



Ruby on Rails Short Course

Part 3: Basic Rails

Armando Fox

UC Berkeley RAD Lab

Outline of the day

1. Web apps, MVC, SQL, Hello World
2. Just enough Ruby
3. **Basic Rails**

Lunch break

4. Advanced model relations
5. AJAX & intro to testing
6. Configure & deploy

Informal discussion: RoR and pedagogy

Outline of Session 3

- Overview of ActiveRecord
 - accessors and attributes, constructors, finders
 - validations, model lifecycle & callbacks
 - after lunch: ActiveRecord *associations*—coolness
- Overview of ActionView
 - RHTML, RXML, RJS, HAML
 - Forms and model objects, tag helpers
 - Preview: AJAX
- Overview of ActionController
 - connections between controller & view
 - sessions: the hash & the flash
 - stupid filter tricks

Outline of Session 3

- Overview of ActiveRecord
 - accessors and attributes, constructors, finders
 - validations, model lifecycle & callbacks
 - after lunch: ActiveRecord *associations*—coolness
- Overview of ActionView
 - RHTML, RXML, RJS, HAML
 - Forms and model objects, tag helpers
 - Preview: AJAX
- Overview of ActionController
 - connections between controller & view
 - sessions: the hash & the flash
 - stupid filter tricks

Quick review: hashes and function call notation

- Immediate hash (any object can be a key, any object can be an attribute)

```
my_hsh = { :foo => 1, "x" => nil, 3 => ['a', 4] }
```

```
my_hsh[:nonexistent_key] returns nil
```

- Parens can be omitted from function calls if parsing is unambiguous



```
x = foo(3, "no")  x = foo 3, "no"
```

- Braces can be omitted from hash if parsing is unambiguous

```
x = foo ( { :a=>1, :b=>2 } )  x = foo ( :a=>1, :b=>2 )
```

- easy way to do keyword arguments
- Caveat: passing immediates to a function that accepts multiple hashes as its arguments

Active Record: what is it?

- A class library that provides an object-relational model over a plain old RDBMS
- Deal with objects & attributes rather than rows & columns
 - query result rows  enumerable collection
 - object hierarchy  join query

Review: the Student Example

- **object attributes** are “just” instance methods
- ActiveRecord
 - default `attr_accessor` for each table column
 - perform type-casting as needed
 - can be overridden, virtualized, etc.

```
class Foo
  # constructor
  def initialize(args={})
    @bar = args[:bar]
  end
  # getter
  def bar
    @bar
  end
```

```
  end
  # setter
  def bar=(newval)
    @bar = newval
  end
```

```
end

class Autofoo
  attr_accessor :bar
end
```



Example: open up Student class...

```
class Student

  def youngster?
    self.degree_expected > Date.parse("June 15, 2008")
  end

  def days_till_graduation_as_string
    graduation = self.degree_expected
    now = Date.today
    if graduation.nil?
      "This person will never graduate."
    elsif graduation < now
      "Graduated #{now-graduation} days ago"
    else
      "Will graduate in #{graduation-now} days"
    end
  end
end
```


Virtual attributes example: simple authentication

- Only salt & hashed password are stored

```
class Customer

  def password=(pass)
    pw=pass.to_s.strip
    self.salt = String.random_string(10)
    self.hashed_password = Digest::SHA1.hexdigest(pw + self.salt)
  end

  def self.authenticate(username, pass)
    (u=find(:first, :conditions=>["username LIKE ?", username]) &&
     Customer.encrypt(pass,u.salt) == u.hashed_password)
  end
end
```

Constructors

- Initializer knows if it's been handed a block
(predicate method Kernel#block_given?)

```
s = Student.new(:last_name => "Fox",
                # unspecified attributes get
                # table column's DEFAULT values
                :ucb_id => 99988)

s = Student.new do |stu|
  stu.last_name = "Fox"
  stu.ucb_id = 99988
end

s = Student.new
s.last_name = "Fox"
s.ucb_id = 99988
```

