

# Introduction to Rails

Dominic Mitchell

<http://happygiraffe.net/blog/>

SkillSwap, 2007-04-05

# Who Am I

- I do Java and Perl for a living.
- Rails is a hobby... But I love it!
  - Done some demos, apps at home.
  - Spoke at RailsConf Europe

# The Plan

- What is Rails?
- Why it works
- Introduction to Ruby
- Practical example

# What is Rails?



# What is Rails?

Ruby on Rails is a framework that makes it easier to develop, deploy and maintain web applications.



*“Agile Web Development with Rails”*

# What is Rails?

Ruby on Rails is astounding. Using it is like watching a kung-fu movie where a dozen bad-ass frameworks prepare to beat up on the little newcomer only to handed their asses in a variety of imaginative ways.

*Nat Torkington, O'Reilly OSCON chair*

# What is Rails?

- Web
- Application
- Framework

Basically, it's another way of building web applications. Probably done this already using ASP, PHP, JSP, etc. Framework? It calls you, instead of you calling into a library to get things done. You **could** call Apache+PHP a framework. But it's quite low level. Rails is much higher level.

# Rails is different

- Simpler
- Quicker
- Better ~~hyped~~  
marketed

Simpler than say, J2EE. Quicker dev time, anyway. Rails cuts the crap. This is in large part due to its use of Ruby.

Yes, it's also over hyped. We'll get to that in a bit.

# Rails features

- MVC framework
- Written in Ruby
- ActiveRecord
- Portable
- Integrated JavaScript & Ajax
- Great testing support
- Migrations
- Excellent URL handling
- Web services
- Caching
- Plugins
- No pony

# History

- 37 Signals (basecamp, tadalist, backpack &c.)
- Rails *extracted* from basecamp in 2004 by David Heinemeier-Hansson
- Now at v1.2



Why Rails works

# The Rails philosophy

- To understand Rails, you need to understand the notions it's built upon

# Opinionated Software

It believes that you should do things a certain way, and will try to encourage you down that path.

e.g. there is a standard directory layout; this reinforces MVC: controllers live in `app/controllers`.  
Everything in its place and a place for everything.

# Convention over configuration

aka “sensible defaults.” Try to avoid configuration where possible. Avoid the Java/XML plague. Does not mean no configuration!  
e.g. Rails assumes your primary key column is named “id” and is an integer.  
e.g. Rails assumes that the table name is the plural name of the model.

# Don't Repeat Yourself

(from the pragmatic programmer book). This stems from Ruby's flexibility. Cut'n'paste is **evil**. Leads to Rails code having a high signal-to-noise ratio.

e.g. no duplication of the schema in code; e.g. URL generation is all in one place.

# Agile

Very strong support for tests. Quick turnaround time for changes. Rails tries to keep projects flexible and amenable to change. Designed to help you be customer-focused.

# So why does Rails work?

- Because it's quick to develop in
- Because it's *fun* to develop in

Compared to the heavyweight J2EE stuff, it's quick to code. Compared to the low end PHP stuff, it gives you more support and you don't get bogged down further on. Ultimately, you end up having a lot of "that's neat" moments with Rails.

Why not Rails?

# Performance

- Ruby is still a fairly slow language
- This doesn't matter, *most* of the time
- Ruby (and JRuby) are getting faster

The key thing is that Rails is optimised for developer time. JRuby is a version of Ruby that runs on the JVM.

# Deployment

- Not as simple as PHP
- Shared hosting also not as simple
- But, capistrano really helps

capistrano is a tool for copying your code onto a server, and stopping and restarting the web server.

# Ruby



# Ruby

- Another “scripting” language
- More-or-less similar to PHP / Perl / Python
- Scary to Java programmers

The syntax is different, but the underlying ideas are mostly similar.

If you're coming from Java or C#, the lack of strong typing will feel weird. Don't worry, that feeling goes away.

# Ruby Overview

I'm going to go through a few features of the language to try and help you read code that's written in Ruby.

```
foo = 1 + 1
```

```
bar = "foo is #{foo}"
```

Note the lack of type declarations and sigils. Also, note the interpolation style.

```
mylist = [1, 2, 3]
```

Array construction is much more concise than PHP or Java.

myhash = { "answer" => 42 }

```
def hello(whom)
  puts "hello, #{whom}"
end
```

Again, no types declared. The return value of a function is the last statement executed.

```
class Hello
  attr_reader :name
  def initialize(name)
    @name = name
  end
  def greet
    puts "hello, #{name}"
  end
end
```

NB: initialize is the constructor. “:name” is a symbol (like a string, there are lots of these in Rails).

attr\_reader is the first example of **magic!** More in a slide or two...

```
h = Hello.new('dom')  
h.greet()
```

Usage of the previous class.

