do |t|
      t.string :name
      t.string :seat_id_seq
      t.text :address
      t.decimal :price_paid
      t.string :email_address
      t.timestamps
    end
  end
  def self.down
    drop_table :tickets
  end
end
```

Musimy utworzyć kod przypominający ten, jednak zamiast tworzenia tabeli nasza migracja musi dodać kolumnę.

Nie istnieją głupie pytania

P: Część kodu migracji wygląda tak, jakby usuwała tabelę. Dlaczego tak jest?

O: Migracje mogą zrobić o wiele więcej, niż pokazujemy tutaj. Każda migracja może na przykład odwrócić wprowadzone przez siebie zmiany. Z tego powodu kod tworzący nową tabelę jest połączony z kodem mogącym tę tabelę usunąć. Nie musisz teraz wiedzieć więcej na ten temat.

P: Nie muszę rozumieć kodu? Czy żeby posługiwać się Rails nie należy rozumieć kodu w języku Ruby?

O: Im lepiej będziesz rozumiał Ruby, tym większą kontrolę będziesz miał nad Rails. W miarę lektury książki będziesz się dowiadywał coraz więcej o języku Ruby.

P: Jeśli migracja to tylko skrypt w języku Ruby, dlaczego muszę używać `rake`? Czemu nie mogę po prostu wykonać skryptu?

O: Dobre pytanie. Część Ruby zaprojektowana została tak, by bezpośrednio wykonać kod, a część nie. Migracji nie należy wykonywać w sposób bezpośredni. Powinny one być wykonywane za pomocą `rake`.

P: Świetnie — ale dlaczego?

O: Polecenie `rake` jest „sprytniejsze" od `ruby`. Kiedy wywołasz `rake db:migrate`, tak naprawdę mówisz do `rake`: „Upewnij się, że wszystkie migracje zostały wykonane". `rake` może zadecydować, by nie wywoływać migracji, jeśli nie ma takiej potrzeby. Ruby nie może samodzielnie podejmować takich decyzji.

P: Czy nie mogę po prostu ręcznie zmodyfikować tabeli `tickets`?

O: Mógłbyś tak zrobić, jednak lepiej jest zarządzać strukturą bazy danych za pomocą migracji. Kiedy opublikujesz aplikację, będziesz musiał odtworzyć strukturę danych w produkcyjnej bazie danych. Jeśli skorzystasz z migracji, polecenie `rake` będzie w stanie dopasować struktury danych w bazie produkcyjnej do tego, czego potrzebuje Twoja aplikacja. Jeśli ręcznie zmodyfikujesz strukturę danych, rzeczy łatwo mogą się rozsynchronizować. Tak jak w większości sytuacji w Rails, jeśli będziesz się stosował do konwencji, ułatwisz sobie życie.

Rails może generować migracje

Przypomnij sobie, jak wygenerowaliśmy rusztowanie za pomocą polecenia:

```
ruby script/generate scaffold ticket name:string seat_id_seq:string
address:text price_paid:decimal email_address:string
```

`generate` to skrypt służący do tworzenia kodu w języku Ruby. Dobra wiadomość jest taka, że `generate` pisze *nie tylko* kod rusztowania. Generuje również migracje.

Załóżmy, że wpisałbyś takie polecenie:

Nie wpisuj go naprawdę.

