

# EPI: Interfejs Graficzny 2011/2012

## Wykład nr 6

### Warstwa modelu – ActiveRecord

Aleksander Pohl

15 listopada 2011

# ActiveRecord – railsowy ORM

- ▶ implementacja wzorca *active record*:
  - ▶ **tabelom** bazy danych odpowiadają **klasy**
  - ▶ **wierszom** tabeli odpowiadają **obiekty**
  - ▶ **komórkom** tabeli odpowiadają **atrybuty** obiektu
  - ▶ niektóre tabele odpowiadają **związkom** między klasami (dla zależności m-do-n)
    - ▶ każdy rekord posiada podstawowy **klucz główny** id
- ▶ **migracje** – pozwalają modyfikować schemat bazy danych bez użycia SQL
- ▶ **walidacje** – pozwalają weryfikować poprawność danych przed ich zapisaniem
- ▶ **zapytania** – zapewniają obiektowy dostęp do danych zgromadzonych w bazie
- ▶ **callbacks** – pozwalają wykonywać określone zadania stowarzyszone z cykle życia obiektu

# Sposób użycia

## ▶ definicja klasy

```
class Book < ActiveRecord::Base
end
```

## ▶ utworzenie nowego rekordu

```
book = Book.new(:title => 'Dziady')
book.save
```

## ▶ odczytanie rekordu

```
book = Book.find(1)
book.title      #=> 'Dziady'
```

## ▶ modyfikacja rekordu

```
book.title = "Dziady IV"
book.save
# lub
book.update_attributes(:title => "Dziady IV")
```

## ▶ usunięcie rekordu

```
book.destroy
```

# Konfiguracja bazy

## Plik config/database.yml

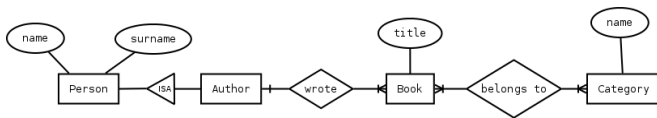
```
development:
  adapter: sqlite3
  database: db/development.sqlite3
  pool: 5
  timeout: 5000
test:
  adapter: sqlite3
  database: db/test.sqlite3
  pool: 5
  timeout: 5000
production:
  adapter: sqlite3
  database: db/production.sqlite3
  pool: 5
  timeout: 5000
```

# Generator modelu

```
rails generate model Book title:string
```

```
  invoke  active_record
  create  db/migrate/20111108182152_create_books.rb
  create  app/models/book.rb
  invoke  test_unit
  create  test/unit/book_test.rb
  create  test/fixtures/books.yml
```

# Model konceptualny a model obiektowy



```
class Person < ActiveRecord::Base
end

class Author < Person
  has_many :books
end

class Book < ActiveRecord::Base
  belongs_to :author
  has_and_belongs_to_many :categories
end

class Category < ActiveRecord::Base
  has_and_belongs_to_many :books
end
```

# Związek jeden-do-wiele po stronie wiele

```
class Book < ActiveRecord::Base
  belongs_to :author
end
```

Obiekty klasy 'Book' posiadają teraz następujące dodatkowe metody:

```
book.author
book.author=(author)
book.author?(some_author)
book.author.nil?
book.build_author(...)
book.create_author(...)
```

## Związek jeden-do-wiele po stronie jeden

```
class Author < ActiveRecord::Base
  has_many :books
end
```

Obiekty klasy 'Author' posiadają teraz następujące dodatkowe metody:

```
author.books
author.books<<(book)
author.books.delete(book)
author.books=[book1,book2,...]
author.book_ids
author.book_ids=[id1,id2,...]
author.books.clear
author.books.empty?
author.books.size
author.books.find(...)
author.books.exists?(...)
author.books.build(...)
author.books.create(...)
```