# attr\_reader

- Automatically writes methods
- Magic!
  - aka “Metaprogramming”

Your code

```
attr_reader :name
```

Actual code

```
def name()  
  return @name  
end
```

# What's in a name?

Symbol	Meaning
foo	Local
\$foo	Global
@foo	Instance
FOO	Constant

You don't see globals that much...

# Blocks

```
list = [1, 2, 3]
```

```
list.each do |n|
```

```
  puts "got: #{n}"
```

```
end
```

Ruby's killer feature. Note that I haven't shown any loops yet.

More positive example: open takes a block and automatically closes the filehandle afterwards.

# Rails Practical

# Getting Started

```
% rails -d sqlite3 squawkr  
  create  
  create  app/controllers  
  create  app/helpers  
  create  app/models  
  create  app/views/layouts  
  create  config/environments  
  
...
```

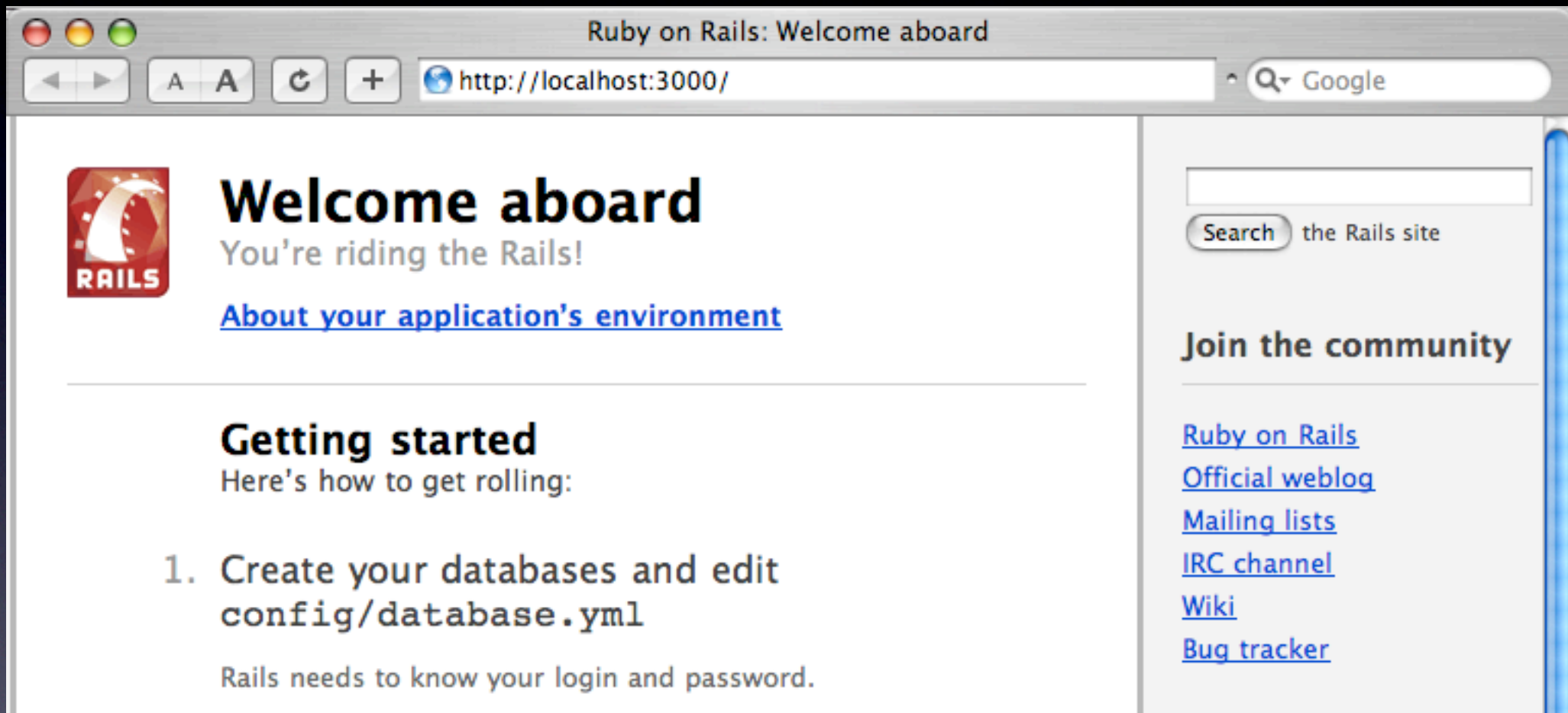
Lays out the standard directory structure: app, config, script and so on.