```
ruby script/generate migration PhoneNumber
```

Wygenerowałoby ono nowy pusty plik migracji. Moglibyśmy wtedy dodać kod w języku Ruby w celu zmodyfikowania tabeli. Problem polega na tym, że nie wiemy, jak napisać kod kończący migrację.

Co zatem możemy zrobić? I co może zrobić dla nas Rails?

Nadaj swojej migracji odpowiednią nazwę, a Rails napisze za Ciebie kod

Zauważyłeś już pewnie, że nazwy są dla Rails naprawdę istotne. Kiedy tworzyliśmy rusztowanie o nazwie `ticket`, platforma Rails udostępniła nam aplikację pod adresem *http://localhost:3000/tickets* i wygenerowała migrację tworzącą tabelę o nazwie `tickets`.

Konwencje nazewnictwa są w Rails istotne, ponieważ oszczędzają Ci pracy. Tak samo jest z nazewnictwem migracji. Zamiast nadawać nowej migracji starą nazwę, spróbuj nazwać ją tak:

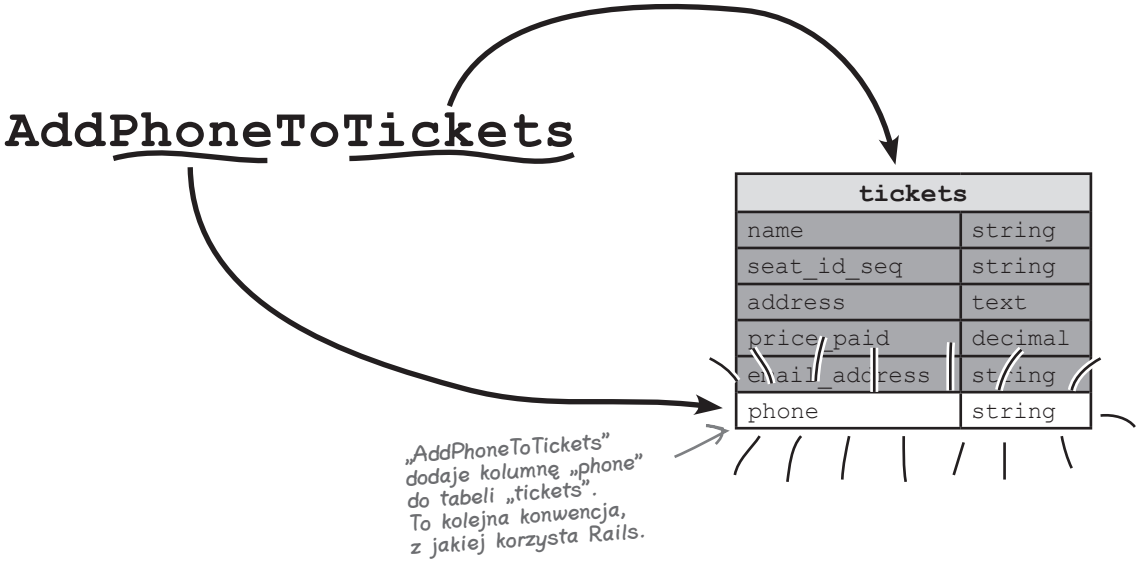
Najważniejszą częścią jest ta nazwa. Przyjmuje postać „Add... To...”

```
Plik Edycja Okno Pomoc
> ruby script/generate migration AddPhoneToTickets phone:string
```

Musisz wykonać to polecenie.

Dlaczego nazwa ma takie znaczenie?

Rails wie, że migracja o nazwie `Add...To...` będzie najprawdopodobniej dodawać określoną kolumnę do określonej tabeli, więc zamiast generować pustą migrację, którą będziesz musiał uzupełnić, ***Rails napisze kod migracji za Ciebie***.

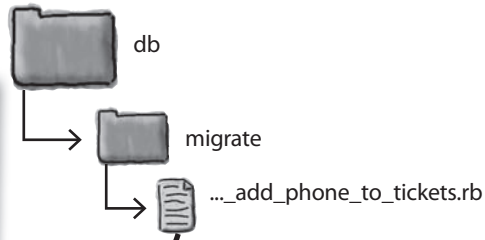


Migrację należy wykonać za pomocą rake

Oto migracja, jaką platforma Rails sprytnie dla Ciebie wygenerowała:

```
class AddPhoneToTickets < ActiveRecord::Migration
  def self.up
    add_column :tickets, :phone, :string
  end

  def self.down
    remove_column :tickets, :phone
  end
end
```



Kiedy wcześniej chcieliśmy wykonać migrację, korzystaliśmy z polecenia `rake`:

```
rake db:migrate
```

Ale czy możemy to zrobić teraz? W końcu nie chcemy, by polecenie `rake` wykonało znowu przez pomyłkę pierwszą migrację.

Data i czas mówią rake, które migracje należy wykonać i w jakiej kolejności

Rails zapisuje ostatnią datę i czas wszystkich wykonywanych migracji. Pozwala to poleceniu `rake` określić, które migracje zostały już wykonane, a które nie. Oznacza to, że przy każdym wykonaniu `rake db:migrate` ***Rails wykona jedynie najnowsze migracje***.

Przetestujmy to. Wykonaj raz jeszcze `rake db:migrate` w celu dodania kolumny `phone` do tabeli `tickets`.


