

All You Ever Wanted To Know About Routing And Never Dare To Ask

Luca Mearelli

14 September 2006

Beautiful applications, Attention to details

Designing applications to succeed and give users an enticing experience while remaining usable and simple to understand requires attention to details.

URL Usability

Something that we usually don't pay enough attention to is the usability of the URL scheme we adopt for our web applications.

A RESTful face

URLs should have a clear and precise meaning in the context of the application and should be easily guessed to increase usability and ease the building of RESTful web services on top of the data.

Rails Routing

In this talk we'll see how the Rails routing module implements a simple but effective domain specific language to help in designing the public face of a site by mapping URL to executable code (namely controllers and actions).

These mappings are entirely defined using ruby code (in the routes.rb configuration file) and are platform agnostic (they work alike on any combination of O.S. and web server supporting Rails).



URL
=
Uniform Resource Locator

Back to the basics

URL = Uniform Resource Locator
what's an URL

Importance of the word "**Resource**"

is the address of a digital artifact,
ideally it should be immutable,
It has semantic importance

<http://www.site.com/viewcatalog.asp?category=hats&prodID=53>

<http://www.site.com/catalog/hats/53>

Before routing

Mod_rewrite

Example of ugly url vs nice url

There should be a better way: routing

A DSL for URL mapping

Convention Over Configuration

From Requests To Executable Code

Easy configuration 80% of the time

Very complex possible

Customization relatively easy (with refactored code)

Convention over configuration

Routes come preconfigured on application generation and offer a simple and consistent initial setup which doesn't require any coding to be used. In case it's needed, the developer is free to tweak the routing setup adding custom mappings.

translating URLs to controllers and actions

Each mapping consists of a path definition and a set of options. The path definition describes the url pattern handled by the mapping i.e. how the path should be decomposed in segments and how each segment should be used to build the parameter hash which is then passed to the application controller.

A path segment can define a symbol which will be used as a key in the parameter hash, the value for that key will be extracted from the actual request path using dynamically generated regular expressions. Special symbols are used to drive the request to the correct controller and action.

Routes can also be given names and in this case Rails defines special helper functions to ease the burden of defining those urls inside the application code.

RAILS_ROOT/config/routes.rb

```
ActionController::Routing::Routes.draw
do |map|
  map.connect
    ':controller/service.wsdl',
      :action => 'wsdl'
  map.connect
    ':controller/:action/:id'
end
```

Routing configuration example of default:
The basic route works also without routes.rb

Routing Navigator

```
script/plugin install http://svn.techno-weenie.net  
/projects/plugins/routing_navigator
```

To use on edge rails:

```
ActionController::Routing.controller_paths <<  
  RAILS_ROOT +  
  '/vendor/plugins/routing_navigator/lib'
```

Add these to the page where you want to see the routing information:

```
<%= javascript_include_tag 'routing_navigator' %>  
<%= stylesheet_link_tag 'routing_navigator' %>
```

To update the project with the required javascript:

```
rake update_routing_navigator
```

To use only in part of your controller:

```
class FooController < ApplicationController  
  routing_navigator :off # turns the routing navigator off completely  
  routing_navigator :only => [:show, :edit]  
  routing_navigator :except => [:create, :update, :destroy]  
end
```

Segments: Static, Dynamic, Path

```
map.connect 'entry/:id',
           :controller => 'entries',
           :action => 'show'

map.connect 'my/clients',
           :controller => 'contacts',
           :action => 'list',
           :kind=>'clients'
```

Each mapping consists of a path definition and a set of options. The path definition describes the url pattern handled by the mapping i.e. how the path should be decomposed in segments and how each segment should be used to build the parameter hash which is then passed to the application controller.

A path segment can define a symbol which will be used as a key in the parameter hash, the value for that key will be extracted from the actual request path using dynamically generated regular expressions. Special symbols are used to drive the request to the correct controller and action.

Static segments are path sections (divided by a separator, e.g. “/”) which are not put into the parameter hash.

Dynamic segments are path sections which are used to catch part of the path into the parameter hash. Each dynamic segment defines a symbol.

Special symbols

:controller identifies/extracts a controller name. Some special checks are done to insure that the name corresponds to a controller class (e.g. looking up into some defined directories)

:action identifies/extracts an action name

:id identifies/extracts an id (it can be anything, a number or other string in that position on the path)

```
map.with_options(:controller => 'invoices',
                :action => 'list')
  do |route|
    route.connect 'invoices/pending',
                  :status => 'pending'
    route.connect 'invoices/pastdue',
                  :action => 'pastdue'
  end
```

Using the `with_options` we can make the definition of similar routes much more compact.

The options defined within the `with_options` parameters are applied to all the route defined inside the block.