On the Mac, use SQLite. If you're using Instant Rails, leave it out and it'll pick MySQL.

# Try it out!

```
% script/server  
=> Booting WEBrick...  
=> Rails application started on http://0.0.0.0:3000  
=> Ctrl-C to shutdown server; call with --help for options  
[2007-03-28 22:39:12] INFO WEBrick 1.3.1  
[2007-03-28 22:39:12] INFO ruby 1.8.6 (2007-03-13) [...]  
[2007-03-28 22:39:12] INFO WEBrick::HTTPServer#start: ...
```

Ruby comes with a builtin web server for development. It's your own, personal copy.



The default rails welcome page.

# Start the Model

```
% script/generate model message
  exists app/models/
  exists test/unit/
  exists test/fixtures/
  create app/models/message.rb
  create test/unit/message_test.rb
  create test/fixtures/messages.yml
  create db/migrate
  create db/migrate/001_create_messages.rb
```

Look! It made tests for you! Have a look at `app/models/message.rb`, that's the main code. Also, check out the migration file.

# Table Definition

```
def self.up
  create_table :messages do |t|
    t.column :message, :string
    t.column :whom, :string
    t.column :created_on, :datetime
  end
end
```

*db/migrate/001\_create\_message.rb*

Before anything else, we need a database structure. Look Ma, no SQL! Note that there is also a “down” method in there for rolling back changes. This is a good example of a “DSL” (Domain Specific Language).

# Create Table

```
% rake db:migrate  
(in /Users/dom/Documents/squawkr)  
== CreateMessages: migrating ==  
-- create_table(:messages)  
   -> 0.0839s  
== CreateMessages: migrated (0.0879s) ==
```

database independent. Migrations let you go backward as well as forwards.

# Interaction

```
% script/console
```

```
Loading development environment.
```

```
>> Message.find(:all)
```

```
=> []
```

```
>> msg = Message.create(:message => "hello")
```

```
=> #<Message:0x30f6ad8 ...>
```

```
>> msg.message
```

```
=> "hello"
```

```
>> msg.created_on
```

```
=> Fri Mar 30 07:05:10 +0100 2007
```

Lets you play with your data interactively. You can also look inside other parts of your application.

NB: See how rails uses plurals / singular.

NB: Missing "whom" field.

# Starting The App

```
% script/generate scaffold Message  
exists app/controllers/  
exists app/helpers/  
create app/views/messages  
exists app/views/layouts/  
exists test/functional/
```

...

Generates a really simple CRUD controller. Good for a starting point. Looks hideous.

Let's take a look at the generated controller.

Messages: index



http://localhost:3000/messages



Google

# Listing messages

**Message Whom**

**Created on**

hello

Fri Mar 30 07:05:10 +0100 2007 [Show](#) [Edit](#) [Destroy](#)

[New message](#)

# How'd that work?

- `config/routes.rb`
- `app/controllers/messages_controller.rb`
- `app/views/messages_controller/list.rhtml`

routes specifies how to parse the URL. controller interprets the request and forwards to the view. the view just renders HTML.

# Scaffold

- Scaffold is a good start for CRUD apps...
- ... But we're building something simpler.

# SquawkController

```
% script/generate controller Squawk  
  exists app/controllers/  
  exists app/helpers/  
  create app/views/squawk  
  exists test/functional/  
  create app/controllers/squawk_controller.rb  
  create test/functional/squawk_controller_test.rb  
  create app/helpers/squawk_helper.rb
```

# SquawkController

```
class SquawkController < ApplicationController  
end
```

*app/controllers/squawk\_controller.rb*

Look in `app/controllers/squawk_controller.rb`. Hmm, that's a bit empty!

# SquawkController

```
class SquawkController < ApplicationController
  def index
  end
end
```

*app/controllers/squawk\_controller.rb*

Better... NB: Methods in a controller are known as “actions”.

# SquawkController

```
class SquawkController < ApplicationController
  def index
    @messages = Message.find(
      :all,
      :order => 'created_on DESC',
      :limit => 5
    )
  end
end
```

*app/controllers/squawk\_controller.rb*

In order to display the page, we need the 5 most recent messages.

NB: In a real app, I would probably put a method on the Message class to run this find.

# Squawk Index View

```
<h1>Recent Messages</h1>
<ul>
  <% for msg in @messages %>
    <li>
      <em><%=h msg.whom %></em>
      said
      <q><%=h msg.message %></q>
    </li>
  <% end %>
</ul>
```

*app/views/squawk/index.rhtml*

This is `app/views/squawk/index.rhtml`. NB: `<% %>` vs `<%= %>` (the latter outputs into the page). Also, the “h” function, which escapes HTML. I have no idea whatsoever why the Rails people didn’t turn that on by default.