New != Create

- Call `s.save` to write the object to the database
 - `s.create(args)` (`s.new(args)` ; `s.save`
 - `s.update_attributes(hash)` can be used to update attributes in place
 - `s.new_record?` is true iff no underlying database row corresponds to `s`
- `save` does right thing (INSERT or UPDATE)
- Convention over configuration:
 - if `id` column present, assumes primary key
 - `updated_at/created_at` (resp. `*_on`) automatically set if present to update/creation date (resp. time)

But!... validations

```
class Student < ActiveRecord::Base
  validates_presence_of :degree_expected, :last_name, :ucb_id
  validates_numericality_of :ucb_id
  validates_length_of :ucb_id, :within => 7..10,
    :message => "ID number must consist of 7 to 10 digits"
  # an alternative:
  # validates_format_of :ucb_id, :with => /[0-9]{7,10}/,
  # :message => "ID number must consist of 7 to 10 digits"
  validates_uniqueness_of :ucb_id
  # only one person with a given last name can graduate on any given day
  validates_uniqueness_of :last_name, :scope => :degree_expected
end
```

- *model lifecycle* specifies well-defined callbacks for ActiveRecord manipulation
 - allows keeping validation semantics with the model
 - allows keeping validation code separate from mainline
- are those macros, language keywords, or what?

How would you use these?

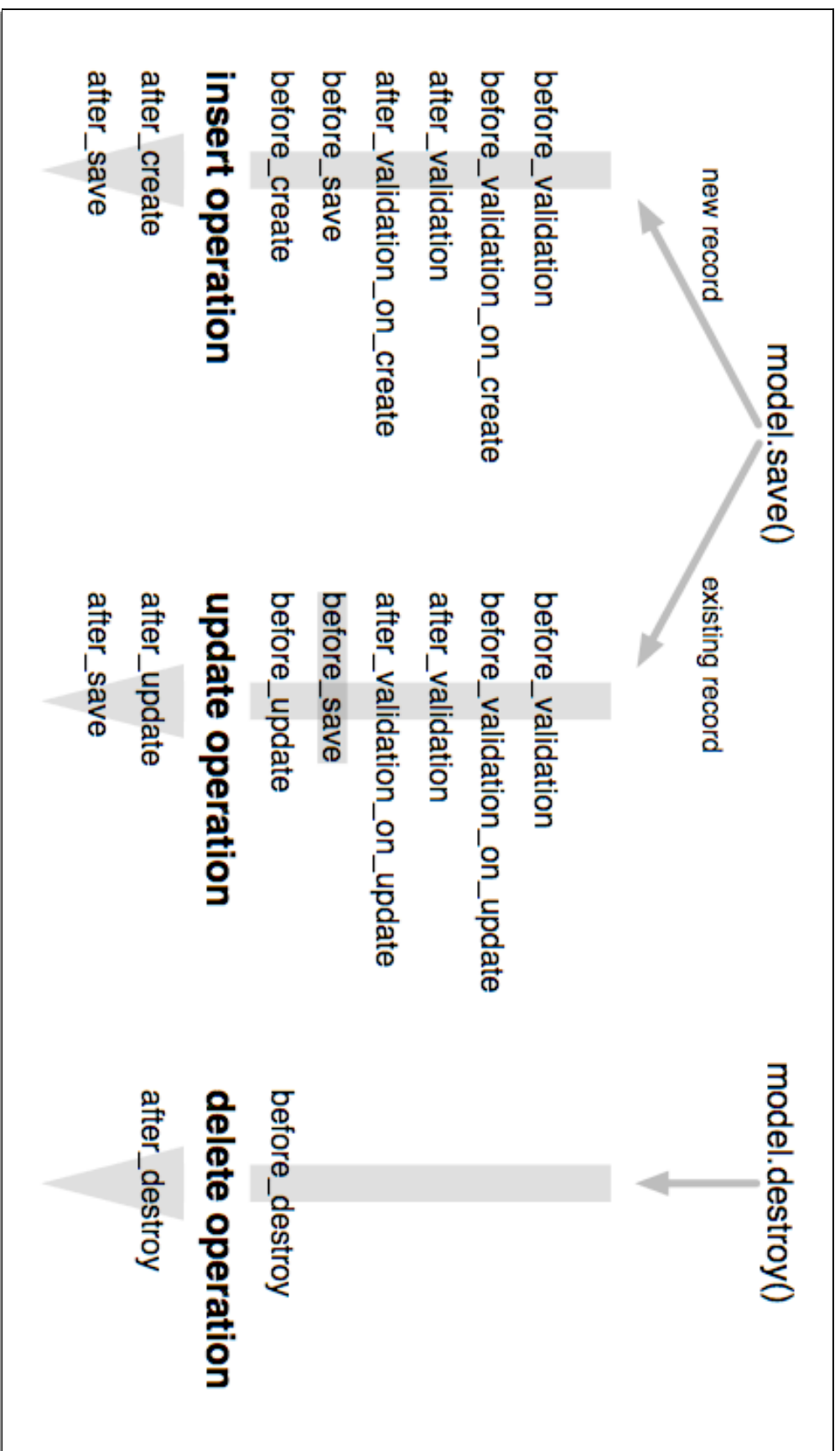
```
# Using validations in controllers
begin
  # ...do complex things with the object...
  object.save!
rescue ActiveRecord::RecordInvalid => invalid_object
  puts invalid_object.record.errors
end

# Another way...do complex things with the object...
unless object.save
  puts object.errors
  return
end
#...continue
```

