Introduction to Rails

August 2007 @ Philly on Rails

Aaron Mulder
CTO, Chariot Solutions

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

CHARIOT SOLUTIONS

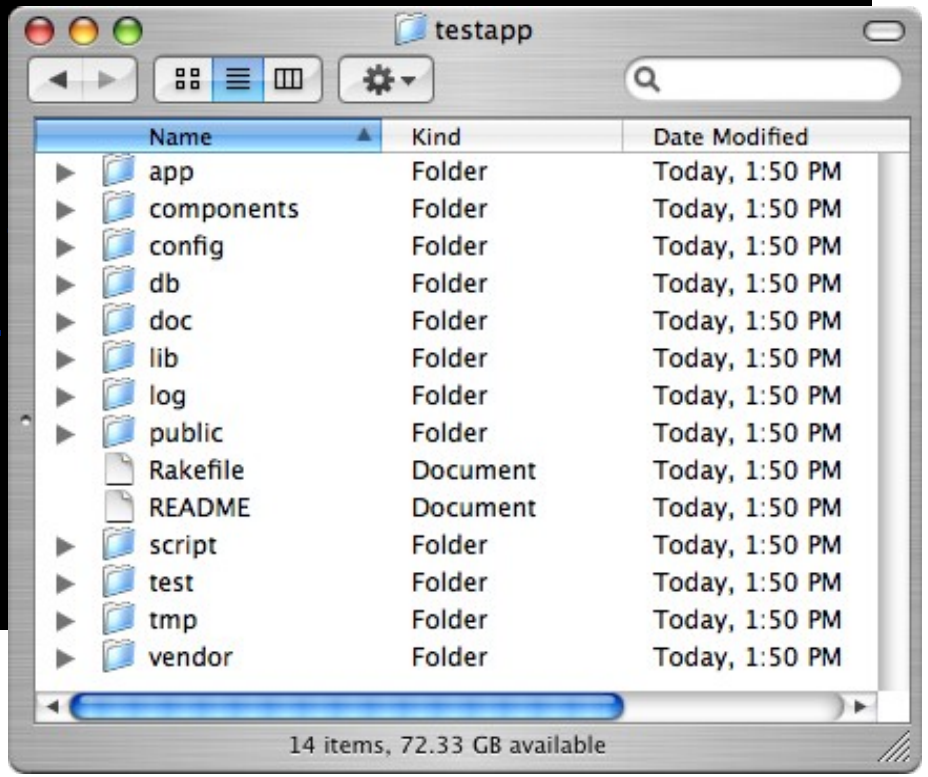
Rails Installation

- Almost no OS includes a working Rails environment (until... say... October 2007)
- 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)
  - Ruby Gems
  - MySQL (or another supported DB) & driver
  - The “rails” Gem (and prerequisites)
  - The “mongrel” Gem
- Just point your path to these first

Create a Project

- Code generation in action

```
> rails testapp
create
create  app/contro
create  app/helper
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
- Some Ruby/Rails specific IDEs
  - Komodo
  - RubyIDE (from the JBuilder folks)
  - Mondrian, ArachnoRuby, others
- Some Java IDEs with Rails support
  - Eclipse RDT / Aptana RadRails
  - NetBeans 6 milestone with Ruby module
  - jEdit with Ruby plugin & project view plugin
  - IDEA EAP with Ruby plugin

Why an IDE?

- Syntax highlighting
- Auto-indentation
- Maybe some code completion?
- 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"

CHARIOT SOLUTIONS

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 is the stereotypical Rails DB, but many others are now supported
- Once you've got drivers, edit **`config/database.yml`**

config/database.yml

```
development:
    adapter: mysql
    database: testapp_development
    username: root
    password:
    socket: /tmp/mysql.sock
test:
    adapter: mysql
    database: testapp_test
    username: root
    password:
    socket: /tmp/mysql.sock
production:
    adapter: mysql
    database: testapp_production
    username: root
    password:
    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 “accounts” and “people”
  - Foreign keys like “account_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

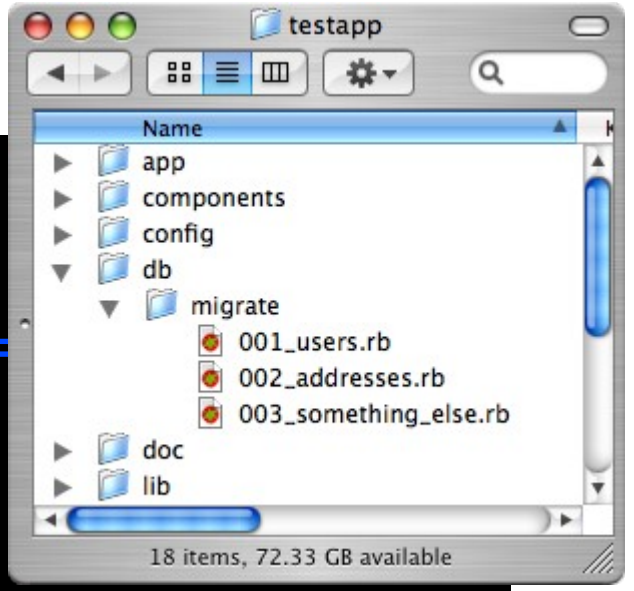
Migrations Code

```
class Users < ActiveRecord::Migration
  def self.up
    create_table :users 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 :users
  end
end
```