# Związek wiele-do-wiele

```
class Category < ActiveRecord::Base
  has_and_belongs_to_many :books
end
```

Obiekty klasy 'Category' posiadają teraz następujące dodatkowe metody:

```
category.books
category.books<<(book)
category.books.delete(book)
category.books=[book1,book2,...]
category.book_ids
category.book_ids=[id1,id2,...]
category.books.clear
category.books.empty?
category.books.size
category.books.find(...)
category.books.exists?(...)
category.books.build(...)
category.books.create(...)
```

# Konwencje bazy danych

1. każda tabela (z wyjątkiem złączeniowej) posiada sztuczny klucz główny o nazwie `id`
2. w bazie danych wszystkie tabele zaczynają się z małej litery i są w liczbie mnogiej, np. `authors`, `books`, `categories`
3. związek jeden-do-wiele:
  - 3.1 po stronie „jeden” nazwa związku w liczbie mnogiej, np. `author.books`, brak pola w bazie danych
  - 3.2 po stronie „wiele” nazwa związku w liczbie pojedynczej, np. `book.author`, nazwa pola w bazie: nazwa powiązanego modelu + `_id`, np. `author_id`

## Konwencje bazy danych – cd.

4. związek wiele-do-wiele:
  - 4.1 tabela złączeniowa: nazwa pierwszego modelu w liczbie mnogiej + nazwa drugiego modelu w liczbie mnogiej, kolejność nazw modeli zgodna z porządkiem alfabetycznym, np. `books_categories`
  - 4.2 pola tabeli złączeniowej: nazwa pierwszego modelu + `_id`, nazwa drugiego modelu + `_id`, np. `book_id`, `category_id`
  - 4.3 tabela złączeniowa nie musi posiadać klucza id, powinna posiadać klucz unikalny obejmujący oba klucze obce
5. relacja IS-A:
  - 5.1 realizowana jest poprzez mechanizm dziedziczenia
  - 5.2 wszystkie klasy hierarchii mają przypisaną jedną tabelę (STI)
  - 5.3 nazwa tabeli tworzona jest na podstawie klasy bazowej
  - 5.4 typ obiektu zapamiętywany jest w polu tekstowym `type`
  - 5.5 wszystkie klasy mają dostęp do tych samych atrybutów

## Opcje konfiguracyjne

- ▶ wszystkie domyślne ustawienia można zmienić, nie należy jednak robić tego bez powodu
- ▶ odstępstwa od konwencji określa się w klasie, np.:
  - ▶ **set\_table\_name**  
ustała odmienną nazwę tabeli
  - ▶ **belongs\_to :creator, :class\_name => "Author"**  
zmiana powiązanej klasy
  - ▶ **belongs\_to :author, :foreign\_key => "creator\_id"**  
zmiana klucza obcego
- ▶ pozostała ważniejsze opcje:
  - ▶ **:dependent** – zachowanie obiektu powiązanego przy usuwaniu obiektu nadrzędnego
  - ▶ **:polymorphic** – związek polimorficzny
  - ▶ **:through** – związek zapożyczony

# Motywacja

- ▶ pozwalają na przyrostowe budowanie schematu bazy danych
- ▶ upraszczają synchronizację schematu bazy danych w środowisku programisty i na zdalnym serwerze
- ▶ pozwalają dzielić się z innymi programistami zmianami, które muszą wprowadzić w swojej bazie
- ▶ pozwalają na wycofywanie wprowadzonych zmian
- ▶ automatycznie wykrywają, które zmiany muszą być wprowadzone w bazie
- ▶ pisane są w Rubim – są niezależne od konkretnej implementacji bazy danych

# Model konceptualny a migracje

```
class CreatePeople <
  ActiveRecord::Migration
  def self.up
    create_table :people do |t|
      t.string :name
      t.string :surname
      t.string :type
    end
  end

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

```
class CreateBooks <
  ActiveRecord::Migration
  def self.up
    create_table :books do |t|
      t.string :title
      t.references :author
      # to samo co
      # t.integer :author_id
    end
  end

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

## Model konceptualny a migracje – cd.

```
class CreateCategories < ActiveRecord::Migration
  def self.up
    create_table :categories do |t|
      t.string :name
    end
  end

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

## Model konceptualny a migracje – cd.