# New Messages

```
<% form_for :message,  
           :url => { :action => 'add' } do |f| %>  
  <%= f.text_field :whom, :size => 10 %>  
  says  
  “<%= f.text_field :message %>”  
<% end %>
```

*app/views/squawk/\_form.rhtml*

We'll create a “partial”, Rails' way of splitting up HTML pages. Why use the rails helpers? They get the name / ID attributes correct, so that the rest of the framework knows what to expect. This form POSTs to /squawk/add.

# New Messages

```
<h1>Recent Messages</h1>
<ul>
  ...
</ul>
<%= render :partial => 'form' %>
```

*app/views/squawk/index.rhtml*

Adjust index.rhtml to point at the view.

# New Messages

```
class SquawkController < ApplicationController
  def add
    @message = Message.new(params[:message])
    if @message.save
      redirect_to :action => "index"
    else
      index
      render :action => "index"
    end
  end
end
```

*app/controllers/squawk\_controller.rb*

We need to add some control to the controller too. With this, the form should be hooked up and working. We make an object and try to save it to the DB. If it works, go to the index page. If it fails, display the current page, but because we have the problem object available, we can display errors. This will make more sense in a minute...

# Validation

```
class Message < ActiveRecord::Base
  validates_presence_of :message, :whom
end
```

*app/models/message.rb*

Oops, I've just realised that we don't actually require any message. Or username! But Rails can take care of that for us. But that doesn't display an error.

# Validation

```
<% form_for :message,  
           :url => { :action => 'add' } do |f| %>  
  <%= error_messages_for :message %>  
  <%= f.text_field :whom, :size => 10 %>  
  says  
  “<%= f.text_field :message %>”  
<% end %>
```

*app/views/squawk/\_form.rhtml*

`error_messages_for()` will add in a div with all errors (which needs a lot of styling). This is dependent on having an `@message` variable, which contains the errors. That's why we did render instead of redirect in the controller. This isn't perfect for your app, but it's a good starting point.

# Refactoring Time

```
<li>
  <em><%=h message.whom || 'Somebody' %></em>
  said
  <q><%=h message.message %></q>
</li>
```

*app/views/squawk/\_message.rhtml*

Let's clean up a little bit. The loop showing the page should go into its own partial. NB: renamed msg to message.

# Refactoring Time

```
<h1>Recent Messages</h1>
<ul id="messages">
  <%= render :partial => 'message',
           :collection => @messages %>
</ul>
<%= render :partial => 'form' %>
```

*app/views/squawk/index.rhtml*

There was no need for that for loop after all!

# Ajax

- Let's try adding a bit of spice.
- Make the form submit and update the current page.

Pretty trivial and not really needed in this case. But it's a demo...

# Ajax

```
<%= javascript_include_tag :defaults %>
<h1>Recent Messages</h1>
<ul id="messages">
  <%= render :partial => 'message',
            :collection => @messages %>
</ul>
<%= render :partial => 'form' %>
```

*app/views/squawk/index.rhtml*

Pull in Prototype and Script.aculo.us. Not always a good idea (they're fairly large), but OK for now.

# Ajax

```
<% remote_form_for :message,  
  :update => 'messages',  
  :success => 'new Effect.Highlight("messages")',  
  :url => { :action => 'add' } do |f| %>  
  <%= error_messages_for :message %>  
  <%= f.text_field :whom, :size => 10 %>  
  says  
  “<%= f.text_field :message %>”  
<% end %>
```

*app/views/squawk/\_form.rhtml*

Switch to `remote_form_for()` and pass the id of an element to update in the page. The “success” parameters specifies a “yellow fade”. For more complex JavaScript, you’d probably want to define a JavaScript function.

# Ajax

```
class SquawkController < ApplicationController
  def add
    @message = Message.new(params[:message])
    if @message.save
      if request.xhr?
        index
        render :partial => "message",
              :collection => @messages
      else
        redirect_to :action => "index"
      end
    else
      index
      render :action => "index"
    end
  end
end
```

*app/controllers/squawk\_controller.rb*

We can use `request.xhr?` to see if we're looking at an ajax request, and respond accordingly. Here, we render the contents of the messages div. This is starting to get really ugly and repetitive (and the error handling is broken). Best to refactor the Ajax bits into a separate action, perhaps.

# What next?

- Thanks for listening!
- [rubyonrails.org](http://rubyonrails.org) - docs, screencasts
- [Agile Web Development with Rails](#) (book)
- [Brighton-Ruby.org](http://Brighton-Ruby.org)