Lecture 3
ActiveRecord – Rails persistence layer

Aleksander Smywiński-Pohl

Elektroniczne Przetwarzanie Informacji

9 maja 2013

Agenda

ActiveRecord basics

Query language

Associations

Validations

Callbacks

ActiveRecord problems

Alternatives

Class and migration

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

```ruby
class CreateBooks < ActiveRecord::Migration
  def change
    create_table :books do |t|
      t.string :title
      t.references :author
      # same as
      # t.integer :author_id

      t.timestamps
    end
  end
end
```

CRUD – create

```ruby
book = Book.new(:title => 'Ruby')
book.save
# or
Book.create(:title => 'Ruby')

# In the controller context
if Book.create(params[:book])
  # success
else
  # failure
end

# params[:book] looks like
{ :title => 'Ruby' }
```

CRUD – read

```ruby
# find by id
book = Book.find(1)
book.title            #=> 'Ruby'
# or
book = Book.find_by_id(1)

# find by title
book = Book.find_by_title('Ruby')
# or
book = Book.where(:title => 'Ruby')
```

CRUD – update

```ruby
book = Book.find(1)
book.title = 'Ruby for beginners'
book.save
# or
Book.update_attributes(:title => 'Ruby for beginners')

# In the controller context
if Book.update_attributes(params[:book])
  # success
else
  # failure
end

# params[:book] looks like
{ :title => 'Ruby for beginners' }
```

CRUD – destroy

```ruby
book = Book.find(1)
book.destroy

book.title = 'Ruby for beginners' #=> error
Book.find(1)                      #=> error
Book.find_by_id(1)                #=> nil
```

Agenda

ActiveRecord basics

Query language

Associations

Validations

Callbacks

ActiveRecord problems

Alternatives

Primary key (id)

```ruby
# find by primary key:
Book.find(1)
# => book1

# as above (safe version):
Book.find_by_id(1)
# => book1

# find many by primary key:
Book.find(1,3,4)
# => [ book1, book3, book4 ]

# first record
Book.first
# => book1

# last record
Book.last
# => book4
```

id	title	year
1	Ruby	2008
2	Ruby	2009
3	Python	2009
4	Ruby	2010

Find by

```ruby
# Find ALL
Book.find_all_by_title('Ruby')
# => [ book1, book2, book4 ]

Book.find_all_by_title('Perl')
# => []

# Find ONE
Book.find_by_title('Ruby')
# => book1

Book.find_by_title('Perl')
# => nil

# Find with two attributes
Book.find_by_title_and_year('Ruby',2009)
# => book2
```

id	title	year
1	Ruby	2008
2	Ruby	2009
3	Python	2009
4	Ruby	2010

Query language

- **where** – defines the query condition(s)
- **order** – defines the query order
- **select** – defines the fields that will be returned (all by default)
- **limit** – defines the maximal number of results
- **offset** – defines the results offset
- **includes** – eagerly loads associations
- **group** – groups results by given attribute
- **having** – defines the query condition(s) in case the results are grouped

where

```ruby
Book.where("title = #{params[:title]}")
# - doesn't perform a query (is lazy evaluated)
# - is a security risk - SQL-injection!

Book.where("title = ?", params[:title])
# - is automatically escaped - no SQL-injection

Book.where("title = ? AND year = ?",
           'Ruby', 2009).all
# [book2]

Book.where("title = :title",
           :title => 'Ruby').first
# book1

Book.where(:title => 'Ruby').first
# book1

Book.where(:year => (2009..2010)).all
# [book2, book4]
```

id	title	year
1	Ruby	2008
2	Ruby	2009
3	Python	2009
4	Ruby	2010

order

```ruby
Book.order("year").all
# [ book1, book2, book3, book4 ]

Book.order("id DESC").all
# [ book4, book3, book2, book1 ]

Book.order("id").reverse_order.all
# [ book4, book3, book2, book1 ]

Book.order("title DESC, year ASC").first
# book1
```

