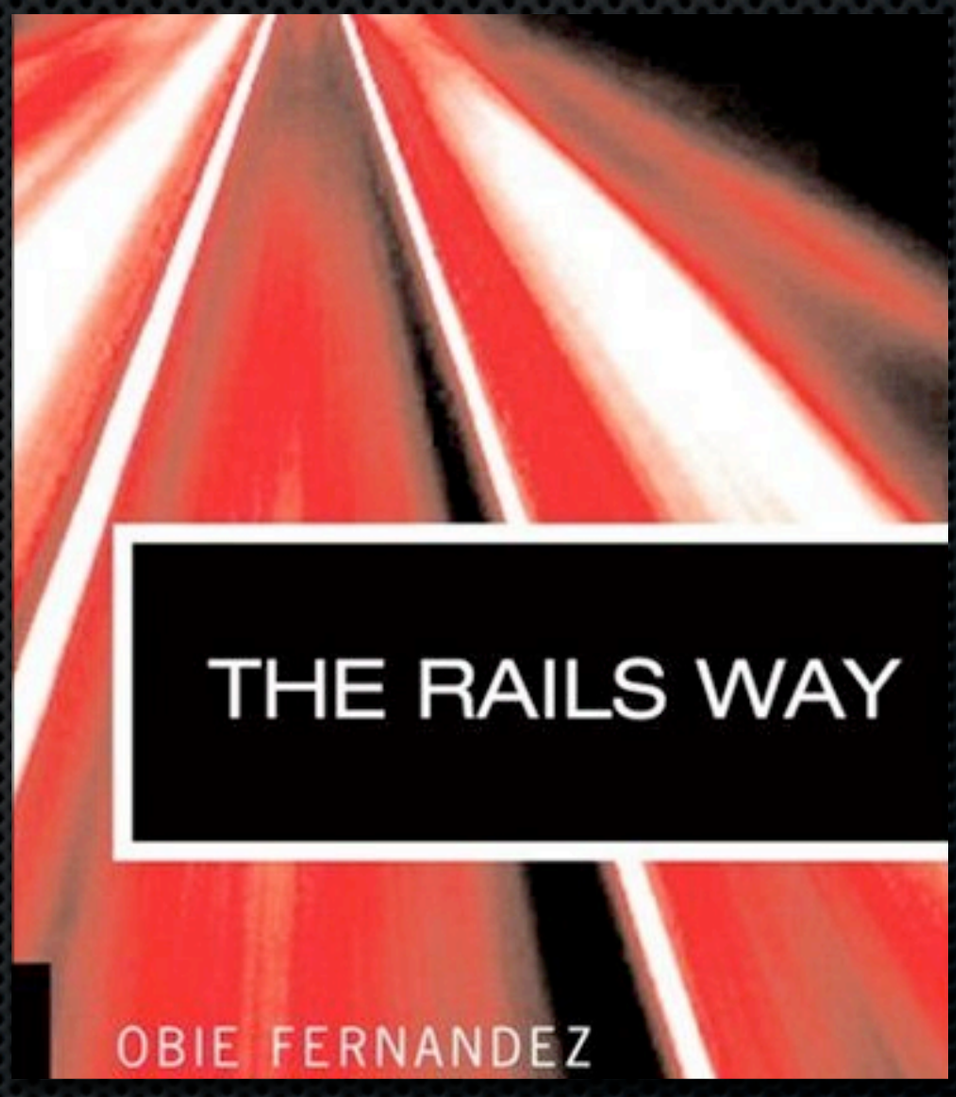
Rails Controllers and Routing

Obie Fernandez

Prepared exclusively for
RubyEast 2007



The controller layer is what really sets Rails apart from everyone else.

Like any computer program, your Rails application involves the flow of control from one part of your code to another. The flow of program control gets pretty complex with Rails applications. There are many bits and pieces in the framework, many of which execute each other. And part of the framework's job is to figure out, on the fly, what your application files are called and what's in them, which of course varies from one application to another.