```
Plik Edycja Okno Pomoc
> rake db:migrate
```

Sama zmiana bazy danych nie wystarczy

Rusztowanie *generuje* kod, co jest świetne, ponieważ pozwala nam szybko zacząć. Jednak wadą takiego rozwiązania jest to, że gdy kod zostanie już wygenerowany, **uaktualnienie** go należy do **programisty**.

Dodaliśmy właśnie atrybut `phone` do bazy danych. Jednak ponieważ formularze zostały już wygenerowane przez rusztowanie, nie pobiorą one nowego pola z telefonem automatycznie. Musimy zatem wrócić do szablonów i dodać do nich odniesienia do numeru telefonu:

```
<p>
  <%= f.label :email_address %><br />
  <%= f.text_field :email_address %>
</p>
<p>
  <%= f.label :phone %><br />
  <%= f.text_field :phone %>
</p>
<p>
  <%= f.submit "Update" %>
</p>
```

To znajduje się wewnątrz pliku edit.html.erb.

To jest kod dodany do pliku show.html.erb.

```
<p>
  <b>Email address:</b>
  <%=h @ticket.email_address %>
</p>

<p>
  <b>Phone:</b>
  <%=h @ticket.phone %>
</p>
```

```
<p>
  <%= f.label :email_address %><br />
  <%= f.text_field :email_address %>
</p>
<p>
  <%= f.label :phone %><br />
  <%= f.text_field :phone %>
</p>
<p>
```

A to do pliku new.html.erb — strony zawierającej formularz służący do tworzenia rezerwacji.

I wreszcie do pliku index.html.erb generującego listę wszystkich biletów. Kod musimy tutaj dodać w dwóch miejscach: w nagłówku kolumny oraz w każdym wierszu tabeli.

```
  <th>Email address</th>
  <th>Phone</th>
 </tr>

<% for ticket in @tickets %>
 <tr>
   <td><%=h ticket.name %></td>
   <td><%=h ticket.seat_id_seq %></td>
   <td><%=h ticket.address %></td>
   <td><%=h ticket.price_paid %></td>
   <td><%=h ticket.email_address %></td>
   <td><%=h ticket.phone %></td>
   <td><%= link_to 'Show', ticket %></td>
```

Nie istnieją głupie pytania

P: Dlaczego w niektórych miejscach jest napisane `<%=h ... %>`? Co oznacza `h`?

O: `h` to metoda pomocnicza. Metody pomocnicze wykorzystywane są najczęściej do rzeczy takich, jak formatowanie danych wyjściowych. Metoda pomocnicza `h` znosi znaczenie specjalne pewnych znaków w polu, takich jak `<` czy `&`. Zapobiega to przesłaniu za pomocą strony tekstu zawierającego JavaScript czy inny potencjalnie niebezpieczny kod.

P: Dlaczego w niektórych miejscach użyte zostały łańcuchy znaków, a w innych symbole?

O: Łańcuchy znaków wykorzystywane są w szablonach stron tam, gdzie wymagany jest fragment zwykłego tekstu. Symbole (słowa rozpoczynające się od dwukropków) są często wykorzystywane w podpisach.

P: Dlaczego?

O: Symbole są wydajne pod względem pamięci i większość metod (takich jak `f.label`) przyjmujących parametry lubi otrzymywać symbole zamiast łańcuchów znaków. Jednak w większości przypadków metody Rails pozwalają opcjonalnie wykorzystywać łańcuchy znaków zamiast symboli, jeśli są one łatwiejsze do sformatowania.

Ćwiczenie (nieco dłuższe)

Szef jest zadowolony z nowego sposobu działania aplikacji i teraz chce obok sprzedaży biletów zapisywać również imprezy. Oto struktura danych imprez:

Event (impreza):

artist — wykonawca (łańcuch znaków)

description — krótka biografia (tekst)

price_low — najtańsze bilety (liczba dziesiętna)

price_high — cena sprzedaży biletu (liczba dziesiętna)

event_date — kiedy się odbywa (data)

Jakie polecenie należy wpisać w konsoli, by utworzyć rusztowanie dla danych imprez?

.....

.....

Jakie polecenie wpisałbyś w celu utworzenia tabeli `events` w bazie danych?

.....

Szef chce, by podpisy na stronach brzmiały następująco: „Prices from” dla `price_low`, „To” dla `price_high` oraz „Date” dla `event_date`. Będziesz musiał zmodyfikować cztery szablony w celu wprowadzenia zmian. Wypisz zmiany dla pokazanego poniżej szablonu strony *new.html.erb*:

Plik new.html.erb