id	title	year
1	Ruby	2008
2	Ruby	2009
3	Python	2009
4	Ruby	2010

select

```ruby
book = Book.select('title').first
book.title
# => 'Ruby'
book.year
# ActiveModel::MissingAttributeError:
# missing attribute: 'year'

book.title = 'Ruby for beginners'
book.save
# ActiveRecord::ReadOnlyRecord

books = Book.select('title').uniq
books.size
# => 2
books[0].title
# 'Ruby'
books[1].title
# 'Python'
```

id	title	year
1	Ruby	2008
2	Ruby	2009
3	Python	2009
4	Ruby	2010

limit, offset

```ruby
Book.limit(2).all
# [ book1, book2 ]

Book.offset(1).all
# [ book2, book3, book4 ]

Book.offset(0).all
# [ book1, book2, book3, book4 ]

Book.limit(2).offset(1).all
# [ book2, book3 ]

Book.limit(2).offset(1).reverse_order.all
# [ book3, book2 ]
```

id	title	year
1	Ruby	2008
2	Ruby	2009
3	Python	2009
4	Ruby	2010

Pagination

- gem 'kaminari'

```
Book.page(2)
# fetch the second page

Book.page(2).per(50)
# fetch the second page of 50 items (25 by default)
```

- gem 'will_paginate'

```
Book.paginate(:page => 2)
# or
Book.page(2)
# fetch the second page

Book.paginate(:page => 2, :per_page => 50)
# fetch the second page of 50 items (30 by default)
```

includes

n+1 query problem – assuming there is an association between two classes, if we query for a collection of n objects, we will have to issue n queries for the associated objects (thus $n + 1$ queries will be issued). This is *very* slow.

We can resolve this problem, using `includes`.

includes

```ruby
Book.all.each do |book|
  book.author.name
end
# This will require 5 sql queries:
# select * from books
# select * from authors where author_id = 1
# select * from authors where author_id = 2
# ...

Book.includes(:author).all.each do |book|
  book.author.name
end
# This will require only 2 sql queries:
# select * from books
# select * from authors where id in (1, 2, 3)
```

id	title	year	author_id
1	Ruby	2008	1
2	Ruby	2009	2
3	Python	2009	1
4	Ruby	2010	3

group, having

```ruby
books = Book.select("title, count(*)").group("title").all
books[0].title
# => 'Ruby'
books[0].count
# => 3
books[1].title
# => 'Python'
books[1].count
# => 1

Book.select("title, count(*)").group("title").
  having("count(*) > 1").all
# => 'Ruby', 3

Book.select("title, count(*)").group("title").
  where("year > 2008").all
# => 'Ruby', 2
# => 'Python', 1
```

id	title	year
1	Ruby	2008
2	Ruby	2009
3	Python	2009
4	Ruby	2010

Scoping

Some finder methods are more popular than others. We can make them more readable by providing *scopes* for them:

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

# now we can write
Author.living.polish.order("last_name, first_name")

# instead of
Author.where(:death_date => nil).
  where(:nationality => "Polish").order("last_name, first_name")
```

Default scope

The same way we can define the default scope:

```
class Book
  default_scope where("year > 1980")
end

# Issuing any query will attach this condition
Book.all
# in fact Book.where("year > 1980").all

# To skip the scoping, we write
Book.unscoped.all
```

Agenda

ActiveRecord basics

Query language

Associations

Validations

Callbacks

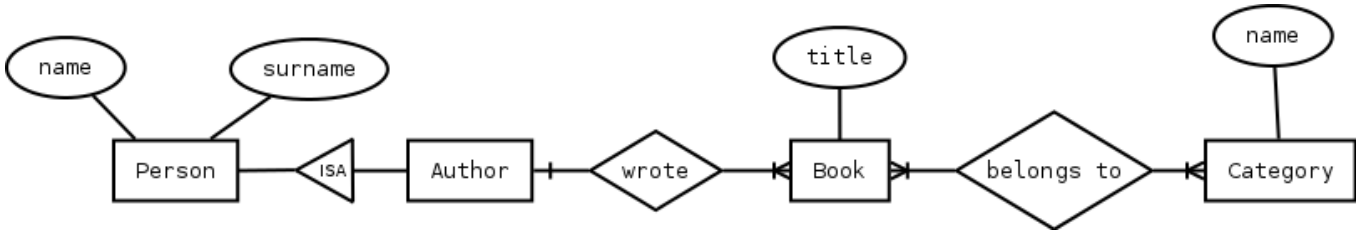
ActiveRecord problems

Alternatives

Types of associations

- `has_many`
- `belongs_to`
- `has_one`
- `has_and_belongs_to_many`
- `has_many :through`
- `belongs_to :polymorphic`

Conceptual model vs. object model




