

Rails² 101

March 2008 @ Philly Emerging Tech

Aaron Mulder

CTO, Chariot Solutions

Introduction...

- This is not an “evangelism” talk
 - I assume you're already on board
- This is not a “develop ApplicationX in 5 minutes” talk
 - Those tend to gloss over too much for my taste
 - But see point 1 if you don't believe it can be done
- This is not a comprehensive talk
 - But I'll try to hit the highlights so you have some context when you start playing around
 - Do get 2 good books! (1 Ruby and 1 Rails)

Why Rails?

- Compared to PHP
 - More structure / separation
 - More DB automation
 - More libraries / plugins
 - More powerful
- Compared to Java/J2EE
 - One framework, more targeted
 - Less code
 - Less configuration
 - Less to learn
- For more detail, check out one of the evangelism talks

Some Things to Know

- Rails favors code generation
 - Often saves you time
 - Occasionally code you generated before would look different if you did it again today
- Convention over Configuration
 - You can always be explicit
 - But you don't have to, and in fact, it's normally discouraged
- Rubyisms
 - You will (and Rails does) use blocks and mixins and `@variables` and methods with hash arguments and etc.
- Don't Repeat Yourself (“DRY”)

Administrative Details

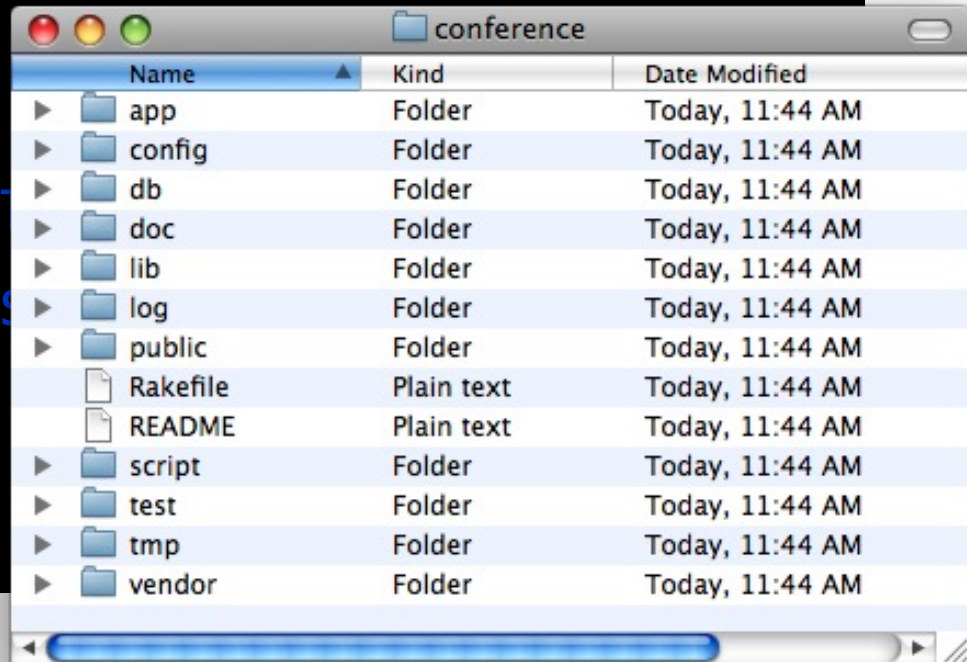
Rails Installation

- Few OSs includes a working Rails environment (OS X 10.5 is close...)
- There are “instant Rails” kits, but I have more luck separately installing these (find a blog for OS-specific procedure):
 - A current Ruby (1.8.6), if needed
 - Ruby Gems, if needed
 - MySQL & driver (if not just SQLite)
 - The “rails” Gem (and prerequisites)
 - The “mongrel” Gem
- Just make sure your path is set...

Create a Project

- Code generation in action

```
> rails conference  
create  
create app/controllers  
create app/helpers  
create app/models  
create ...
```



Project Structure

- **app** Application-specific code & views
- **config** DB, env, URL-mapping config files
- **db** DB scripts
- **lib** Your non-app-specific code
- **public** Static pages, images, etc.
- **script** Web server, IRB, install plugin, etc.
- **test** Unit & integration tests
- **vendor** Installed plugins, etc.

IDEs / Rails Editors

- Everyone (else) loves TextMate on the Mac – or even Xcode
- Some Ruby/Rails specific IDEs
 - Komodo
 - 3rdRail (from the JBuilder folks)
 - Mondrian, ArachnoRuby, others
- Some Java IDEs with Rails support
 - Aptana RadRails (Eclipse)
 - NetBeans 6 with Ruby module
 - jEdit with Ruby plugin & project view plugin
 - IDEA 7 with Ruby plugin

Why an IDE?