```
<h1>New event</h1>
<% form_for(@event) do |f| %>
  <%= f.error_messages %>
  <p>
    <%= f.label :artist %><br />
    <%= f.text_field :artist %>
  </p>
  <p>
    <%= f.label :description %><br />
    <%= f.text_area :description %>
  </p>
  <p>
    <%= f.label :price_low %><br />
    <%= f.text_field :price_low %>
  </p>
  <p>
    <%= f.label :price_high %><br />
    <%= f.text_field :price_high %>
  </p>
  <p>
    <%= f.label :event_date %><br />
    <%= f.date_select :event_date %>
  </p>
  <p>
    <%= f.submit "Create" %>
  </p>
<% end %>
<%= link_to 'Back', events_path %>
```

Podaj nazwy trzech pozostałych szablonów z katalogu *app/views/events*, jakie trzeba będzie zmienić.

.....

Ćwiczenie
(nieco dłuższe)
Rozwiązanie

Szef jest zadowolony z nowego sposobu działania aplikacji i teraz chce obok sprzedaży biletów zapisywać również imprezy. Oto struktura danych imprez:

Event (impreza):

artist — wykonawca (łańcuch znaków)

description — krótka biografia (tekst)

price_low — najtańsze bilety (liczba dziesiętna)

price_high — cena sprzedaży biletu (liczba dziesiętna)

event_date — kiedy się odbywa (data)

Jakie polecenie należy wpisać w konsoli, by utworzyć rusztowanie dla danych imprez?

ruby script/generate scaffold event artist:string description:text price_low:decimal price_high:decimal event_date:date

Jakie polecenie wpisałbyś w celu utworzenia tabeli `events` w bazie danych?

rake db:migrate

Szef chce, by podpisy na stronach brzmiały następująco: „Prices from” dla `price_low`, „To” dla `price_high` oraz „Date” dla `event_date`. Będziesz musiał zmodyfikować cztery szablony w celu wprowadzenia zmian. Wypisz zmiany dla pokazanego poniżej szablonu strony *new.html.erb*:

```
<h1>New event</h1>
<% form_for(@event) do |f| %>
  <%= f.error_messages %>
  <p>
    <%= f.label :artist %><br />
    <%= f.text_field :artist %>
  </p>
  <p>
    <%= f.label :description %><br />
    <%= f.text_area :description %>
  </p>
  <p>                :prices_from
    <%= f.label :price_low %><br />
    <%= f.text_field :price_low %>
  </p>
  <p>                :to
    <%= f.label :price_high %><br />
    <%= f.text_field :price_high %>
  </p>
  <p>                  :date
    <%= f.label :event_date %><br />
    <%= f.date_select :event_date %>
  </p>
  <p>
    <%= f.submit "Create" %>
  </p>
<% end %>
<%= link_to 'Back', events_path %>
```

Plik new.html.erb

"Prices from" także zadziała, ale użycie symbolu jest lepsze

Lub „To”

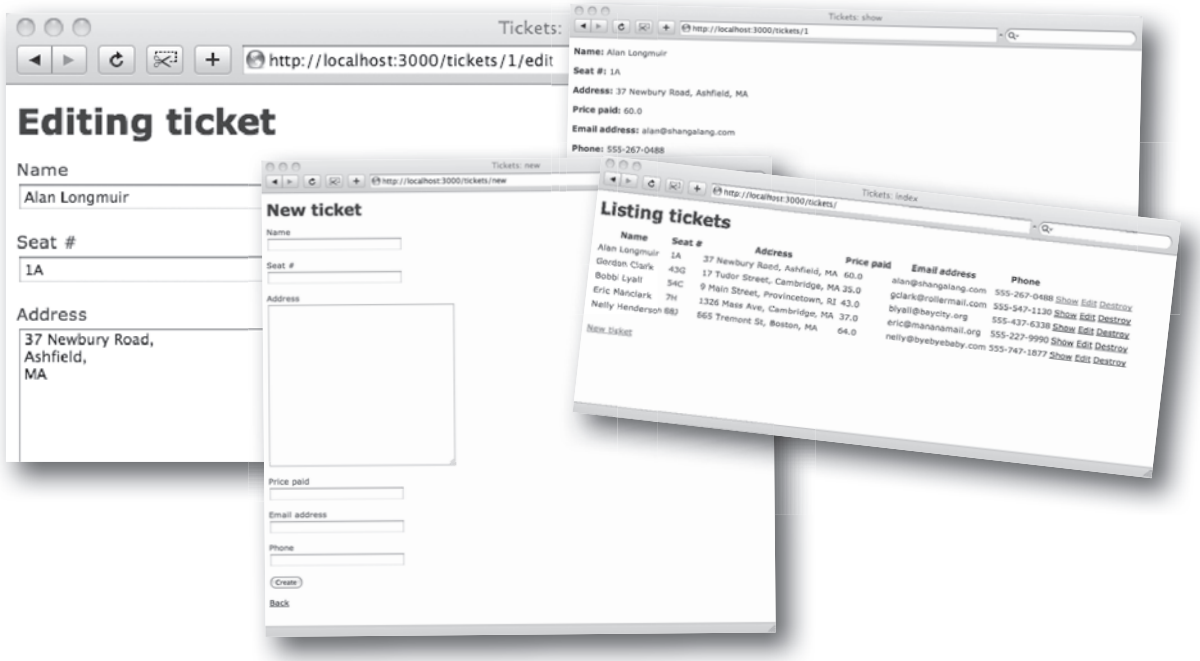
Lub „Date”

Podaj nazwy trzech pozostałych szablonów z katalogu *app/views/events*, jakie trzeba będzie zmienić.

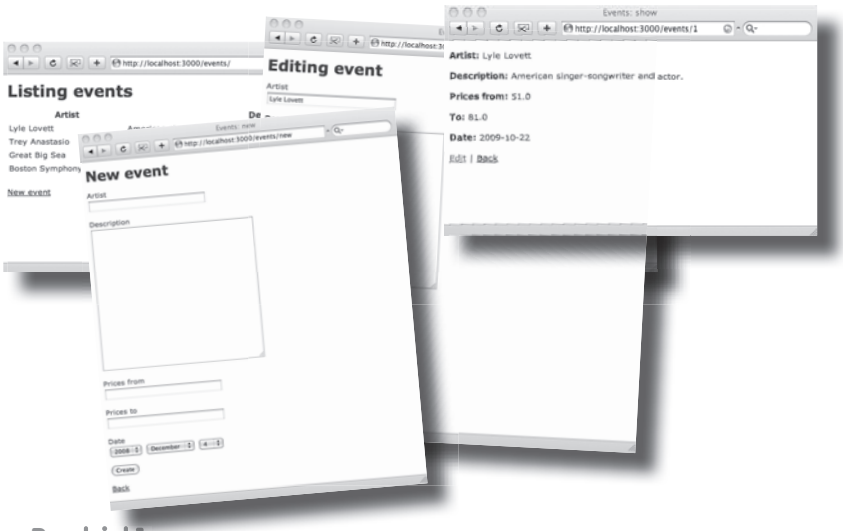
edit.html.erb, show.html.erb oraz index.html.erb

Jazda próbna

Aplikacja ma teraz na stronach z biletami komplet informacji kontaktowych:



Zapisywane są również wszystkie imprezy:

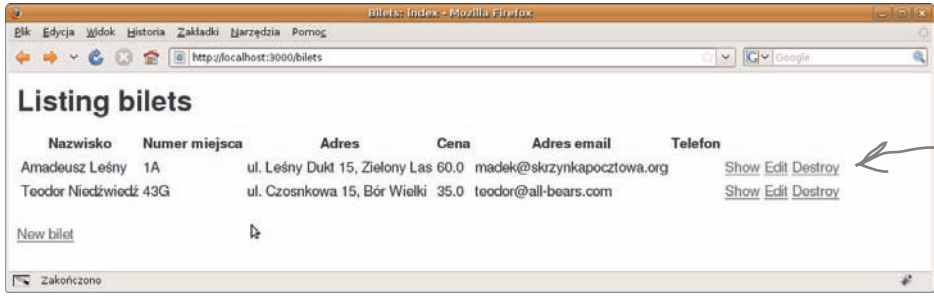


Dlaczego Rails mówi do mnie po angielsku?

Prawdopodobnie od dłuższej chwili nurtuje Cię pewien problem: ***dlaczego aplikacje Rails są w języku angielskim*** i co można zrobić, by ***używały innego języka***? W trakcie pracy w żadnym miejscu nie definiowałeś przecież nagłówków stron czy podpisów przycisków, a tymczasem wszystkie są w języku angielskim. Dlaczego?

