STUDIA INFORMATICA 2010
Volume 31 Number 2B (90)

Jacek GRUBER, Adam WALOCHA
Politechnika Wrocławska, Instytut Informatyki

BAZA DANYCH W NOWOCZESNEJ APLIKACJI INTERNETOWEJ

Streszczenie. W artykule przedstawiono sposoby pracy z bazą danych we współczesnych frameworkach do wytwarzania aplikacji internetowych na przykładzie platformy deweloperskiej Ruby on Rails. Praca nawiązuje również do metodyk stosowanych przy tworzeniu oprogramowania w podobnych środowiskach.

Słowa kluczowe: aplikacja internetowa, Ruby on Rails, framework

DATABASE IN MODERN WEB APPLICATION

Summary. A set of methods to developing and to manage databases in modern web application frameworks on the example of Ruby on Rails are presented in this article. Also, the paper raises the subject of methodologies used in the process on such a kind of frameworks.

Keywords: web application, Ruby on Rails, framework

1. Wstęp

Dynamiczny rozwój Internetu na przestrzeni ostatnich 20 lat zapoczątkował erę aplikacji internetowych. Początkowo witryny WWW składały się tylko z treści statycznych i w gruncie rzeczy, nie zasługiwały na miano aplikacji. Wraz z rozwojem Internetu ewoluowały potrzeby użytkowników - hipertekst przestał wystarczać. Początkowo generowanie stron było implementowane w językach programowania. Zaowocowało to powstaniem standardu CGI komunikacji serwerów WWW z zewnętrznymi aplikacjami generującymi kod HTML. Równolegle powstała pierwsza wersja interpretera PHP zaimplementowana w języku Perl na potrzeby własne przez duńskiego programistę Rasmusa Lerdorfa. PHP zyskał popularność dzięki rozwijaniu go przez dwóch innych znanych programistów. Przez ten czas PHP zyskał ogromną popularność, ale jest on nadal jedynie specjalizowanym językiem skryptowym

używanym do generowania stron internetowych. Za kolejny krok w ewolucji wytwarzania aplikacji internetowych można uznać powstanie frameworków do wytwarzania aplikacji internetowych. Same specjalizowane języki nie dostarczały programistom gotowych rozwiązań w postaci wzorców projektowych czy implementacji funkcjonalności niezbędnych do pracy każdej witryny internetowej.

W wielu frameworkach kładzie się szczególny nacisk na bazę danych. Wynika to z prostej obserwacji, że właśnie w bazie danych zawarta jest większość semantyki i logiki biznesowej większości aplikacji internetowych. Aplikacje internetowe stanowią właściwie mniej lub bardziej złożoną fasadę pomiędzy przeglądarką a bazą danych. Fakt ten został dostrzeżony przez autorów frameworków, co zaowocowało rozwiązaniami wspomagającymi wytwarzania ukierunkowanych na bazę danych aplikacji internetowych. Standardem w świecie frameworków webowych są obecnie zintegrowane mapery relacyjno-obiektowe, mechanizmy wersjonowania schematu bazy danych oraz generatory interfejsu użytkownika na podstawie schematu danych. Jednym z najpopularniejszych frameworków do zorientowanego na bazę danych wytwarzania aplikacji internetowych jest Ruby on Rails. Na przykładzie tego środowiska wytwórczego przedstawimy dalej nowoczesne podejście do wytwarzania i projektowania aplikacji webowych.

2 . Ruby on Rails

Ruby on Rails (w skrócie RoR) jest przestrzenią projektową (ang. *framework*) służącą do szybkiego wytwarzania aplikacji internetowych. Jak każdy famework, RoR zapewnia szkielet przyszłego projektu. Szkielet aplikacji webowej jest w przypadku RoR oparty na wzorcu model-widok-kontroler (ang. *Model-View-Controller*). Dzięki temu można skupić się na programowaniu logiki biznesowej aplikacji, gdyż zastosowanie wzorca MVC we frameworku zapewnia nie tylko abstrakcyjny model aplikacji internetowej, lecz także stawia deweloperowi dwa inne ważne składniki: mechanizm automatycznego testowania oraz dedykowany serwer http. Framework izoluje programistę od powtarzających się w każdym projekcie webowym technicznych szczegółów implementacyjnych, czego przykładem może być sposób przesyłania danych pomiędzy przeglądarką a serwerem. Najbardziej istotnym elementem środowiska RoR z punktu widzenia oszczędności nakładów pracy jest zintegrowany mapper obiektowo-relacyjny (ORM). Również wybór języka nie jest tutaj przypadkowy. Języki statyczne nie sprawdzają się w wytwarzaniu systemów i aplikacji WWW. Języki te powstały w okresie, gdy większy nacisk kładziono na wydajność niż funkcjonalność. W dobie dynamicznego rozwoju technologicznego szybkość komputera przestaje być wąskim gardłem w procesie wytwarzania oprogramowania, a zaczyna być nim człowiek. Zjawisko to powo-

duje zwiększenie zainteresowania językami dynamicznymi, które bez wątpienia ustępują wydajnością statycznym, umożliwiając jednak sprawniejsze wytwarzanie oprogramowania. Wzrost produktywności programisty rekompensuje z nadwyżką spadek wydajności używanego języka programowania i procesu wytwórczego. Wybór dynamicznego języka interpretowanego Ruby jest więc odpowiedni zarówno z punktu widzenia klasy problemu wspomaganego przez framework, jak i z perspektywy trendów współczesnej inżynierii oprogramowania. Reasumując, Ruby on Rails jako framework zapewnia programiście:

- szablon projektu zgodny z wzorcem MVC,
- szkielet aplikacji generowany automatycznie,
- zaimplementowane i zintegrowane mechanizmy niezbędne do pracy każdej większej aplikacji webowej (maper relacyjno-obiektowy, obsługę sesji i komunikacji klient-serwer, abstrakcyjne klasy bazowe dla każdej z warstw MVC),
- automatyczne testy.

Biorąc pod uwagę ilość dostępnych środowisk programistycznych umożliwiających wytwarzanie aplikacji internetowych, zasadne wydaje się pytanie o sens budowania kolejnego. Autorzy RoR postanowili wyciągnąć wnioski z poprzednich projektów i ich wad. Określony został zestaw następujących założeń, które znacząco upraszczają i przyspieszają wytwarzanie aplikacji webowych:

- **Konwencja nad konfiguracją**

  Konfiguracja w tym kontekście nie oznacza tylko ustawień takich, jak ścieżki do plików czy parametry używane do łączenia z bazą danych, ale również sposób nazywania tabel w bazie danych czy obiektów biznesowych. Używanie domyślnego nazewnictwa na każdym poziomie, od bazy danych do plików z kodem HTML, uwalnia programistę od uciążliwego zestawiania komunikacji pomiędzy kolejnymi warstwami. Świetnym przykładem jest tutaj wbudowane narzędzie mapujące relacyjne tabele na obiekty w warstwie kontrolera. Przestrzegając konwencji programista, nie pisząc ani jednej linii kodu, otrzymuje z tabel bazy danych działający model obiektowy dostępny w wyższych warstwach aplikacji. Istnieje też możliwość wprowadzenia własnej konwencji nazewniczej, co jednak w praktyce zmusza do napisania własnej konfiguracji, tak jak się to dzieje w innych środowiskach, np. Javie.

- **Gotowy wzorzec**

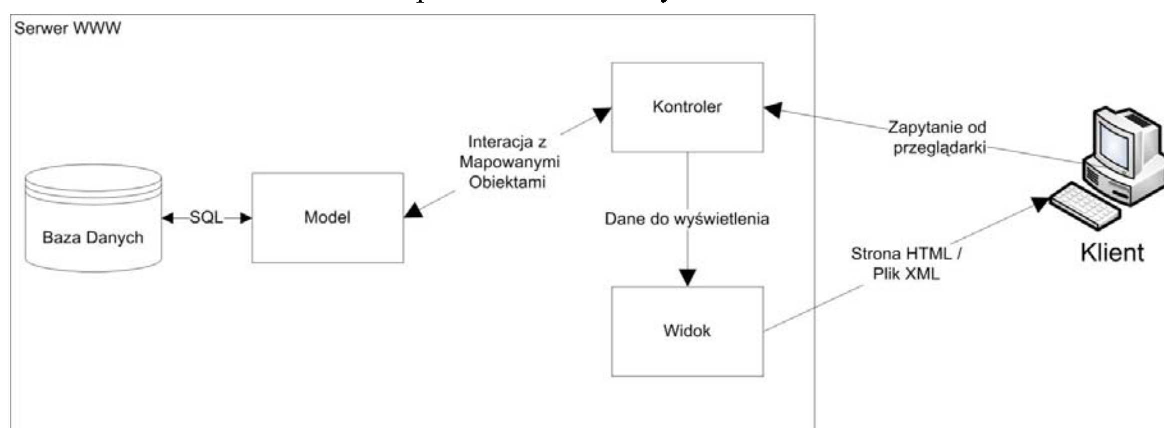
  Każdy doświadczony programista wie, że nie trzeba po raz kolejny odkrywać przysłowiowego koła. Znakomita większość problemów projektowych czy też architektonicznych została już kiedyś rozwiązana i spisana w postaci wzorców. Analiza problemu najczęściej sprowadza się do dopasowania do niego jednego ze wzorców i odpowiedniej adaptacji tego wzorca. Na poziomie architektonicznym dla znakomitej większości projektów aplikacji internetowych optymalnym wzorcem jest MVC. W RoR z założenia tworzy

się aplikacje zgodne właśnie z tym wzorcem. Pozorne ograniczenie w praktyce daje wiele korzyści. Przede wszystkim programista uzyskuje bez nakładu pracy całościowy szkielet aplikacji z mechanizmami wspierającymi współpracę elementów i izolującymi te elementy. Uściślenie charakteru (typu) tworzonych aplikacji umożliwia szersze wprowadzenie generowania kodu. Programiście pozostaje jedynie uzupełnienie aplikacji o logikę biznesową.