```ruby
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
```

belnogs_to

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

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

has_many

```ruby
class Author < Person
  has_many :books
end

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(...)
```

has_and_belongs_to_many

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

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(...)
```

has_many :through

```ruby
class Developer < ActiveRecord::Base
  has_many :projects
  has_many :tasks, :through => :projects
end

class Project < ActiveRecord::Base
  belongs_to :developer
  has_many :tasks
end

class Task < ActiveRecord::Base
  belongs_to :project
end
```

Note: `through` association is read-only.

has_many :polymorphic

```ruby
class Comment < ActiveRecord::Base
  belongs_to :item, :polymorphic => true
end

class Post < ActiveRecord::Base
  has_many :comments, :as => :item
end

class Image < ActiveRecord::Base
  has_many :comments, :as => :item
end

post = Post.new
post.comments << Comment.new

image = Image.new
image.comments << Comment.new
```

Agenda

ActiveRecord basics

Query language

Associations

Validations

Callbacks

ActiveRecord problems

Alternatives

Predefined validations

- **presence** – presence of an attribute
- **acceptance** – acceptance of a condition
- **confirmation** – confirmation (equality with 'confirmation' field) of an attribute
- **exclusion** – exclusion of the value of an attribute
- **inclusion** – inclusion of the value of an attribute
- **format** – match with a pattern
- **length** – length (of text)
- **numericality** – numericality (the feature of being a number) of an attribute
- **uniqueness** – uniqueness of an attribute (better served on the DB level with appropriate field option)
- **associated** – validation of an associated object

presence

```ruby
class Book < ActiveRecord::Base
  validate :title, :presence => true
end

book = Book.new
book.save
# => false
book.title = 'Ruby'
book.save
# => true
```

acceptance

```ruby
class Order < ActiveRecord::Base
  validate :terms_of_service, :acceptance => true
end

order = Order.new
order.save
# => false
order.terms_of_service = true
order.save
# => true
```

confirmation

```ruby
class Account < ActiveRecord::Base
  validate :password, :confirmation => true
end

account = Account.new
account.password = '123456'
account.save
# => false
account.password_confirmation = '123'
account.save
# => false
account.password_confirmation = '123456'
account.save
# => true
```

exclusion

```ruby
class Language < ActiveRecord::Base
  validate :name, :exclusion => { :in => ['PHP'] }
end

language = Language.new(:name => 'PHP')
language.save
# => false
language.name = 'Ruby'
language.save
# => true
```

inclusion

```ruby
class Person < ActiveRecord::Base
  validate :gender, :inclusion => { :in => %w{male female} }
end

person = Person.new(:gender => "child")
person.save
# => false
person.gender = 'female'
person.save
# => true
```

format

```ruby
class Book < ActiveRecord::Base
  validate :title, :format => { :with => /\A[A-Z](\w|\s)+\z/ }
end

book = Book.new(:title => "bad title")
book.save
# => false
book.title = 'Ruby'
book.save
# => true
```

length

```ruby
class Person < ActiveRecord::Base
  validate :name, :length => { :in => 4..10 }
end

person = Person.new(:name => "Jo")
person.save
# => false
person.name = 'Jolomolodonton'
person.save
# => false
person.name = 'Jolomolo'
person.save
# => true
```

- in
- minimum
- maximum
- is

numericality

```ruby
class Person < ActiveRecord::Base
  validate :age, :numericality => { :only_integer => true }
  validate :shoe_size, :numericality => true
end

person = Person.new(:age => 10.1, :shoe_size => "a")
person.save
# => false
person.age = 10
person.shoe_size = 5.5
person.save
# => true
```

uniqueness

```ruby
class Person < ActiveRecord::Base
  validate :email
end

person = Person.new(:email => "john@domain.com")
person.save
# => true
person = Person.new(:email => "john@domain.com")
person.save
# => false
```

- scope

associated

```
class Person < ActiveRecord::Base
  has_many :orders
  validates_associated :orders
end
```

- Calls `valid?` on each associated object.