Cóż, odpowiedź jest dość prosta. Tworzenie aplikacji w Rails wymaga minimalnego nakładu pracy użytkownika, a **pewne zdefiniowane standardowe operacje i elementy są częścią samej platformy**. Wystarczy, że podasz nazwę aplikacji i za pomocą rusztowania utworzysz odpowiednią strukturę danych, a platforma Rails wygeneruje dla Ciebie pliki widoku ze standardowymi tekstami, ale... *wszystkie te teksty są w języku angielskim*.

Nawet jeśli zaraz na początku spróbujesz nadać aplikacji oraz poszczególnym polom tabeli bazy danych polskie nazwy, uzyskasz co najwyżej przedziwnego potworka językowego (a do tego będziesz się musiał zdrowo nakombinować z automatyczną odmianą wyrazów w liczbie mnogiej — kto by się domyślił, że liczba mnoga od „bilet" to „bilets", prawda?):



Tragedia! Przecież takiego czegoś nie możemy zaprezentować użytkownikom!

Na szczęście nie oznacza to, że tworzenie aplikacji Rails w innym języku jest niemożliwe. Jest możliwe — wymaga jedynie nieco większego nakładu pracy.

Kluczem są pliki widoku

Pewnie nie zdajesz sobie z tego sprawy, ale potrafisz już samodzielnie wprowadzić odpowiednie zmiany. Pamiętasz, jak zmienialiśmy podpisy pól formularza w aplikacji służącej do sprzedaży biletów? No właśnie. Co stoi na przeszkodzie, by podpisy te zmienić na teksty w innym języku, takim jak polski?

Zrób tak!

Otwórz w edytorze tekstu wszystkie pliki widoku — *edit.html.erb*, *index.html.erb*, *new.html.erb* oraz *show.html.erb*. W podobny sposób jak wcześniej zmień wszystkie podpisy formularzy na ich odpowiedniki w języku polskim. Pamiętaj, że polskie teksty musisz także wprowadzić do standardowych elementów HTML tworzących stronę — `<h1>` czy `<th>` — oraz przycisków i odnośników generowanych przez Rails za pomocą metod pomocniczych, takich jak `link_to` oraz `f.submit`.

Uczymy Rails języków obcych

A oto kilka przykładów zmian, jakie będziesz musiał wprowadzić:

```
<h1>Lista biletów</h1>
<table>
  <tr>
    <th>Imię i nazwisko</th>
    <th>Numer miejsca</th>
```

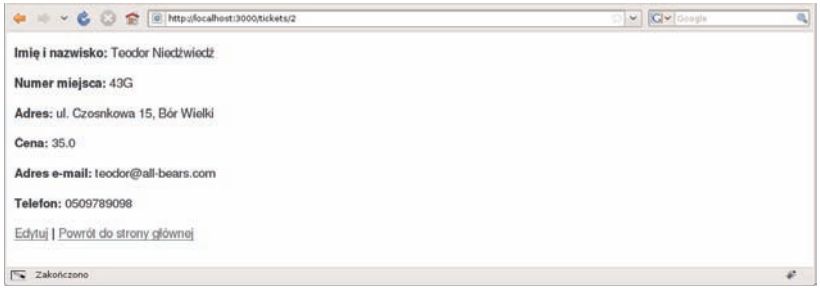
To fragment strony index.html.erb. Pamiętaj o zmianie tekstów elementów HTML!

To część kodu strony new.html.erb. Pokazuje, jak należy zmienić podpisy pól formularza, przycisku oraz odnośnika.

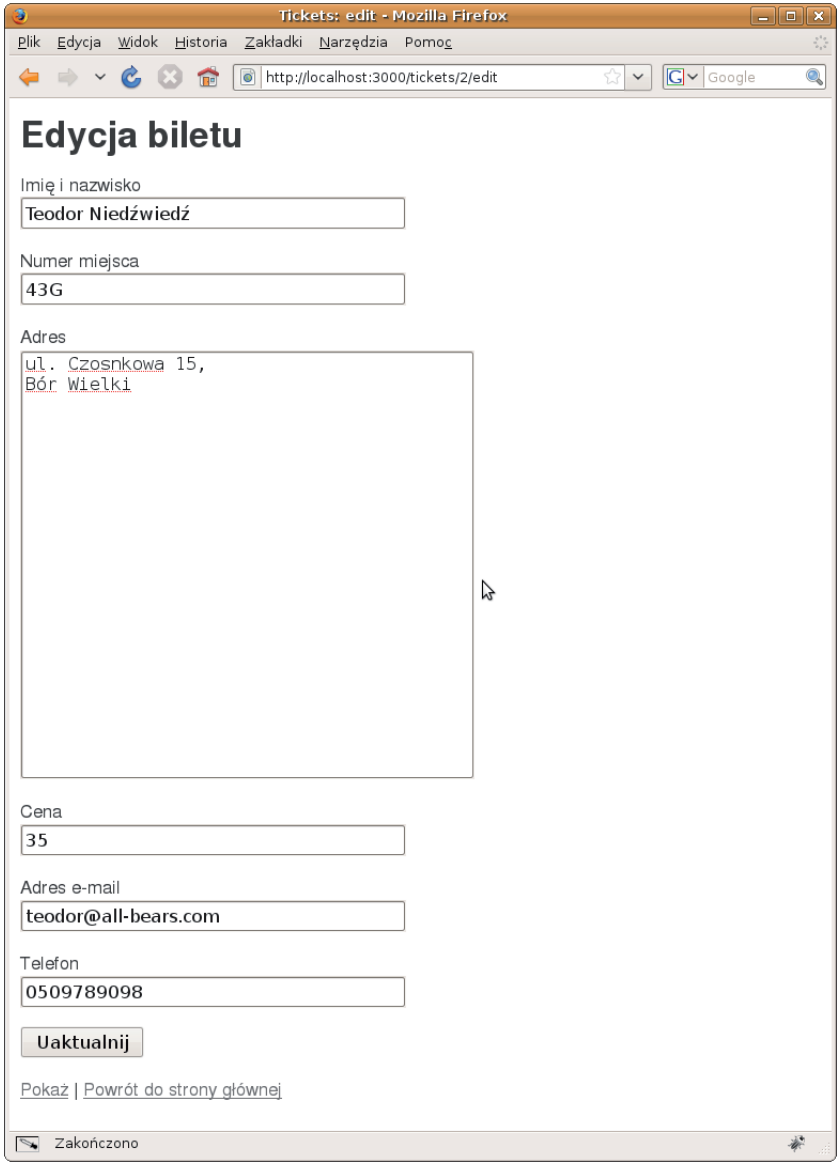
```
  <p>
    <%= f.label :telefon %><br />
    <%= f.text_field :phone %>
  </p>
  <p>
    <%= f.submit „Utwórz” %>
  </p>
<% end %>
<%= link_to 'Powrót do strony głównej',
tickets_path %>
```

Jazda próbna

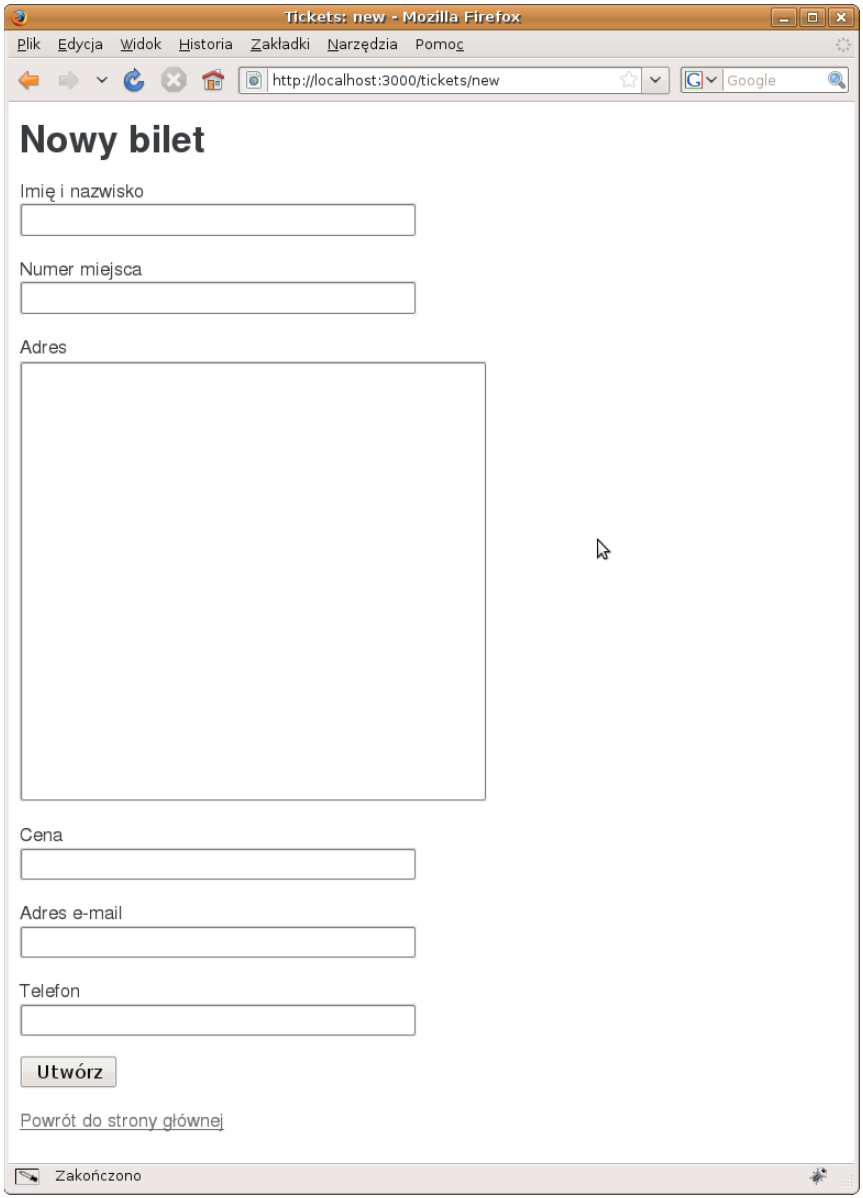
Po wprowadzeniu zmian odśwież stronę *http://localhost:3000/tickets*. Teraz możesz już sprzedać swoją aplikację właścicielowi jakiejś hali koncertowej w Polsce i nie martwić się o to, że zatrudnione tam osoby nie będą potrafiły jej obsługiwać!



Kontynuacja...



Kontynuacja...



Żeby nie wprowadzać dodatkowego zamieszania, wszystkie kolejne aplikacje w książce przedstawione będą w oryginalnych, angielskich wersjach językowych, w takiej formie, jak generowane są przez Rails. Nie powinno to jednak być dla Ciebie żadnym problemem — w końcu **wiesz już, co należy zmienić, by zmodyfikować język aplikacji Rails**!

Koncert jest wyprzedany!

Aplikacja przez cały tydzień działa idealnie, a w następny piątek wszystkie miejsca w hali koncertowej są wyprzedane.