- **Nie powtarzaj się (DRY – don't repeat yourself)**
  Powtarzający się kod, poza wzrostem nakładów pracy, wprowadza nieporządek. Błąd w zduplikowanym kodzie wymaga nie tylko poprawienia, ale propagacji nowej wersji kodu na pozostałe wystąpienia.

Architektura

Wytwarzanie aplikacji webowych w Ruby on Rails, bazujące na wzorcu projektowym MVC, polega na zbudowaniu aplikacji składającej się z trzech części: modelu, kontrolera i widoku. Schemat wzorca MVC przedstawiono na rysunku 1.



Rys. 1. Schemat wzorca MVC
Fig. 1. MVC architectural pattern schematic diagram

Najważniejsze składowe wzorca to:
Model – część kodu odpowiedzialna za dostęp do danych biznesowych, czyli najczęściej za komunikację z bazą danych. Dodatkowo model realizuje walidację danych przed zapisem do bazy. Należy podkreślić, że nie jest to duplikowanie roli ograniczeń bazodanowych. Model dostarcza bardziej złożonych mechanizmów sprawdzania poprawności danych. Poza tym, jeśli umieścimy w nim pełną walidację, tworzony system przestaje być zależny od implementacji bazy danych w docelowym systemie zarządzania bazami danych.
Kontroler – część aplikacji zapewniająca sterowanie oraz przetwarzanie logiki biznesowej. Kontroler wykorzystuje model jako źródło danych oraz widok jako interfejs z użytkownikiem bądź aplikacją kliencką. Odpowiada za autoryzację, obsługę sesji oraz przepływ danych.

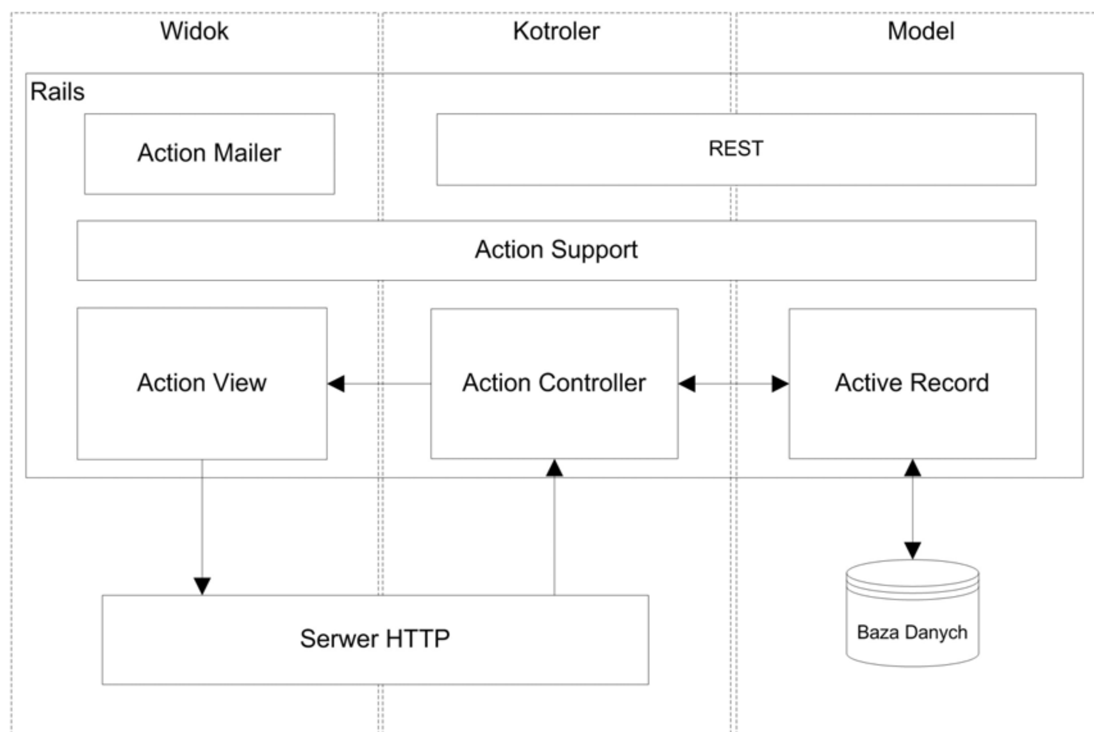
Widok – kod odpowiadający za prezentację danych. Otrzymane od kontrolera dane formatuje i przekazuje w wybranym formacie odbiorcy, niekoniecznie przeglądarki internetowej. Poza kodem HTML, wynikiem jego pracy może być między innymi XML.

Środowisko Rails składa się z sześciu modułów. Na rysunku 2 pokazano schemat modułów z uwzględnieniem ich roli z punktu widzenia MVC. Są to:

Action Record – moduł udostępnia w pełni obiektową warstwę dostępu do danych. Mapuje tabele z bazy danych udostępniając obiektowy interfejs dla wyższych warstw. Umożliwia walidację danych oraz kontroluje powiązania między obiektami.