Agenda

ActiveRecord basics

Query language

Associations

Validations

Callbacks

ActiveRecord problems

Alternatives

Creating an object

- before_validation
- after_validation
- before_save
- around_save
- before_*create*
- around_*create*
- after_*create*
- after_save

Updating an object

- before_validation
- after_validation
- before_save
- around_save
- before_*update*
- around_*update*
- after_*update*
- after_save

Destroying an object

- before_*destroy*
- around_*destroy*
- after_*destroy*

Callback example

```ruby
class Post < ActiveRecord::Base
  validate :title, :format => {:with => /\A\w+\z/}

  before_validation {|post| post.title.strip! }
  # same as
  before_validation :strip_title

  protected
  def strip_title
    self.title.strip!
  end
end
```

Do not use callbacks to implement sophisticated business logic. Use service objects instead.

Service object

```
class UserCreator
  def initialize(factory=User,social_network=SocialNetwork,
                 mailer=UserMailer)
    @factory = factory
    @mailer = mailer
    @social_network = social_network
  end

  def create(params)
    user = @factory.new(params)
    user.save
    @mailer.send_confirmation(user)
    @social_network.user_created(user)
    user
  end
end

user = UserCreator.new.create(params)
```

Agenda

ActiveRecord basics

Query language

Associations

Validations

Callbacks

ActiveRecord problems

Alternatives

User input and SQL

- SQL-injection
  passing SQL commands as SQL query fragments to perform unintended queries
- attribute mass-assignment problem
  passing attribute values that are not expected by the developer

SQL-injection

```
User.where("name = '#{params[:name]}'")

params[:name] = "some name'; DROP TABLE users; --"

"SELECT * from users where name = 'some name'; DROP TABLE users; --'"

# This is a valid SQL query (two in fact). The code after -- is omitted.

# Safe:
User.where("name = ?", params[:name])
User.where(:name => params[:name])
```

Mass assignment

```ruby
User.update_attributes(params[:user])

# Rails by default converts the passed key-value pairs into a hash.
# An attacker can easily provide the following data:
params[:user] = { :admin => true, ... }

# In older Rails we have to protect individual attributes
# (black-list approach)
class User < ActiveRecord::Base
  attr_protected :admin
end

# In current Rails we have to specify all accessible attributes
# (white-list approach):
class User < ActiveRecord::Base
  attr_accessible :name, :surname
end
```

Mass assignment

```
class AccessRulesController
  def grant_administrator_rights
    user = User.find(params[:id])
    user.admin = true
    user.save
  end
end
```

In the next version of Rails the problem will be solved on the controller level with strong parameters.

Strong parameters

```ruby
class BooksController < ActionController::Base
  def create
    @book = Book.create(params[:book]) # raises exception
  end

  def update
    @book = Book.find(params[:id])
    @book.update_attributes(book_params)
    redirect_to @book
  end

  private
  def book_params
    params.require(:book).permit(:title, :author_id)
  end
end
```

ActiveRecord interface

- `where`, `order`, `select`, ... – there are many methods that are added to the class if it inherits from AR
- it is very hard to write isolated tests by stubbing these methods
- AR defines *infinite protocol* for the classes

How to overcome these problems

- do not call AR methods in the cooperating classes
- provide domain-specific methods which hide the AR calls
- separate unit tests that don't touch the database from integration tests that need access to the persistence layer
- when running the unit tests use `NullDB`
- use `FigLeaf` to make most of the AR methods private