- Syntax highlighting
- Auto-indentation
- Some code completion / navigation?
- Debugging
- A tree view of the project, or a most helpful component-type view where you don't need to fool with directories
- Keyboard shortcuts or other quick links to generate commands, etc.

• ...

Dive into Rails: “Back End”

Database Setup

- Rails expects you to have 3 databases:
 - **Development** : used during the development process; normally overwritten frequently
 - **Test**: automatically recreated from the structure of the development DB (and populated with test data) every time you run tests
 - **Production** : for that magic moment
- MySQL was the stereotypical Rails DB, but SQLite3 is now the default and many others are supported too
- Once you've got drivers, edit `config/database.yml`

config/database.yml

development:

```
adapter: sqlite3
database: db/development.sqlite3
timeout: 5000
```

test:

```
adapter: sqlite3
database: db/test.sqlite3
timeout: 5000
```

production:

```
adapter: mysql
database: conference
username: conf
password: ete2008
socket: /tmp/mysql.sock
```

Digression: YaML

- A file format; you just saw it
- Used for DB config, test data, etc.
- Can embed dynamic values too
- Later can retrieve values by name

```
entry_name:
```

```
  key1: 17
```

```
  key2: string value
```

```
  key3: <%= Time.now %>
```

```
another_entry:
```

```
  key1: value
```

```
  . . .
```

Existing Database?

- Rails works best for databases structured “the Rails way”
 - Every table has an “id” column as PK
 - Table names like “sessions” and “people”
 - Foreign keys like “session_id”
- If not, you can normally work around this
 - Will just need some more configuration since you can't rely as much on convention
 - Disable pluralizing table names
 - Manually set table name and/or primary key on every model object
 - Extra properties for relationships
 - Or, just create views for every table

DB not the Rails way?

- **Green bits** are always necessary
- **Red bits** are used to override defaults if you need to
- Notice a “bad” DB doubles the code!

```
class Person < ActiveRecord::Base
  set_table_name "person"
  set_primary_key "person_id"
  has_many :addresses, :class_name => "location",
                  :foreign_key => "add_id"
end
```


New Database?

- Lucky You!
- Migrations
 - Write your DB definition in simple Ruby syntax, and Rails can apply it to any supported DB product
 - Some elements, like foreign keys, will still require SQL, so make it portable!
 - DB changes are versioned, and you can instantly upgrade or downgrade to any version
 - Especially useful for upgrading an app in production

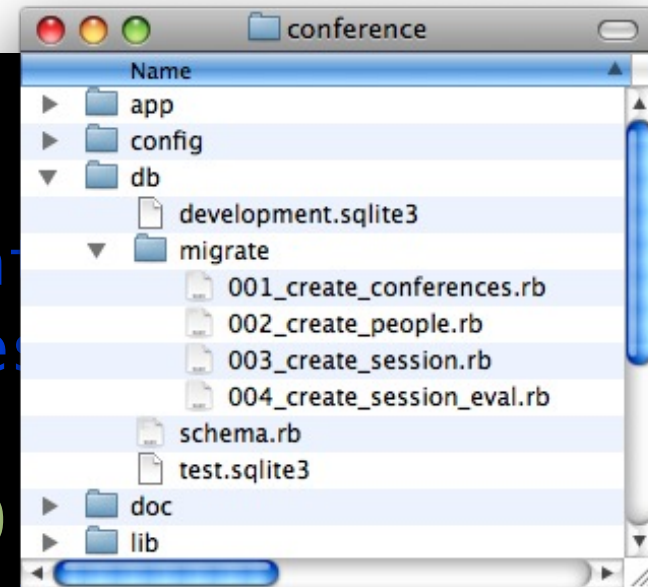
Migrations Code

```
class CreatePeople < ActiveRecord::Migration
  def self.up
    create_table :people do |table|
      table.column :reg_date, :date, :null => false
      table.column :username, :string, :limit => 60,
      :null => false
      table.column :email, :string, :limit => 100,
      :null => false
      table.column :first_name, :string, :limit => 30
      table.column :last_name, :string, :limit => 30
    end
  end

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

Migrations Usage