Action View – moduł generujący kod HTML, XML lub inny używany do komunikacji z klientem. Obsługuje 3 rodzaje szablonów:

- RHTML/ERB
- RXML/Builder - obsługa XML i RSS
- RJS - szablony Javascript i Ajax



Rys. 2. Schemat modułów Rails
Fig. 2. Rails system modules

Action Support – biblioteki współdzielone przez wszystkie moduły.

Action Resource (REST) – moduł implementuje architekturę REST (ang. Representational State Transfer).

Action Controler – moduł realizuje zaimplementowaną logikę biznesową, ale zawiera również wbudowaną obsługę sesji, system buforowania, obsługi adresów URL (routing).

Action Mailer – moduł umożliwiający automatyczne generowanie i wysyłanie listów elektronicznych.

3 . Realizacja warstwy danych

Warstwa modelu w Ruby on Rails realizowana jest przez moduł Active Record. Do najważniejszych udostępnianych przez niego funkcjonalności należą:

- maper Relacyjno-obiektowy
- mechanizm migracji umożliwiający zarządzanie zmianami w bazie danych,
- walidacja i obsługa zdarzeń.

Tabele z bazy danych są mapowane na obiekty języka Ruby. Klasy tych obiektów dziedziczą po klasie ActiveRecord::Base, a ich nazwy są, zgodnie z konwencją, nazwami mapowanych tabel w liczbie pojedynczej. Deklarację klasy odwzorowującej tabelę o nazwie „products" prezentuje kod:

```
class Product < ActiveRecord::Base
  belongs_to :category
  has_many :orderdetails
  validates_presence_of :name, :message=> " is obligatory"
  validates_numericality_of :price
end
```

Kod zawiera dodatkowo definicję powiązania oraz walidacji, które zostały opisane szerzej w dalszej części artykułu. Klasy nazywane są często modelami, co niestety, wprowadza nieco zamieszania i powoduje niejasności, bo nazwa model pokrywa się z nazwą warstwy model. Do tworzonych w opisany sposób klas automatycznie dodawane są definicje pól. Ich nazwy odpowiadają nazwom kolumn mapowanej tabeli.

Łączenie z bazą

Jedną z korzyści ze stosowania mapowania relacyjnej bazy danych na model obiektowy jest zgrabne ukrycie logiki związanej z obsługą połączeń. Programista nie musi zajmować się powtarzającym się otwieraniem i zamykaniem połączenia lub własną implementacją warstwy obsługującej ten poziom komunikacji z bazą danych. W Ruby on Rails parametry połączenia należy podać tylko raz, w pliku konfiguracyjnym *database.yml* zgodnym z formatem *yaml*. Resztą zajmuje się Active Record. W pliku konfiguracyjnym znajdują się informacje o wszystkich bazach wykorzystywanych w tworzonej aplikacji. Domyślnie proces wytwarzania oprogramowania odbywa się z wykorzystaniem trzech baz danych: produkcyjnej, rozwojowej i testowej. Pierwsza z nich pracuje z włączonym buforowaniem i jest wersją bazy dedykowaną dla środowiska produkcyjnego. Druga jest wykorzystywana przez programistów podczas właściwego tworzenia aplikacji, a ostatnia służy wykonywaniu testów. Moment synchronizacji bazy produkcyjnej i testowej ze źródłową produkcyjną wybiera programista. Oprócz tego ActiveRecord wspiera wykorzystywanie przez aplikację wielu baz danych. Każdy zmapowany obiekt może być połączony z inną bazą danych, co jednak wymaga jawnego wskazania połączenia do bazy danych w definicji modelu.

Migracje

Problem wersjonowania podczas prac z bazą danych spędza sen z powiek wielu programistom. Praktycznie każdy komercyjny projekt wymaga pracy z bazami danych, szczególnie gdy jest wykonywany przez kilkuosobowe zespoły. Standardowe systemy kontroli wersji, takie jak SVN czy TFS, zostały zbudowane z myślą o pracy z kodem źródłowym, a nie z relacyjnymi bazami danych. Ruby on Rails łagodzi problemy wersjonowania dzięki koncepcji i mechanizmowi migracji. Koncepcja ta zakłada, że każda zmiana schematu relacji jest zapisywana oddzielnie, umożliwiając zarówno dokonanie tej zmiany, jak i jej wycofanie. Naturalnie, pojedyncza zmiana nie musi być pojedynczą operacją bazodanową. W Ruby zmiany nie są zapisywane w postaci skryptów w języku zależnym od typu bazy, lecz jako klasy w Ruby. Klasa ta zawiera dwie metody – up i down. Pierwsza z nich odpowiada za wprowadzenie zmian do schematu a druga za ich wycofanie. ActiveRecord udostępnia definicję schematu bazy za pomocą języka Ruby, dzięki czemu migracje można wykonać na bazach różnych typów, operujących różnymi dialektami języka SQL. Migracje pozwalają zatem wersjonować strukturę bazy danych oraz udostępniają zapis tej struktury abstrahując od rodzaju bazy danych. Poniższy kod przedstawia migrację tworzącą dwie tabele, ich atrybuty oraz powiązanie między tabelami.

```
class CreateProduct < ActiveRecord::Migration
  def self.up
    create_table :categories do |t|
      t.column :name, :string,:null => false
      t.column :description, :string
    end
    create_table :products do |t|
      t.column :name, :string,:limit=>50,:null => false
      t.column :price, :float
      t.column :manufacturer,:string,:limit=>50
      t.column :discount, :float
      t.column :available, :boolean,:default=>true,:null => false
      t.references :category
    end
  end

  def self.down
      drop_table categories
      drop_table products
  end
end
```