Migrations Usage

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



Digression: Rake

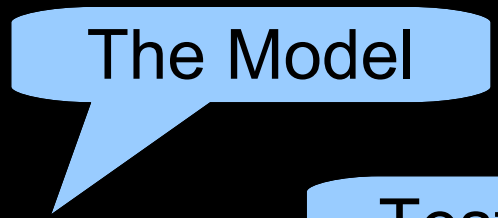
- "Make" for Rails?
  - Just a handy tool with some useful "tasks"
- We just saw the "db:migrate" task
- Try `rake --tasks` to see what's available
- Normal syntax is "rake task arguments"
  - As in `rake db:migrate VERSION=3`
- Can write your own rake tasks and put them in `lib/tasks`
- Will use rake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

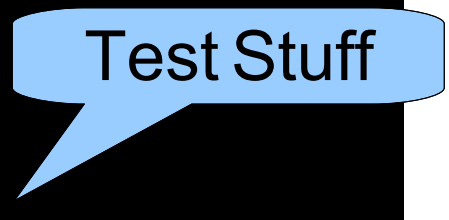
DB Models

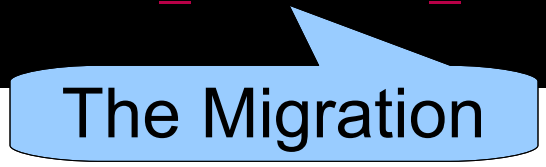
- Based on the ActiveRecord library
- A lot is done automagically
  - properties for each column
  - finders for arbitrary columns or combinations
  - relationships based on simple keywords
- You won't get code completion, but a huge number of things “just work”
- We'll start by generating a model
  - 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 user
exists  app/models/
exists  test/unit/
exists  test/fixtures/
create  app/models/user.rb
create  test/unit/user_test.rb
create  test/fixtures/users.yml
exists  db/migrate
create  db/migrate/001_create_users.rb
```







Digression: script/generate

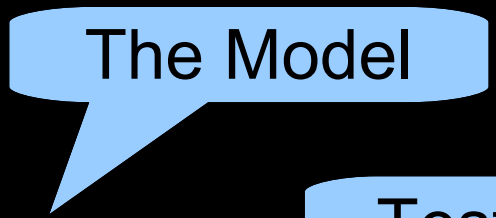
- Used for generating basic code
  - We just saw it for a DB model (plus migration, unit test, test data)
  - Will later see it for all of that plus front end stuff too (controller, views, functional tests)
  - Some plugins add generation options
- Common syntax is (or try --help):

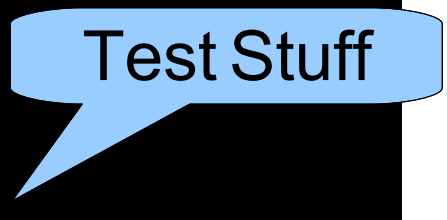
```
script/generate [generator] [class name]
  [other args]
```

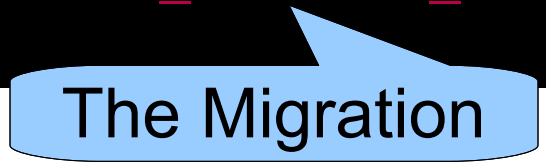
```
> script/generate model user
```

script/generate Output

```
> script/generate model user
exists  app/models/
exists  test/unit/
exists  test/fixtures/
create  app/models/user.rb
create  test/unit/user_test.rb
create  test/fixtures/users.yml
exists  db/migrate
create  db/migrate/001_create_users.rb
```







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 User < 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 or create a hash of property values to set on the new object

Typical “Create” Code

```
def new
  @user = User.new # Empty user for create form
end

def create
  @user = User.new(params[:user]) # Now with data
  if @user.save
    flash[:notice] = "User #{@user.name} created!"
    redirect_to :action => 'list'
  else
    render :action => 'new'
  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 User < 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
- 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

Models with Relations

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

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

```
@user = User.create!(...)
@user.addresses.find(:first, :conditions =>
                        ["type=?", "work"])
@address = Address.new(...)
@user.addresses << @address
puts "User is #{@address.user}"
```