ActiveRecord can stand on its own and often does, and ActionView is to a large degree just a lot of fanciness on top of Ruby's built-in templating system, ERb.

The Dispatcher

Rails is used to build Web-based applications, so before anything else happens, and for anything that does happen, a Web server[md]Apache, Lighttpd, Nginx, and so on [md]handles a request. The server then forwards that request to the Rails application, where it is handled by the dispatcher.

Request Handling

- Web server provides request info
  - Http method (e.g. GET, POST)
  - URI (e.g. “demo/index”)
  - Parameters (if any)
- The dispatcher’s job is to:
  - Figure out which controller and action to execute
  - Load the controller file and instantiate the controller
  - Invoke the requested action

When a request comes in, the server interacts with the Rails dispatcher and gives it the URI and parameters to work with...

All of this happens quickly, behind the scenes. It’s unlikely that you would ever need to dig into the source code for the dispatcher; it’s the sort of thing that you can take for granted to just work.

Demo

- We'll create a demo rails app
- Tweak one of the views
- Fire up script/console
- Manually trigger a dispatch

The Controller

```ruby
# implicit behavior, by convention
def index
  render :template => "demo/index"
end
```

When in Doubt, Render

Rails doesn't need an explicit controller action if there is a template that matches the request

What you learn from seeing the empty action is that, at the end of every controller action, if nothing else is specified, the default behavior is to render the template whose name matches the name of the controller and action. In this case, that means app/views/demo/index.rhtml.

In other words, every controller action has an implicit render command in it. And render is actually a real method. You could write the preceding example like the code shown.

You don't have to, though, because it's assumed that that's what you want, and that is part of what Rails people are talking about when they discuss convention over configuration. Don't force the developer to add code that could simply be assumed by convention.

Many Kinds of **render**

- render :action => ‘new’
- render :template => ‘abuse/report’
- render :file => ‘/shared/templates/global.html.erb’
- render :partial => ...
- render :inline => “<%= auto_complete_result(...)”
- render :text, :json, :xml
- render :nothing => true, :status => 401

Rendering Options

- :content_type
- :layout
- :status

All content flying around the web is associated with a MIME type2. For instance, HTML content is labeled with a content-type of text/html. However, there are occasions where you want to send the client something other than HTML. Rails doesn't validate the format of the MIME identifier you pass to the :content_type option, so make sure it is valid.

By default, Rails has conventions regarding the layout template it chooses.

The HTTP protocol includes many standard status codes3 indicating a variety of conditions in response to a client's request. Rails will automatically use the appropriate status for most common cases, such as 200 OK for a successful request.

Preventing Double Renders

Make sure to return from the controller action if you render or redirect.

```
def show
  @user = User.find(params[:id])
  if @user.activated?
    render home_path and return
  end

  case @user.info
    ...

    redirect_to setup_user_path(@user)
  end
end
```

Controller/View Communication

There's a bit of irony and possible confusion in the choice of instance variables to share data between the controller and view templates.

The main reason that instance variables exist is so that objects (whether Controller objects, String objects, and so on) can hold on to data that they don't share with other objects.

When the view template is rendered, the context is that of a different object, an instance of ActionView::Base. That instance has its own instance variables, and does not have access to those of the controller object.

So instance variables, on the face of it, are about the worst choice for a way for two objects to share data. However, it's possible to make it happen[md]or make it appear to happen. What Rails does is to loop through the controller object's variables and, for each one, create an instance variable for the view object, with the same name and containing the same data.

If you're a Ruby purist, you might wince a little bit at the thought of instance variables serving to connect objects, rather than separate them. On the other hand, being a Ruby purist should also include understanding the fact that you can do lots of different things in Ruby[md]such as copying instance variables in a loop. So there's nothing really un-Ruby-like about it. And it does provide a seamless connection, from the programmer's perspective, between a controller and the template it's rendering.