Poza transformacjami widocznymi w przykładzie, ActiveRecord implementuje praktycznie wszystkie transformacje dostępne w standardzie języka SQL dla operacji DDL. Dodatkowo framework zajmuje się sam zakładaniem kluczy głównych, co nie tylko skraca kod, ale ułatwia utrzymywanie konwencji. Wykonanie migracji polega na wykonaniu polecenia „Rake”, które jako argument pobiera między innymi wersję docelową struktury bazy. Numer aktualnej oraz wszystkich wcześniejszych wersji schematu bazy danych przechowywany jest w specjalnej tabeli o nazwie schema_migrations.

Powiązania

W przeciwieństwie do pól, w języku Ruby związki pomiędzy tabelami nie są przenoszone na model obiektowy w całości automatycznie. Jeżeli klucze obce w tabelach zostały utworzone poprzez migrację lub po prostu programista przestrzegał konwencji, to do utworzenia powiązania trzeba tylko dodać w klasach modeli biorących udział w powiązaniu informacje o typie tego związku. Naturalnie, można jawnie podać nazwy kluczy obcych, co nie wprowadza nadmiarowości biorąc pod uwagę, że i tak trzeba ręcznie wprowadzić do definicji klas jeszcze inne parametry. Przykłady 1 i 2 ilustrują sposób deklarowania różnych rodzajów powiązań w modelu. W pierwszym przykładzie struktura bazy nie jest zgodna pod prawie każdym względem z konwencją Ruby on Rails. Należy więc podać jawnie klucze obce i główne. Z kolei przykład 2 przedstawia schemat bazy danych zgodny z nazewnictwem. W bazie musi, oczywiście, istnieć fizycznie tabela pośrednicząca przechowująca powiązania między tabelami powiązanymi związkami wiele do wielu. W przykładzie 2 nosi ona nazwę *products_offers* i zawiera trzy kolumny, dwa klucze obce i własny klucz główny. Poza przedstawionymi istnieją jeszcze dwa inne rodzaje powiązań – jeden do jeden oraz polimorficzne.

Przykład 1

```
class Product < ActiveRecord::Base
    set_table_name "Produkt"
    primary_key "ident"
    belongs_to :category, :foreign_key => 'katIdent'
end

class Category < ActiveRecord::Base
    set_table_name "Kategoria"
    primary_key "ident"
    has_many :products, :foreign_key=> 'katIdent'
end
```

Często zdarza się, że kilka tabel w projektowanej bazie danych posiada kilka powiązań jednakowego typu z pewną inną tabelą. Przykładem może być następująca reguła. Encje pracownik, firma i klient są w związku jeden do wielu z encją adres. Postępując analogicznie do przykładu 1, należałoby utworzyć trzy oddzielne tabele adresów. Powiązania polimorficzne pozwalają tego uniknąć. Umożliwiają one definiowanie powiązań jednej tabeli (w tym przypadku tabeli adres) z kilkoma tabelami na raz. Wymaga to, naturalnie, dodania w tabeli adres kolumny identyfikującej, z którymi i jakiego typu kolumnami w innych tabelach powiązany jest każdy adres. Zgodnie z konwencją, należy dodać dwie kolumny o nazwach: addressable_id i addressable_type, gdzie „addressable” to dowolnie wybrana nazwa powiązania polimorficznego. Zastosowanie w tym miejscu polskiej nazwy jest możliwe, ale niestety, będzie ona spluralizowana zgodnie z zasadami języka angielskiego.

Przykład 2

```
class Product < ActiveRecord::Base
    has_and_belongs_to_many :offers
end

class Offer < ActiveRecord::Base
    has_and_belongs_to_many :products
end
```

Implementacja powiązania w modelach adresu i pracownika wygląda następująco:

```
class Adres < ActiveRecord::Base
    belongs_to :addressable,
               :polymorphic =>true
end

class Pracownik < ActiveRecord::Base
    has_many :adreses,
             :as => :addressable
End
```

Ostatnim rodzajem związków jest dziedziczenie. Możliwe są trzy rodzaje realizacji dziedziczenia w bazie danych:

- jedna tabela dla wszystkich klas,
- oddzielne tabele dla każdej klasy,
- oddzielne tabele dla klas konkretnych.

Pierwszy sposób jest wspierany przez ActiveRecord poprzez dziedziczenie modeli:

```
class Osoba < ActiveRecord::Base
end

class Pracownik < Osoba
end
```

Niestety, metoda ta wprowadza nadmiarowość, ponieważ każdy rekord w bazie musi posiadać tyle kolumn, ile razem unikalnych pól posiadają wszystkie klasy w hierarchii. Druga metoda może zostać zaimplementowana po stronie bazy danych, ale mapowane klasy pochodne w Ruby nie będą klasami pochodnymi, a tylko klasami powiązanymi z pseudo-klasą bazową za pomocą powiązania jeden do jednego. Oczywiście, związek ten trzeba zdefiniować w modelu ręcznie.