- **Note convention:** `save!` **vs.** `save` (also `create`, `update`, ...)
- Scaffolding provides a default use via a view helper method `errors_for`

Callbacks: the Return (get it?) of Aspect-Oriented Programming

Allows Pre and Post Operations



Another way to do passwords

Encrypt a password before saving the record

```
# Encrypts some data with the salt.
def self.encrypt(password, salt)
  Digest::SHA1.hexdigest("--#{salt}--#{password}--")
end

def before_save
  return if password.blank?
  self.salt = Digest::SHA1.hexdigest("--#{Time.now.to_s}--#{login}--") if new_record?
  self.encrypted_password = encrypt(password)
end
```

find() — SQL SELECT

```
# To find an arbitrary single record:
s = Student.find(:first)
# To find all records:
students = Student.find(:all)
```

```
# find by 'id' primary key (Note! throws RecordNotFound)
book = Book.find(1235)
# Find a whole bunch of things
ids_array = get_list_of_ids_from_somewhere()
students = Student.find(ids_array)
```

```
# To find by column values:
armando = Student.find_by_last_name('Fox')
a_local_grad =
  _Student.find_by_city and degree_expected('Berkeley',
    Date.parse('June 15, 2007'))
```

```
# To find only a few, and sort by an attribute
many_localgrads =
  Student.find all by city and degree_expected('Berkeley',
    Date.parse('June 15, 2007'), :limit=>30, :order=>:last_name)
```


Find by conditions

Use ? for values from parameters. Rails will sanitize the SQL and prevent any SQL injection

```
Student.find(:all, :conditions => "last_name LIKE 'fox' AND
degree_expected > #{Date.parse('June 15,2007').to_formatted_s}")
# better - sanitizes SQL to avoid injection attacks, and does type casting:
Student.find(:all, :conditions => ["last_name LIKE ? AND degree_expected > ?",
tainted_lastname, Date.parse('Jun 15,07')])
```

You can also specify ordering and use arbitrary SQL operators
(caveat emptor: database portability may be jeopardized)

```
# Using SQL conditions
books = Book.find(:all,
:conditions => ['pub_date between ? and ?',
params[:start_date], params[:end_date]],
:order => 'pub_date DESC')
```

Advanced Find

You can also specify limits and offsets, and oh so much more

```
books = Book.find(:all,  
  :conditions => ['pub_date between ? and ?',  
    params[:start_date], params[:end_date]],  
  :limit => 10, :offset => params[:page].to_i * 10)
```

- :lock - Holds lock on the records (default: share lock)
- :select - Specifies columns for SELECT (default *)
- :group - (used with select) to group
- :readonly - load as read-only (object can't be saved)
- :include - Prefetches joined tables (try :include first; more about this in Section 4)
- Note: use SQL-specific features at your own risk....

Caveat!