NullDB

```ruby
# Avdi Grimm "Objects on Rails"

module SpecHelpers
  def setup_nulldb
    schema_path = File.expand_path('../db/schema.rb',
                                   File.dirname(__FILE__))
    NullDB.nullify(schema: schema_path)
  end

  def teardown_nulldb
    NullDB.restore
  end
end
```

FigLeaf

```ruby
# Avdi Grimm "Objects on Rails"

class TodoList < ActiveRecord::Base
  include FigLeaf
  hide ActiveRecord::Base, ancestors: true,
       except: [Object, :init_with, :new_record?,
                :errors, :valid?, :save]
  hide_singletons ActiveRecord::Calculations,
                  ActiveRecord::FinderMethods,
                  ActiveRecord::Relation

  #...
  def self.with_name(name)
    where(:name => name)
  end
end

TodoList.where(:name => "Some name")
# error: call to private method 'where'

TodoList.with_name("Some name")
# OK
```

Agenda

ActiveRecord basics

Query language

Associations

Validations

Callbacks

ActiveRecord problems

Alternatives

DataMapper

- variety of data-stores
- legacy database schemas
- composite primary keys
- auto-migrations
- support for DB-level data integrity
- strategic eager loading
- querying by associations
- identity map
- lazy properties
- object-oriented comparison of fields

Legacy DB support

```
class Post
  include DataMapper::Resource

  # set the storage name for the :legacy repository
  storage_names[:legacy] = 'tblPost'

  # use the datastore's 'pid' field for the id property.
  property :id, Serial, :field => :pid

  # use a property called 'uid' as the child key (the foreign key)
  belongs_to :user, :child_key => [ :uid ]
end
```

Composite keys

```ruby
class LineItem
  include DataMapper::Resource

  property :order_id,    Integer, :key => true
  property :item_number, Integer, :key => true
end

order_id, item_number = 1, 1
LineItem.get(order_id, item_number)
# => [#<LineItem @orderid=1 @item_number=1>]
```

Auto-migrate

```
class Person
  include DataMapper::Resource
  property :id,   Serial
  property :name, String, :required => true
end

DataMapper.auto_migrate!
```

Auto-upgrade

```ruby
class Person
  include DataMapper::Resource
  property :id,   Serial
  property :name, String, :required => true

  # new property
  property :hobby, String
end

DataMapper.auto_upgrade!
```

Data integrity – DB level

```
class Person
  include DataMapper::Resource
  property :id, Serial
  has n, :tasks, :constraint => :destroy
end

class Task
  include DataMapper::Resource
  property :id, Serial
  belongs_to :person
end
```

Lazy properties

```
class Animal
  include DataMapper::Resource

  property :id,    Serial
  property :name,  String
  property :notes, Text     # lazy-loads by default
end
```

Field comparison

```ruby
Zoo.first(:name => 'Galveston')

# 'gt' means greater-than. 'lt' is less-than.
Person.all(:age.gt => 30)

# 'gte' means greather-than-or-equal-to. 'lte' is also available
Person.all(:age.gte => 30)

Person.all(:name.not => 'bob')

# If the value of a pair is an Array, we do an IN-clause for you.
Person.all(:name.like => 'S%', :id => [ 1, 2, 3, 4, 5 ])

# Does a NOT IN () clause for you.
Person.all(:name.not => [ 'bob', 'rick', 'steve' ])

# Ordering
Person.all(:order => [ :age.desc ])
# .asc is the default
```

Sequel

- concise DSL for constructing SQL queries and table schemas
- optimized for speed
- thread safety
- connection pooling
- prepared statements
- bound variables
- stored procedures
- savepoints
- two-phase commit
- transaction isolation
- master/slave configurations
- database sharding

Short example

```ruby
DB.create_table :items do
  primary_key :id
  String :name
  Float :price
end

items = DB[:items] # Create a dataset

# Populate the table
items.insert(:name => 'abc', :price => rand * 100)
items.insert(:name => 'def', :price => rand * 100)
items.insert(:name => 'ghi', :price => rand * 100)

# Print out the number of records
puts "Item count: #{items.count}"

# Print out the average price
puts "The average price is: #{items.avg(:price)}"
```