Korzystanie z modeli

ActiveRecord udostępnia dla obiektów i klas mapowanych z bazy danych wszystkie podstawowe operacje, czyli odczyt, tworzenie, modyfikowanie, usuwanie i wyszukiwanie. Większość tych operacji można wykonywać zarówno bezpośrednio na obiektach, jak i poprzez metody statyczne klas.

Alternatywne sposoby edycji rekordu są następujące:

```
person = Person.find(1)            Person.update 1, :name= Adam
person.name= Adam
person.save
```

Każdy wczytany obiekt przechowuje flagę z modyfikacją, dzięki czemu wiadomo, czy został on zmodyfikowany, a co za tym idzie, czy należy go zapisać. Służą do tego metody „changed" i „changes". Ta pierwsza zwraca listę zmodyfikowanych pól, a druga dodatkowo dołącza wartości tego pola przed i po zmianie. Usuwanie rekordu można przeprowadzić na dwa sposoby – z pominięciem walidacji na poziomie ActiveRecord za pomocą metody „delete" oraz w kontrolowany sposób z wykorzystaniem metody „destroy". Pierwsza metoda powoduje wyłącznie wykonanie operacji DELETE na bazie danych, druga z kolei przeprowadza walidacje oraz może rzucać wyjątki, na przykład w przypadku próby usunięcia nieistniejącego obiektu. Rozbudowany jest również odczyt rekordów, co nie jest zaskoczeniem, gdyż jest to operacja najczęściej wykonywana. Programista ma do dyspozycji dwa sposoby wczytywania danych – zapytanie SQL i interfejs obiektowy. Kwerendę SQL wykonuje się podając ją jako parametr do statycznej metody modelu „find_by_sql". Na tym samym poziomie dostępna jest metoda find oraz jej dynamiczne odpowiedniki. Wyszukiwanie za pomocą operacji obiektowych jest możliwe na wiele sposobów, np.:

- po numerze id: person = Person.find **1**
- metodą dynamiczną: person = Person.find_by_name 'Adam'
- wiele rekordów na raz: persons = Person.find [**1**,**2**,3]
- według warunku: persons = Person.find(**:all**, **:conditions** => ['birth_date > ? ', '1990'])

Dodatkowo istnieje możliwość podania metodzie find następujących parametrów:

- `:order`- określa sposób sortowania,
- `:limit`- ilość maksymalnie wczytanych rekordów,
- `:offset` - określa od którego elementu zacząć pobierania wierszy (występuje tylko z :limit),
- `:include` - złączenie zewnętrzne,
- `:joins`- złączenie wewnętrzne,
- `:select` - określa jakie kolumny mają zostać pobrane,
- `:readonly` - uniemożliwia modyfikację wczytanego z tym parametrem obiektu,
- `:group`- grupowanie.

Jak widać, operowanie na modelu odbywa się w przejrzysty sposób zgodnie z podejściem obiektowym. Framework dodatkowo oferuje wiele zaawansowanych opcji. Jedną z nich są transakcje. Użycie transakcji prezentuje blok kodu realizujący przelew pewnej kwoty (sum) z jednego konta na inne:

```
begin
     Account.transaction(accountFrom, accountTo) do
        accountFrom.transfer(-sum)
        accountTo.transfer(sum)
     end
rescue
     flash[:error] = "Error! - insufficient funds!"
     render :action => "add_items"
     return
end
```

Metoda „transfer" rzuca wyjątek, jeżeli niemożliwe jest dodanie kwoty podanej w parametrze. Blok o nazwie rescue jest wykonywany, jeśli dojdzie do błędu podczas wykonania operacji objętych transakcją i jej wycofania.

Kolejne ważne zagadnienie to sposób ładowania rekordów powiązanych. Domyślnie Ruby on Rails wykorzystuje tzw. leniwe wczytywanie, które polega na pobieraniu powiązanych rekordów z bazy dopiero podczas pierwszego ich użycia. Podejście to przyspiesza pracę, o ile aplikacja nie żąda pojedynczo dużej ilości powiązanych obiektów. W takiej sytuacji warto wykorzystać „skwapliwe" ładowanie (ang. *eager loading*). W tym rozwiązaniu programista wymusza załadowanie konkretnych obiektów pozostających w związku z danym obiektem, podczas pierwszego dostępu do niego. Sposób użycia skwapliwego ładowania polega na dołączeniu do metody wczytującej find parametru :include oraz podaniu jako parametru tablicy nazw powiązań, dla których ma zostać użyty ten typ dostępu.