- The result of a find-all operation *mixes in* `Enumerable`
- `Enumerable` defines methods `find` and `find_all`
- **Not to be confused with**

`ActiveRecord::Base#find!`

```
students = Student.find(:all, :conditions => ["degree_expected > ?", Time.now])
palindromic = students.find_all { |s| s.last_name.reverse == s.last_name }
lucky = palindromic.find { |s| s.uch_id.odd? }
```

Outline of Session 3

- Overview of ActiveRecord
 - accessors and attributes, constructors, finders
 - validations, model lifecycle & callbacks
 - after lunch: *ActiveRecord associations*—coolness
- Overview of ActionView
 - RHTML, RXML, RJS, HAML
 - Forms and model objects, tag helpers
 - Preview: AJAX
- Overview of ActionController
 - connections between controller & view
 - sessions: the hash & the flash
 - stupid filter tricks

Action View

- A template for rendering views of the model that allows some code embedding
 - commonly RHTML; also RXML, HAML, RJS
 - note...too much code breaks MVC separation
 - convention: views for model *foo* are in `app/views/foo/`
- “Helper methods” for interacting with models
 - model values `HTML` elements (e.g. menus)
 - HTML form input `assignment` to model objects
- DRY (Don't Repeat Yourself) support
 - *Layouts* capture common page content at application level, model level, etc. (`app/views/layouts/`)
 - *Partials* capture reusable/parameterizable view patterns

Helper Methods for Input & Output

- Review: we saw a **simple view** already...
 - Anatomy: `<% code %>` `<%= output %>`
- But these form tags are generic...what about **model-specific form tags**?
- In the RHTML template:

```
<%= text_field 'student', 'last_name' %>
```
- In HTML delivered to browser:

```
<input id="student_last_name"
name="student[last_name]" size="30"
type="text" value="Fox" />
```
- What happened? For that we have to look at *partial*.

- Reusable chunk of a view
 - e.g., one line of a Student table
 - e.g., form to display/capture Student info that can be used as part of Edit, Show, Create,...
 - file naming convention: the partial *foo* for model *bar* is in `app/views/bar/_foo.rhtml`
- default partial *form* generated by scaffolding
 - so *edit.rhtml* (the *edit view*) is really trivial, and differs minimally from *new.rhtml*
 - but both of them set the instance variable *student*
- So what's the point of model-specific form fields? We'll revisit shortly when we discuss controllers.

What about a collection?

- **Common idiom:**

```
@students.each do |student|  
  render :partial => 'student'  
end
```

- **Captured by:**

```
render :partial => :student, :collection =>  
  @students
```

- other options allow passing local variables to
partial & specifying “divider” template

Validation error reporting in views: CSS+HTML+Rails

- form **partial** sets ID, class of specific elements
 - text_field helper conditionally wraps HTML element in `<div class="fieldWithErrors">`
 - error_messages_for (in 'form' partial) wraps `@student.errors` (set by ActiveRecord validation callbacks) with `<div id="errorExplanation">`

- Default **layout** for class

(`app/views/layouts/students.rhtml`)

- generated by script/generate scaffold student
- pulls in **stylesheet** `scaffold.css` (generic scaffolding styles) that define visual appearance for element ID `errorExplanation` and class `fieldWithErrors`

Yow!

Note what does *not* happen

- No explicit conditional code in views
- No conflation of logical structure with visual appearance (CSS used wisely)
 - `error_messages_for` returns generic HTML tagged with (user-specified) id's and classes
- No needless repetition: use templates to DRY out code
 - 'form' partial
 - 'student' layout: elements common to all Student-related views, e.g. page title
 - (not in this example) reuse of top-level formatting via `application.rhtml` template
- Another way of looking at it: the world's going declarative

Outline of Session 3

- Overview of ActiveRecord
 - accessors and attributes, constructors, finders
 - validations, model lifecycle & callbacks
 - after lunch: ActiveRecord *associations*—coolness
- Overview of ActionView
 - RHTML, RXML, RJS, HAML
 - Forms and model objects, tag helpers
 - Preview: AJAX
- Overview of ActionController
 - connections between controller & view
 - sessions: the hash & the flash
 - stupid filter tricks


Action Controller

- Each incoming request instantiates a new Controller object with its own instance variables
 - Routing (Sec. 4) determines which method to call
 - Parameter unmarshaling (from URL or form sub.) into `params [] hash`
- 🐥...well, not really a hash...but responds to `[]`, `[] =`
- Controller methods set up instance variables
 - these will be visible to the view
 - controller has access to model's class methods; idiomatically, often begins with `Model.find(...)`
- Let's see some **examples**...