```
class CreateBooksCategories < ActiveRecord::Migration
  def self.up
    create_table :books_categories, :id => false do |t|
      t.references :book
      t.references :category
    end
    add_index :books_categories, [:book_id, :category_id], :unique => true
  end

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

## Pliki migracji – db/migrate/\*

- ▶ plik migracji opisuje zmianę schematu bazy danych
- ▶ 2 metody: **up** i **down** opisują zmianę schematu i jej odwrotność
- ▶ posiadają numer, który w Rails 2.x oraz 3.x jest generowany na podstawie czasu powstania pliku
- ▶ pusty plik migracji generowany jest przez polecenie:  
**rails generate migration migration\_name**
- ▶ plik migracji tworzony jest też automatycznie w trakcie tworzenia nowego modelu
- ▶ w obu poleceniach można dodać pary **nazwa\_pola:typ**, które zostaną automatycznie uwzględnione w migracji

## Wprowadzanie zmian w bazie

- ▶ `schema_info` jest tabelą w bazie danych posiadającą jedno pole tekstowe – `version`
- ▶ `schema_info` zawiera numery wszystkich wprowadzonych migracji
- ▶ polecenie `rake db:migrate` sprawdza numery w bazie danych i porównuje je z numerami migracji w katalogu `db/migrate`
- ▶ jeśli w bazie danych brakuje jakiś numerów migracji, są one aplikowane w kolejności odpowiadającej czasowi ich powstania
- ▶ w celu wycofania ostatniej zmiany wprowadzamy `rake db:rollback`  
powodzenie tego procesu zależy od poprawnej implementacji metody `down`

## Inne polecenia rake do zarządzania bazą

- ▶ **db:create** – tworzy bazę danych dla środowiska programisty
- ▶ **db:create:all** – tworzy bazę danych dla wszystkich środowisk
- ▶ **db:migrate:redo** – wycofuje i wprowadza ostatnią wprowadzoną migrację
- ▶ **db:migrate:reset** – czyści bazę i od początku przeprowadza wszystkie migracje
- ▶ **db:migrate:status** – wyświetla status poszczególnych migracji
- ▶ **db:seed** – wczytuje dane inicjujące z pliku db/seed.rb
- ▶ **db:version** – wyświetla numer ostatniej wprowadzonej migracji

# Motywacja

- ▶ deklaracyjny mechanizm weryfikacji poprawności wprowadzanych danych
- ▶ istotnie rozszerza zestaw więzów dostępnych w bazie danych
- ▶ zawiera zestaw predefiniowanych walidatorów, często wykorzystywanych w aplikacjach internetowych
- ▶ pozwala na weryfikację wartości tekstowych, liczbowych, logicznych, itp.
- ▶ pozwala na wyabstrahowanie bardziej skomplikowanych walidacji (np. adresu e-mail)
- ▶ umożliwia ograniczenie działania walidacji do tworzenia/modyfikowanie obiektu
- ▶ może być używany do obiektów niezwiązanych z bazą danych

# Przykład walidatorów

```
class Person < ActiveRecord::Base
  validates :shoe_size, :numericality => true
  validates :first_name, :presence => true,
    :length => {:maximum => 30}
  validates :user_name,
    :uniqueness => {:scope => :account_id},
    :length => {:within => 6..20,
      :too_long => "pick a shorter name",
      :too_short => "pick a longer name"}
  validates :email, :format => {
    :with => /\A(?:[^\s]+)@((?:[-a-z0-9]+\.)+[a-z]{2,})\Z/i,
    :on => :create
  }, :confirmation => {
    :message => "should match confirmation"
  }
  validates :terms_of_service, :acceptance => {:on => :create}
end
```