Przy projektowaniu aplikacji opartej na bazie danych i języku SQL trzeba mieć na uwadze niebezpieczeństwo ataków SQL Injection, czyli wstrzykiwanie niebezpiecznego kodu SQL, gdy niewalidowane dane są wprowadzane do pól edycyjnych i adresu URL po stronie klienckiej. W przypadku Rails wystarczy pamiętać o korzystaniu z zapytań parametryzowanych. Kod autoryzujący użytkownika na dwa sposoby – bezpieczny (u góry) i podatny na atak (poniżej):

```
User.find(:all,:conditions=> ["login=? AND password=?",login,password])
User.find(:all,:conditions=>"login='#{login}' AND password='#{password}'")
```

Walidacje

Zapisywane w aplikacji internetowej dane w przeważającej części pochodzą od użytkownika, co wymusza sprawdzanie poprawności tych informacji. ActionRecord udostępnia w tym celu system walidacji, zawierający kilkanaście predefiniowanych kryteriów. Dodanie walidacji wygląda w kodzie podobnie jak dodanie powiązania. Jest to metoda modelu pobierająca jako parametr nazwę walidowanego pola oraz dodatkowe informacje, jeśli są potrzebne do weryfikacji poprawności, np. wyrażenie regularne. Poniższy model zawiera walidację z jednym, jak i walidację z kilkoma parametrami.

```
class Person < ActiveRecord::Base
  validates_numericality_of :age
  validates_inclusion_of :sex,
      :in => %w( Male Female ),
```

```
      :message=>"should be: Male Female"
  validates_format_of :email, :with =>
    /^([^@\s]+)@((?:[-a-z0-9]+\.)+[a-z]{2,})$/i
end
```

Jeżeli któryś z warunków nie zostanie spełniony, dane nie są zapisywane, a do widoku zostanie przesłana informacja o błędzie. Do jej wyświetlenia służy funkcja „error_messages_for", której parametrem jest nazwa modelu, dla którego funkcja ma pokazać błędy walidacyjne.

4. Proces wytwarzania aplikacji internetowej w Ruby on Rails

Obecnie przedstawimy próbę uchwycenia i opisu procesu wytwarzania aplikacji internetowych w sposób przyrostowy i inkrementacyjny przy wykorzystaniu środowiska Ruby on Rails. W proponowanej metodyce, zgodnie z modelem przyrostowym, pierwszym krokiem procesu jest inicjalizacja. Polega ona na wstępnym określeniu wymagań dla całego projektu. Dokładnie planowana jest tylko pierwsza iteracja. Pojedyncza iteracja realizowana jest według modelu kaskadowego przystosowanego do specyfiki frameworka i zwinnego podejścia do wytwarzania oprogramowania. Poniżej zostanie opisana typowa iteracja, w której realizuje się lub rozszerza pewne przypadki użycia, czyli ujmując to ogólnie, zaspokaja się wymagania funkcjonalne.

Iteracja

Większość iteracji podczas tworzenia aplikacji internetowej w Ruby on Rails przebiega bardzo podobnie. Zbierane są pewne życzenia klienta, przekładane są one na wymagania funkcjonalne, ewentualnie są one wyrażane za pomocą modeli, a następnie implementowane. Poniżej zostały przedstawione standardowe kroki jednej iteracji.

Zebranie wymagań

Pierwszym krokiem tej fazy jest określenie zakresu dla iteracji na podstawie zarysu planu przygotowanego podczas inicjalizacji projektu. Po zdefiniowaniu fragmentu systemu, który ma być zbudowany w tej iteracji, buduje się lub modyfikuje istniejące przypadki użycia. W obu czynnościach powinni brać aktywny udział przedstawiciele klienta. Forma, w jakiej tworzone są przypadki użycia, zależy od kilku czynników – wielkości projektu, preferencji klienta czy nawyków programistów. Dla małych projektów, w których dokumentacja nie jest klientowi niezbędna, może wystarczyć odręczny rysunek lub prosta specyfikacja będąca notatkami z rozmowy z przedstawicielem klienta. Diagramy przypadków użycia budowane są też za pomocą modelerów zawartych designerach, np. w Sybase PowerDesigner. Zwiększa to czytelność, a zarazem może być udostępnione innym członkom zespołu realizującego pro-

jekt. Należy pamiętać, że diagram ma być pomocny w dalszych etapach iteracji i dlatego nie może on stanowić ostatecznego kontraktu, utrudniającego reagowanie na zmiany w wymaganiach na wytwarzany system. Wybrana forma specyfikacji przypadków użycia jest finalnym artefaktem etapu zebrania wymagań.

Projektowanie

Projekt aplikacji internetowej obejmuje najczęściej dwa najbardziej złożone elementy – dane i hipertekst. Podobnie jak w przypadku wymagań, sposób projektowania zależy od wielkości i złożoności projektu. Dla prostych portali projektowanie wcale nie musi być wykonywane i w praktyce często pomija się je. Projektowanie przebiega często w oparciu o uproszczone modele rysowane odręcznie. Dla przykładu, w [4] autor proponuje modelowania bazy danych w postaci odręcznego diagramu podobnego budowanego w UML. Hipertekst modeluje się z kolei tworząc także odręcznie diagram przepływu stron (ang. pages flow). Jeśli zachodzi potrzeba, można wykorzystać notację specjalizowaną dla aplikacji internetowych, na przykład WebML. Przy użyciu dedykowanego narzędzia WebRatio można opracować modele danych i hipertekstu. WebRatio umożliwia również projektowanie personalizacji aplikacji. Najczęściej nie modeluje się całej aplikacji, lecz jej najbardziej złożone fragmenty. Większość aplikacji internetowych zawiera wiele prostych stron, których projektowanie byłoby zwykłym marnowaniem czasu. Ogólnie, w programowaniu zwinnym zaleca się rozpoczynać projektowanie od najprostszych metod. Dopiero gdy okażą się niewystarczające, trzeba się zastanowić nad bardziej zaawansowanym podejściem projektowym.