```
> rake db:migrate
== CreateConferences: migrating
-- create_table(:conferences)
...
> rake db:migrate VERSION=0
== CreateConferences: reverting
-- drop_table(:conferences)
...
```



Digression: Rake

- “Make” for Rails?
 - Just a handy tool with some useful “tasks”
- We just saw the “db:migrate” task
- Try `rake --tasks` to see what's available
- Normal syntax is “rake task arguments”
 - As in `rake db:migrate VERSION=3`
- Can write your own rake tasks and put them in `lib/tasks`
- Will use rake to manipulate the DB, run tests, clear log files, etc.

DB Models

- Based on the ActiveRecord library
- A lot is done automagically
 - properties for each column
 - finders for arbitrary columns or combinations
 - relationships based on simple keywords
- You won't get code completion, but a huge number of things “just work”
- We'll start by generating a model
 - could just create it by hand in `app/models` but for now this will demonstrate some other features like tests

Generate a Model

```
> script/generate model conference
exists app/models/
exists test/unit/
exists test/fixtures/
create app/models/conference.rb
create test/unit/conference_test.rb
create test/fixtures/conferences.yml
exists db/migrate
create db/migrate/
      001_create_conferences.rb
```

The Model

Test Stuff

The Migration

Digression: script/generate

- Used for generating basic code
 - We just saw it for a DB model (plus migration, unit test, test data)
 - Will later see it for all of that plus front end stuff too (controller, views, functional tests)
 - Some plugins add generation options
- Common syntax is (or try --help):

```
script/generate [generator] [class name]  
[other args]
```

```
> script/generate model conference  
[name:string starts_on:date ...]
```

script/generate Output

```
> script/generate model conference
exists app/models/
exists test/unit/
exists test/fixtures/
create app/models/conference.rb
create test/unit/conference_test.rb
create test/fixtures/conferences.yml
exists db/migrate
create db/migrate/
      001_create_conferences.rb
```

The Model

Test Stuff

The Migration

A Model

- Automatically hooks up columns as properties, finders, create with named properties, etc.
- The actual class is trivially simple if the DB table follows naming conventions:

```
class Conference < ActiveRecord::Base  
end
```

Creating...

- Can use *new* to instantiate a new User (plus *save* or *save!* to save it later)
 - or *create/create!* to instantiate and save all at once
- The *create/save* with ! throw exceptions on errors (validation failures, etc.), while the ones without just set error fields, return false, and don't perform the insert
 - But either way the models have been changed
- Can pass *new/create* a hash of property values to set on the new object

Typical “Create” Code

```
def new
  @person = Person.new # Empty person for the form
end
def create
  @person = Person.new(params[:person]) # With data!
  respond_to do |format|
    if @person.save
      flash[:notice] = "#{@person.name} created!"
      format.html { redirect_to(@person) }
      format.xml { render :xml => @person, ... }
    else
      format.html { render :action => "new" }
      format.xml { render :xml=>@person.errors ... }
    end
  end
end
```

Validation...

- Many helper methods to validate at save time
 - `validate_presence_of` (not nil)
 - `validate_length_of` (between # and #)
 - `validate_confirmation_of` (passwords)
 - `validate_uniqueness_of` (e-mail)
 - `validate_format_of` (e-mail)
 - <http://api.rubyonrails.org/classes/ActiveRecord/Validations/ClassMethods.html> for big list
- All these go on the model object
- This is normally what causes the create or save to fail

Model with Validation

```
class Person < ActiveRecord::Base
  validates_presence_of :username, :email
  validates_presence_of :password_confirmation
  validates_length_of :password, :within => 4..40
  validates_confirmation_of :password
  validates_length_of :username, :within => 3..60
  validates_length_of :first_name,
    :within => 0..30, :allow_nil => true
  validates_format_of :email,
    :with => /^([\^@\s]+)@((?:[-a-z0-9]+\.\.)+[a-z]{2,})$/i,
    :message => "Invalid email address"
  validates_uniqueness_of :username, :email,
    :case_sensitive => false
end
```

Finding...

- can use `find(id)` to find by primary key
- Or `find_by_first_name(value)` Or `find_all_by_last_name(value)` to find by any property
- Or `find(:first, :conditions => ["username=?", name])`
- Or `find(:all, :conditions => ["active=?", active], :order=>"last_name, first_name")`
- Or ...

Relationships...

- Declared on the models
 - has_one + belongs_to
 - has_many + belongs_to
 - has_and_belongs_to_many and has_and_belongs_to_many
 - through (indirect relationships)
- Can override default object type & foreign key name, plus ordering, conditions, etc.
- Creates array-type properties by the same name as the relationship
 - Can use the same find methods on those arrays as on the objects themselves (also methods like sum(:property)...))

Models with Relations

```
class Person < ActiveRecord::Base
  has_many :addresses
end
```

```
class Address < ActiveRecord::Base
  belongs_to :person
end
```

```
@me = Person.create!(...)
@me.addresses.find(:first, :conditions =>
  ["type=?", "work"])
@address = Address.new(...)
@me.addresses << @address
puts "I am #{@address.person}"
```


Playing with Models