## Predefiniowane walidatory

- ▶ **presence** – sprawdzenie niepustości atrybutu
- ▶ **acceptance** – sprawdzenie wartości logicznej, np. zatwierdzenie regulaminu
- ▶ **confirmation** – porównanie identyczności dwóch atrybutów, np. hasła i jego powtórzenia
- ▶ **exclusion** – wykluczenie należenia do określonej grupy wartości
- ▶ **inclusion** – przynależność do określonej grupy wartości, np. płeć – mężczyzna/kobieta
- ▶ **format** – sprawdzenie formatu wartości tekstowej przy użyciu wyrażenia regularnego, np. kod pocztowy
- ▶ **length** – sprawdzenie długości wartości tekstowej
- ▶ **numericality** – sprawdzenie wartości numerycznej
- ▶ **uniqueness** – unikalność wartości, np. loginu
- ▶ **associated** – weryfikowanie powiązanego obiektu

# Sposób działania

```
person = Person.new()
if person.save
  #...
else
  person.errors
end

begin
  person = Person.new()
  person.save!
rescue ActiveRecord::RecordInvalid
  # obsługa błędu
end
```

## Proste zapytania

- ▶ za pomocą klucza głównego (uwaga!):

```
Book.find(1)
```

- ▶ jw. (wersja bezpieczna):

```
Book.find_by_id(1)
```

- ▶ kilka obiektów na raz

```
Book.find(1,2,5)
```

- ▶ pierwszy obiekt

```
Book.first
```

- ▶ ostatni obiekt

```
Book.last
```

- ▶ wszystkie obiekty

```
Book.all
```

## Złożone zapytania

- ▶ określona wartość atrybutu (zwraca tablicę):  
`Book.find_all_by_title('Ruby')`
- ▶ określona wartość atrybutu (zwraca obiekt lub nil):  
`Book.find_by_title('Ruby')`
- ▶ z prostymi opcjami (zwraca tablicę):  
`Author.where('last_name=?', last_name_var).order('first_name ASC')`
- ▶ z prostymi opcjami (zwraca obiekt lub nil):  
`Author.where('last_name=?', last_name_var).order('first_name ASC').first`
- ▶ z zakresem (zwraca tablicę):  
`Author.where(:birth_date => (40.years.ago..30.years.ago)).order(:birth_date)`
- ▶ maksymalna liczba wyników (zwraca tablicę):  
`Author.limit(10).offset(20)`

# Zapytania na powiązanych obiektach

```
author = Author.first
author.books.order(:publish_date).first
author.books.where(:publish_date => (10.years.ago..Time.now)).
  limit(3)
```

```
category = Category.find_by_name("Proza")
category.books.order('rating DESC').limit(10)
```

## Opcje zapytań

- ▶ **where** – określa warunki zapytania, akceptuje tablicę zwykłą lub asocjacyjną
- ▶ **order** – określa kolejność wyników
- ▶ **first** – zwraca pierwszy wynik
- ▶ **last** – zwraca ostatni wynik
- ▶ **limit** – określa maksymalny rozmiar wyniku
- ▶ **offset** – określa przesunięcie wyniku względem początku
- ▶ **select** – zawęży listę zwróconych atrybutów
- ▶ **includes** – zachłanne ładowanie powiązanych obiektów

## scope – nazwane zapytania

Dodając do klas makro `scope`:

```
class Author < Person
  scope :living, where(:death_date => nil)
  scope :polish, where(:nationality => "polski")
end
```

możemy pisać bardziej czytelne zapytania:

```
Author.living.polish.order("last_name, first_name")
# zamiast
Author.where(:death_date => nil).
  where(:nationality => "polski").order("last_name, first_name")
```

## Model – więcej informacji

- ▶ [guides.rubyonrails.org/association\\_basics.html](https://guides.rubyonrails.org/association_basics.html)  
**asocjacje** (związki) modeli
- ▶ [guides.rubyonrails.org/migrations.html](https://guides.rubyonrails.org/migrations.html)  
**migracje** schematu bazy danych
- ▶ [guides.rubyonrails.org/active\\_record\\_validations\\_callbacks.html](https://guides.rubyonrails.org/active_record_validations_callbacks.html)  
**walidacje**
- ▶ [guides.rubyonrails.org/active\\_record\\_querying.html](https://guides.rubyonrails.org/active_record_querying.html)  
**wyszukiwanie** obiektów w bazie danych

# Pytania

PYTANIA?