Artefaktami tej fazy mogą być zatem odręczne diagramy bazy danych i hipertekstu, diagramy w notacji specjalizowanej lub w szczególności tylko przemyślenia programisty.

Implementacja

Proces implementacji w Ruby on Rails jest mocno zautomatyzowany dzięki generatorom kodu. Dodanie nowej encji sprowadza się do jednego wywołania generatora „scaffold”. Stworzy on plik migracji, widoki, kontroler i model. Jeśli dodatkowo poda się podczas generacji listę pól encji, to plik migracji będzie zawierał już ich definicje. Widoki zostaną uszczegółowione polami encji. Kolejny krok to dokończenie pliku migracji tak, aby plik ten odpowiadał wymaganiom, w szczególności wyrażonym w postaci diagramu danych. Powiązania nie są automatycznie generowane, więc należy je dodać ręcznie. Po zakończeniu definiowania migracji należy ją uruchomić. W tym momencie tworzony wycinek aplikacji jest już uruchamialny. Generator tworzy domyślny zbiór hipertekstów umożliwiający operacje CRUD na obiektach encji, dla której był uruchomiony. Na tym etapie należy zaimplementować własną logikę biznesową w kontrolerach oraz zmodyfikować strukturę hipertekstu według własnych potrzeb. Generator tworzy również szkielety testów jednostkowych.

Iteracja może również wymagać modyfikacji istniejących modeli, kontrolerów lub widoków. Także w tym przypadku okazują się przydatne generatory kodu. Można dzięki nim wygenerować dodatkowe metody kontrolera wraz z widokami. Jeśli zmiana dotyczy struktury (schematu) bazy danych, generator może przebudować również widoki.

Weryfikacja

W ostatniej fazie iteracji wynik prac prezentowany jest przedstawicielowi klienta. Nie trzeba przygotowywać specjalnej wersji oprogramowania. Aplikacja jest gotowa do prezentacji praktycznie w każdej chwili. Uwagi klienta do zaprezentowanej wersji aplikacji wpłyną na kolejną iterację.

5. Podsumowanie

W artykule przedstawiono sposób pracy z bazą danych w nowoczesnych frameworkach do wytwarzania aplikacji internetowych na przykładzie środowiska Ruby on Rails. Mechanizmy mapowania relacyjno-obiektowego wydatnie skracają i usprawniają wytwarzanie aplikacji internetowych. Równie przydatne okazują się generatory kodu, dzięki którym programista może skupić się na projekcie danych i implementacji właściwej logiki biznesowej. Wbudowany mechanizm wersjonowania schematu bazy danych ułatwia z kolei wprowadzanie kolejnych zmian do projektu.

Zaprezentowany został również proces wytwarzania oprogramowania w tym frameworku, wykorzystujący możliwie najszerzej potencjał Ruby on Rails.

BIBLIOGRAFIA

1. Orsini R.: Rails Cookbook.
2. Oficjalna dokumentacja środowiska: http://guides.rubyonrails.org
3. Zabiełło J.: Ruby on Rails 2.1. Tworzenie nowoczesnych aplikacji internetowych.
4. Ruby S., Thomas D., Hansson D.H.: Pragmatic - Agile Web Development with Rails.3rd Edition.
5. Ceri S., Fraternali P., Bongio A., Brambilla M., Comai S., Matera M.: Designing data-intensive web applications.
6. Boehm B., Turner R.: Balancing Agility and Discipline: A Guide for the Perplexed. Boston, MA: Addison-Wesley, 2004. Appendix A, s. 165÷194.

Recenzenci: Dr inż. Paweł Kasprowski
Dr inż. Hafed Zghidi

Wpłynęło do Redakcji 30 stycznia 2010 r.

Abstract

Several methods to work with database in modern web application frameworks on Ruby on Rails, as an example, are presented in this paper. In addition, the article raises the subject of methodologies used on such frameworks. In the first part were presented the main components of web application framework which improve the implementation of database layer. The most important of these components are: object-relational mappers, database schema versioning mechanisms and code generators. The second part of this paper explains, how to develop data driven web application using the power of web application frameworks. All presented examples base on one of the most popular web framework - Ruby on Rails.

Adresy

Adam WALOCHA: Politechnika Wrocławska, Instytut Informatyki, ul. Wybrzeże Wyspiańskiego 27, 50-370 Wrocław, adam@walocha.com.pl.
Jacek GRUBER: Politechnika Wrocławska, Instytut Informatyki, ul. Wybrzeże Wyspiańskiego 27, 50-370 Wrocław, jacek.gruber@pwr.wroc.pl.