Then we render...

- Once logic is done, render the view

```
render :action => 'edit'  
render :action => 'edit', :layout => 'false'  
render :text => "a bare string"  
# many other options as well...
```

- exactly one *render* permitted from controller method (1 HTTP request  1 response)
- Convention over configuration: implicit *render* looks for template matching controller method name and renders with default layouts (model, app)
- language geek side note: use of CLU-inspired `yield` in content rendering

What about those model-specific form elements?

- Recall:

```
<input type="text" id="student_last_name"
name="student[last_name]" />
```
- Related form elements for student attributes will be named `student[attr]`
 - marshalled into params as `params[:student][last_name]`, `params[:student][:degree_expected]`, etc.
 - i.e, `params[:student]` is a hash
 - `:last_name=>string`, `:degree_expected=>date`, etc.
 - and can be **assigned directly** to model object instance
 - helpers for dates and other “complex” types...magic

What else can happen?

- `redirect_to` allows falling through to different action *without* first rendering
 - fallback action will call `render` instead
 - works using HTTP 302 Found mechanism, i.e. separate browser roundtrip
- **example**: create method
 - success: redirect to `list` action
 - fail: *render* the `new` action (without `redirect`)...why?

The Session Hash

- Problem: HTTP is stateless (every request totally independent). How to synthesize a *session* (sequence of related actions) by one user?
- Rails answer: `session[]` is a magic persistent hash available to controller
 - 🦆 Actually, it's not really a hash, but it quacks like one
 - Managed at dispatch level using cookies
 - You can keep full-blown objects there, or just id's (primary keys) of database records
 - Deploy-time flag lets sessions be stored in filesystem, DB table, or distributed in-memory hash table

The Flash

- Problem: I'm about to `redirect_to` somewhere, but want to display a notice to the user
- yet that will be a different controller instance with all new instance variables



Rails answer: `flash[]`

- contents are passed to the *next* action, then cleared
- to this action: `flash.now[:notice]`
- visible to views as well as controller
- Strictly speaking, could use session & clear it out yourself

```
def controller_method_1
  if (badness)
    flash[:notice] = "You lose!"
    redirect_to :action => 'try_it'
  end
end

def try_it
  #...some stuff...
end

# in try_it.rhtml:
<% if flash[:notice] %>
  <p class="errorMsg">
    <%= flash[:notice] %>
  </p>
<% end %>
```

- A declarative way to assert various preconditions on calling controller methods
- You can check selectively (`:only`, `:except`) for...
 - HTTP request type (GET, POST, Ajax XHR)
 - Presence of a key in the flash or the session
 - Presence of a key in `params []`
- And if the check fails, you can...
 - `redirect_to` somewhere else
 - `add_to_flash` a helpful message
- A simple **example** in our simple controller

More General Filters

- Code blocks that can go before, after or around controller actions; return Boolean

```
before_filter :filter_method_name  
before_filter { |controller| ... }  
before_filter className
```

- options include :only, :except, etc.
- multiple filters allowed; calls provided to prepend or append to filter chain
- subclasses inherit filters but can use `skip_filter` methods to selectively disable them
- If any before-filter returns false, chain halted & controller action method won't be invoked
 - so filter should redirect_to, render, or otherwise deal with the request
- Simple example: authentication

Summary

- ActiveRecord provides (somewhat-)database-independent object model over RDBMS
 - made *much* more powerful through use of associations
- ActionView supports display & input of model objects
 - facilitates reuse of templates via layouts & partials
- ActionController dispatches user actions, manipulates models, sets up variables for views
 - declarative specifications capture common patterns for checking predicates before executing handlers
- Pervasive use of CSS and HTML class/ID attributes separates appearance from structure, avoids need for explicit conditional code in views



Questions