```
> script/console
Loading development environment.
>> Person.create(:username=>'ammulder',
: first_name=>'Aaron', :last_name=>'Mulder')
=> #<Person:0x3565894 ...
>> me = Person.find(1)
=> #<Person:0x3560ab0 ...
>> me.first_name
=> "Aaron"
>>
```

script/console

- Just like “irb” except it includes the whole Rails environment and your application code
- Often used to try out models or lib/*.rb logic at the command line
 - Not a replacement for unit tests, but a quick sanity check or an easy way to get some data into the dev DB
- Not as useful for controllers because it's harder to simulate an HTTP request
 - But you can see how the functional tests do it

Unit Tests

- Located in `test/unit/*.rb`
- Uses test data in `test/fixtures/*.yaml`
 - In YaML format
- Run with `rake test` Or `rake test:units`
- Runs against the test DB, which will be empty except for the fixtures you list in the unit test class
- Pretty handy if you're customizing `script/generate` code and just want to make sure you didn't bust anything

Fixtures

people.yml

aaron:

```
username: ammulder
first_name: Aaron
last_name: Mulder
reg_date: <%= Time.now %>
```

bob:

```
username: bob
first_name: Bob
last_name: Smith
reg_date: <%= 2.days.ago %>
```

addresses.yml

aarons_home:

```
person: aaron    # Note: no IDs, use entry names
state: PA
```

Sample Unit Test

```
require File.dirname(__FILE__) + '/../test_helper'

class PersonTest < Test::Unit::TestCase
  fixtures :people, :addresses # or better, :all

  def test_find
    someone = Person.find_by_username('ammulder')
    assert_equal someone, people(:aaron)
  end

  def test_create
    Person.create(:username=>'foo', ...)
    someone = Person.find_by_username('foo')
    assert_not_nil someone
  end
end
```

Dive into Rails: “Front End”

Views and Controllers

- URLs from incoming requests are mapped to controller methods and key parameters
- REST: Normally `/controller` or `/controller/id` plus the HTTP method is important
- so `/person/2` and HTTP `GET/PUT/DELETE`
 - `PersonController`
 - `GET=show` method of `PersonController` with `params[:id] = 2`, `PUT=update` method...
- A view is typically called by each controller method (`show.html.erb` for the `show` method, etc.)

Generating a Scaffold

- Can `script/generate scaffold model [controller] [viewname ...]`
 - Emits the same stuff as `generate model`, but also with a controller, views, a layout, a stylesheet, functional tests, etc.
- Often more than you really want, but great to do once to see how everything can fit together “the Rails way”

```
> script/generate scaffold person
```


View the Generated Site

- Generating a scaffold makes a working site!
- Start the web server with `script/server`
 - Uses the development DB by default
 - Listens on port 3000

```
> script/generate scaffold person
> rake db:migrate
> script/server
```

- Visit <http://localhost:3000/people/>
- Try making a quick change to a view and refresh the page!

Controllers

- The controller has the logic that gets executed when the request is handled
 - preparing data for a form
 - processing a form submission
 - redirecting or rendering a different view in case of error
 - etc.
- Then a view is rendered, unless the controller redirected, etc.
- Controller instance variables (@person) are visible to the view

“Create” Code Again

```
class PeopleController < ApplicationController
  def new      # handles GET /people/new
    @person = Person.new # Empty person for the form
  end
  def create   # handles POST /people or /people.xml
    @person = Person.new(params[:person]) # With data!
    respond_to do |format|
      if @person.save
        flash[:notice] = "#{@person.name} created!"
        format.html { redirect_to(@person) }
        format.xml { render :xml => @person, ... }
      else
        format.html { render :action => "new" }
        format.xml { render :xml=>@person.errors ... }
      end
    end
  end
end
```

Viewing People

```
class PeopleController < ApplicationController
  def index # handles GET /people /people.xml
    @people = Person.find(:all)

    respond_to do |format|
      format.html # index.html.erb
      format.xml { render :xml => @people }
    end
  end

  def show # handles GET /people/14 /people/14.xml
    @person = Person.find(params[:id])
    respond_to do |format|
      format.html # show.html.erb
      format.xml { render :xml => @person }
    end
  end
end
```

Views

- Controllers in the `app/controllers` directory
- Views in the `app/views` directory
 - `app/views/controller-name/action.html.erb`
 - `new` => `new.html.erb`
 - `create` – no actual view (redirects)
 - `index` => `index.html.erb`
 - `show` => `show.html.erb`
- An `.html.erb` view is plain HTML with embedded ERb Ruby statements
 - And lots of helper methods for links, forms, escaping HTML, etc.

Viewing a Person