Playing with Models

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

script/console

- Just like “irb” except it includes the whole Rails environment and your application code
- Often used to try out controllers 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/*.yml`**
  - 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

```
aaron:
  id: 1
  username: ammulder
  first_name: Aaron
  last_name: Mulder
  reg_date: <%= Time.now %>
bob:
  id: 2
  username: bob
  first_name: Bob
  last_name: Smith
  reg_date: <%= 2.days.ago %>
```

Sample Unit Test

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

class UserTest < Test::Unit::TestCase
  fixtures :users, :addresses

  def test_find
    user = User.find_by_username('ammulder')
    assert_equal user, users(:aaron)
  end

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

Dive into Rails: "Front End"

CHARIOT SOLUTIONS

Views and Controllers

- URLs from incoming requests are mapped to controller methods and key parameters
- Normally /controller/action/id ...
- so /user/show/2
  - UserController
  - show method of UserController
  - params[:id] = 2
- A view is associated with each controller method (show.rhtml for the show method)
- *Different mappings are used for RESTful Rails, which we'll visit later*

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 user
```

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 user
> script/server
```

  - Visit http://localhost:3000/users/
  - Try making a quick change to a view and refresh the page!
    - However, model changes require restart

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 (@user) are visible to the view

“Create” Code Again

```
class UserController < ApplicationController
  def new    # handles /user/new
    @user = User.new # Empty user for create form
  end

  def create  # handles /user/create
    @user = User.new(params[:user]) # Now with data
    if @user.save
      flash[:notice] = "User #{@user.name} created!"
      redirect_to :action => 'list'
    else
      render :action => 'new'
    end
  end
  ...
end
```

Viewing Users

```
class UserController < ApplicationController
  ...
  def list  # handles /user/list
    @user_pages, @users = paginate(
                :users, :per_page => 10,
                :order => "last_name, first_name")
  end

  def show  # handles /user/show/14
    @user = User.find(params[:id])
  end
end
```

Views

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

Viewing a User

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

<%= link_to 'Edit', :action => 'edit',
                        :id => @user %> |
<%= link_to 'Back', :action => 'list' %>
```

Showing Nested Data

```
<tr>
  <th>My Buddies</th>
  <td>
    <%= @user.buddies.empty? ? "None set" : ""%>
    <% @user.buddies.each do |buddy| %>
      Buddy: <%= buddy.first_name + " " +
                 buddy.last_name %><br />
    <% end %>
  </td>
</tr>
```

Editing a User

```
<h1>Editing user</h1>

<% form_tag :action => 'update', :id => @user do %>
  <%= render :partial => 'form' %>
  <%= submit_tag 'Edit' %>
<% end %>

<%= link_to 'Show', :action => 'show',
                    :id => @user %> |
<%= link_to 'Back', :action => 'list' %>
```

Partials

- Reusable chunks of view (remember DRY)
- named starting with _ but referred to without it
- e.g. `render :partial => foo` looks for _foo.rhtml in the views directory
- Typically several pages include the same partial
- Can pass an object or a collection to a partial if the instance variables won't always be named the same

_form.rhtml for Add/Edit

```
<%= error_messages_for 'user' %>

<p>First name<br/>
<%= text_field 'user', 'first_name'  %></p>

<p>Change Password<br/>
<%= password_field 'user', 'password' %></p>

<p>Confirm New Password<br/>
<%= password_field 'user',
                   'password_confirmation' %></p>
<p>Birthday<br/>
<%= date_select 'user', 'birthday',
      :start_year=>1900, :end_year=>Time.now.year,
  :include_blank=>true, :order=>[:month, :day, :year]
%>
```

Other Form Options

- With form_for, can omit the variable name from each field

```
<% form_for :user, :url => {:action => 'update'}
                                    do |form| %>
  <p>First name<br/>
  <%= form.text_field 'first_name'  %></p>

  <p>Change Password<br/>
  <%= form.password_field 'password' %></p>

  <%= submit_tag 'Edit' %>
<% end %>
```

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.rhtml`** 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

CHARIOT SOLUTIONS

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
- Can specify a controller to handle requests to /

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

```
User.transaction do
  @user = User.create(...)
  @address = Address.create(...)
  @user.addresses << @address
end
```

- Cannot handle XA transactions

Security

- No security included by default
- Many plugins out there
  - acts_as_authenticated seems to be a favorite
    - it has a generator extension to generate a “user” class
    - 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)
    - handles “remember me” and password hashing and so on
  - 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
- RESTful Rails and Web Services
- AJAX, RJS templates, JSON, etc.
- Sending E-mail
- Using Rails Edge / 2.x

Questions?

ammulder@chariotsolutions.com

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

CHARIOT SOLUTIONS