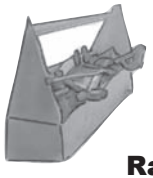
CELNE SPOSTRZEŻENIA

- Rails korzysta z architektury Model-Widok-Kontroler, zwanej również architekturą MVC — od angielskich odpowiedników tych terminów.
- Rails generuje osobne foldery dla kodu modelu, widoku oraz kontrolera.
- Wszystkie zmiany wprowadzane do aplikacji można zobaczyć natychmiast po ich zapisaniu i odświeżeniu strony w przeglądarce. To dzięki temu, że platforma Rails zbudowana jest w języku Ruby, a kod w tym języku nie musi być kompilowany.
- Zmiany w strukturze tabel można wprowadzić za pomocą migracji. By wygenerować migrację, która dodaje kolumnę do tabeli, użyj następującego polecenia:

```
ruby script/generate migration
Add<kolumna>To<tabela>
<kolumna>:<typ danych>
```

- Migrację wykonuje się za pomocą polecenia:

```
rake db:migrate
```



Niezbędnik programisty Rails

Masz za sobą rozdział 1. i teraz do swojego niezbędnika programisty Rails możesz dodać umiejętność tworzenia aplikacji Rails.

Narzędzia Rails

```
rails nazwa-aplikacji
        Tworzy aplikację

ruby script/server
        Uruchamia aplikację

ruby script/generate scaffold...
        Generuje jądro CRUD dla modelu

ruby script/generate migration
        Generuje migrację zmieniającą strukturę bazy danych

rake db:migrate
        Wykonuje nową migrację na bazie danych
```