Default route

```
map.connect '', :controller => 'home',  
             :action => 'index'
```

Default parameters

```
map.connect ':controller/:action/:id',  
            :defaults =>  
              { :controller => 'home'  
                :action => 'index' ,  
                :id => nil }
```

Default route: it is used to define the controller which is in charge of responding to the root url of the application (eventually with all the other related parameters e.g. action & id)

Default parameters: are the values used for a parameter should it not be present in the path (e.g. calling a url without the action and id defined)

Named routes

```
map.login 'login', :controller =>
  'account', :action => 'login'
map.signup 'signup', :controller =>
  'account', :action => 'signup'
map.logout 'logout', :controller =>
  'account', :action => 'logout'
```

Url helpers

```
link_to 'log in!', login_url
link_to 'log out!', logout_url
```

You can give urls names using the map.name call

They are useful when defining links in the view since you can obtain them using a name_url helper function

Route requirements

```
map.connect 'articles/:year/:month/:day',
  :controller => 'articles',
  :action     => 'find_by_date',
  :year       => /\d{4}/,
  :month      => /\d{1,2}/,
  :day        => /\d{1,2}/
```

Regexp rules for requirements

```
:foobar => /foo|bar/
:ratings => /[0-5](to[0-5])?/
:days   => /\d*/
:tags    => /tagged\/([\^\/]*)/
```

Route requirements can help in defining complex routes since they give a way to condition the recognition of a route to the matching of regular expressions for its path segments.

A route where a path segment defines a requirement is recognized if and only if the corresponding url segment is matched by the regular expression.

Globbering

```
map.path 'file/*path',  
  :controller => 'content',  
  :action => 'show_file',  
  :path => []
```

Route conditions

```
map.connect ':controller/:id/destroy'  
  :conditions => { :method=>:post }
```

A segment can also be defined as a **path segment** that is a segment which matches all the remaining part of the url

When matching a url the path segment is translated into an array of segments (by slicing that part of the url at each path separator occurrence)

Routes can be given also **conditions** which are additional checks to be done when matching the url.

The condition defined in the basic routing code is a condition on the http method used for the request.

RESTful routing

```
map.resources :users

map.resources :users,
  :collection => { :filter => :get }
map.resources :users,
  :member => { :deactivate => :post }
map.resources :users,
  :new => { :admin => :put }

map.resources :groups { |groups|
  groups.resources :users }

filter_users_url
```

Extending routing

Why?

How?

Why? Because it is possible to define new conditions to handle complex routing setups with simpler code.

How? By extending two simple methods to get the data from the request and to use it in the recognition of a url.

Routing Internals

Path Segmentation

Requirements, Conditions, Defaults

Path Expiry

Parameter Shell

Query String

Routing internals: Basics

routes path segmentation

options for routing (requirements,conditions,defaults)

path expiry

parameter shell

query string

Two Phases

Generation

Recognition

Extension points

`extract_request_environment`

`recognition_conditions`

Routing Extension points are the methods which the plugin developer can extend to implement some special case or additional options to be used defining routes.

`extract_request_environment`: here we get access to the raw request object from which we can extract any information we need later and put it into an hash

`recognition_conditions`: here we can interpret the conditions defined in each route and build the code (to be executed in the recognition method). Here we use the environment defined in the other method to get request informations.

A Real Example: Request Routing

```
script/plugin install http://svn.vivabit.net/  
external/rubylibs/request_routing/
```

A Simple Extension (a real example).

Request routing adds some conditions to the routing definition allowing checks on many request elements (e.g. the domain and subdomain)

It is particularly interesting since the current distribution contains both the code rewritten to use the new routing implementation and the old one.

```
class ActionController::Routing::Route

  TESTABLE_REQUEST_METHODS =
    [:subdomain, :domain, :method, :port, :remote_ip,
     :content_type, :accepts, :request_uri, :protocol]

  def recognition_conditions
    result = ["(match =
#{Regexp.new(recognition_pattern).inspect}.match(path))"]
    conditions.each do |method, value|
      if TESTABLE_REQUEST_METHODS.include? method
        result << if value.is_a? Regexp
          "conditions[#{method.inspect}] =~ env[#{method.inspect}]"
        else
          "conditions[#{method.inspect}] === env[#{method.inspect}]"
        end
      end
    end
    result
  end
end
```

```
class ActionController::Routing::RouteSet

  def extract_request_environment(request)
    {
      :method => request.method,
      :subdomain => request.subdomains.first.to_s,
      :domain => request.domain,
      :port => request.port,
      :remote_ip => request.remote_ip,
      :content_type => request.content_type,
      :accepts =>
        request.accepts.map(&:to_s).join(', '),
      :request_uri => request.request_uri,
      :protocol => request.protocol
    }
  end

end
```

Testing Routing

assert_routing

```
def test_movie_route_properly_splits
  opts = {:controller => "plugin",
          :action => "checkout", :id => "2"}
  assert_routing "plugin/checkout/2", opts
end
```

assert_recognizes

```
def test_route_has_options
  opts = {:controller => "plugin",
          :action => "show", :id => "12"}
  assert_recognizes opts, "/plugins/show/12"
end
```

Testing routing

`assert_routing` lets you test whether or not the route properly resolves into options.

Note the subtle difference between the two: `assert_routing` tests that an URL fits options while `assert_recognizes` tests that an URL breaks into parameters properly.

Luca Mearelli

l.mearelli@spazidigitali.com

<http://spazidigitali.com>