```
<table>
  <tr>
    <th>Username</th>
    <td><%=h @person.username %></td>
  </tr>
  <tr>
    <th>Full Name</th>
    <td><%=h (@person.first_name + " " +
              @person.last_name) %></td>
  </tr>
</table>
```

```
<%= link_to 'Edit', edit_person_path(@person) %> |
<%= link_to 'Back', people_path %>
```

Showing Nested Data

```
<tr>
  <th>Sessions Attended</th>
  <td>
    <%= @person.sessions.empty? ? "None" : ""%>
    <% @person.sessions.each do |session| %>
      Attended <%= session.speaker + ": " +
        session.name %><br />
    <% end %>
  </td>
</tr>
```

Editing a Person

```
<h1>Editing user</h1>
<%= error_messages_for :person %>
<% form_for(@person) do |f| %>
  <p>
    <b>First name</b><br />
    <%= f.text_field :first_name %>
  </p>

  <p>
    <b>Last name</b><br />
    <%= f.text_field :last_name %>
  </p>
  <%= f.submit 'Update' %>
<% end %>

<%= link_to 'Show', @person %>
```


Partials

- Reusable chunks of view (remember DRY)
- named starting with `_` but referred to without it
- e.g. `render :partial => 'foo'` looks for `_foo.erb` in the views directory
- Typically several pages include the same partial
- Can pass variables into a partial

_session.erb Partial

```
# _session.erb
Attended <%= session.name %>
by <%= session.speaker %>
in room <%= session.location %>
and rated it <%= session.rating %> stars

# views/people/show.html.erb
<tr>
  <th>Sessions Attended</th>
  <td>
    <%= @person.sessions.empty? ? "None" : "" %>
    <% @person.sessions.each do |attended| %>
      <%= render :partial => 'session'
        :locals => { :session => attended } %><br />
    <% end %>
  </td>
</tr>
```

Other Form Options

- Can use `form_for` to construct forms for nested objects (to create an attendance record within the context of a particular user, for example)
- Can use `form_tag` instead of `form_for` to create forms that don't automatically bind to a given `@variable` (but still behave nicely)
 - May be useful if e.g. you want one form to submit data that should go into several different objects
- Can use AJAX helpers for autocomplete fields, etc. (if appropriate plugins installed)

Layouts

- Notice that the views don't have the `<html>` and `<head>` and so on...
- Each controller has an associated layout in `app/views/layouts`
 - Or, if no match is there, defaults to `application.html.erb` in the same directory
- The template applies the header and footer, common stylesheet and JavaScript, etc.
- Look at the one scaffold generates

Dive into Rails: Advanced

Routing

- `config/routes.rb` defines the mapping of URLs to controllers/actions/etc.
 - Can add custom entries to create prettier URL formats for certain actions
 - Can add “named routes” which you can refer to by name elsewhere when you want to emit a particular URL (we've seen these used)
 - Can specify a controller to handle requests to the root (that is, /)
- Try `rake routes` to see what the scaffold set up

Transactions

- Normally each ActiveRecord call is executed in its own transaction
- Can wrap several in a block to include them all in a single transaction
 - e.g. create and validate parent and child records all at once

```
Person.transaction do
```

```
  @person = Person.create(...)
```

```
  @address = Address.create(...)
```

```
  @person.addresses << @address
```

```
end
```

- Cannot handle XA transactions

Security

- CSRF defense included by default
- HTTP authentication included by default
- Many plugins out there
 - RESTful authentication seems to be a favorite
 - it has a generator extension to generate a “user” class (with remember me, password hashing, etc.)
 - it emits code in lib/ to handle logins, track the current user, etc.
 - provides hooks for authorization (e.g. check whether user is an admin for a particular method)
 - Another plugin to force SSL for certain requests (e.g. the login or credit card)

Plugins

- Extended functionality that you can include in your application
- Install with `script/plugin install some_url`
- Plugin ends up in `vendor/plugins/...`
- May make libs available, may add generate or rake targets, often includes self tests, etc.
- Can copy directly from `vendor/plugins` to another project but then you lose any install-time behavior

Topics for Another Day

- The many web servers available, including Apache integration
- Deployment (applying the latest code and DB migrations to the production box)
- Clustering and caching for performance
- XML integration with RESTful Rails
- AJAX, RJS templates, JSON, etc.
- Sending E-mail
- Locking the Rails version (or Edge Rails!)

Questions?

ammulder@chariotsolutions.com

<http://www.chariotsolutions.com/presentations.html>