Controller Filters

Macro-style pre and post processing

```
before_filter :admin_required, :only => [:destroy]
before_filter :activation_required, :only => [:review, :new_reply_to_review, ...
before_filter :login_required, :except => [:named, :index, :show, ...
before_filter :lookup_listing, :except => [:named, :index, :new, ...
before_filter :check_ownership, :only => [:edit, :update, :destroy]
```

Multiple Filters

A real-world example

```ruby
before_filter :load_product, :only => [ :show, :edit, :update, :destroy ]

def load_product
   @product = current_user.products.find_by_permalink(params[:id])
   redirect_to :action => 'index' and return false unless @product.active?
end
```

Loading Data in a Filter

Contentious Issue

Some of us like to use before filters to load the records for single-record operations, where there is some kind of complex logic. Instance variables set in a filter are of course available to any actions.

This is a contentious issue; some developers believe that database actions should stay out of filters and be specified in the action method.

```ruby
class ApplicationController < ActionController::Base
  include AuthenticatedSystem
  include ExceptionNotifiable
  include InvitationHandling

  before_filter :admin_required, :only => :admin
  before_filter :login_from_cookie
  after_filter  :compress

  ...
```

Filter Inheritance

Shared with Subclasses

```ruby
class OutputCompressionFilter
  def self.filter(controller)
    controller.response.body = compress(controller.response.body)
  end
end

class ApplicationController < ActionController::Base

  after_filter OutputCompressionFilter

  ...

end
```

Filter Classes

Reuse common filter functionality

Routing

The routing system in Rails is the system that examines the URL of an incoming request and determines what action should be taken by the application. And it does a good bit more than that. Rails routing can be a bit of a tough nut to crack. But it turns out that most of the toughness resides in a small number of concepts. Once you've got a handle on those, the rest falls into place nicely.

Two Purposes of Routing

- Recognizes URLs so that the right controller and action can be invoked
- Constructs URLs for you for use as arguments to methods like **link_to**, **redirect_to**, and **form_tag**

Routing Rules

- Written in Ruby code
- Stored in **config/routes.rb**
- Consist of a name, match pattern and parameters
- Added in priority-descending order

```
map.connect 'myrecipes/:ingredient',
            :controller => "recipes",
            :action => "show"
```

Bound Parameters

“Hardcoded route”

The :controller and :action keys are bound: This route, when matched by a URL, will always take the visitor to exactly that controller and that action.

```
map.activate '/activate/:key',
             :controller => 'people', :action => 'activate'

map.connect ':controller/:action/:id'
```

Wildcard Parameters

You can think of them as “receptors”

```
map.activate '/activate/:key',
             :controller => 'people',
             :action => 'activate'
```

Static Strings

Anchor the route

The “activate” string anchors the recognition process. When the routing system sees a URL that starts /activate, it will match that to the static string in the activate route. Any URL that does not contain the static string “activate” in the leftmost slot will not match this route.

```ruby
# in config/routes.rb
map.connect ':controller/:action/:id.:format'


# in an Item controller
def show
  @item = Item.find(params[:id])
  respond_to do |format|
    format.html
    format.xml { render :xml => @item.to_xml }
  end
end
```

The Ante-Default Route

Works with **respond_to** in the controller

```
# You can have the root of your site routed by hooking up ''
# -- just remember to delete public/index.html.
map.connect '', :controller => "welcome"
```

The Empty Route

Works with **respond_to** in the controller

You're usually safe leaving the default route alone. But there's another route in routes.rb that plays something of a default role and you will probably want to change it: the empty route.

The empty route says, "I don't want any values; I want nothing, and I already know what controller and action I'm going to trigger!

```
map.connect 'files/list/*specs'
```

Route Globbing

The greediest route

In some situations, you might want to grab one or more components of a route without having to match them one by one to specific positional parameters. For example, your URLs might reflect a directory structure. If someone connects to