




**Addison-Wesley Professional Ruby Series**

---

# Rails Routing

David A. Black

<b>Section 1:</b>	
What This Short Cut Covers.....	3
<b>Section 2:</b>	
Introduction.....	6
<b>Section 3:</b>	
The Routing System.....	13
<b>Section 4:</b>	
Writing Custom Routes.....	31
<b>Section 5:</b>	
Named Routes.....	45
<b>Section 6:</b>	
REST, Resources, Representations, Routing, and Ruby on Rails .....	53
<b>Section 7:</b>	
Reflecting on Routes.....	91
<b>Further Resources.....</b>	<b>112</b>
<b>Acknowledgments.....</b>	<b>113</b>
<b>About the Author.....</b>	<b>114</b>



Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this work, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this work, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

Visit us on the Web: [www.awprofessional.com](http://www.awprofessional.com)

Copyright © 2007 Pearson Education, Inc.

This product is offered as an Adobe Reader<sup>TM</sup> PDF file and does not include digital rights management (DRM) software. While you can copy this material to your computer, you are not allowed to share this file with others.

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc.  
Rights and Contracts Department  
75 Arlington St., Suite 300  
Boston, MA 02116  
Fax: (617) 848-7047

ISBN-13: 978-0-321-50924-6

ISBN-10: 0-321-50924-2

First release, May 2007

### 4.3 A Note on Route Order

to the hash in the earlier `link_to` example, you'll end up with this as your URL:

```
http://localhost:3000/auctions/3/1?some_other_thing=blah
```

## 4.3 A Note on Route Order

Routes are consulted, both for recognition and for generation, in the order they are defined in `routes.rb`. The search for a match ends when the first match is found.

That means you have to watch out for false positives.

For example, let's say you have these two routes in your `routes.rb`:

```
map.connect 'users/help', :controller => "users"
map.connect ':controller/help', :controller => "main"
```

The logic here is that if someone connects to `/users/help`, there's a `users/help` action to help her. But if she connects to `/any_other_controller/help`, she gets the `help` action of the main controller.

Now, consider what would happen if you reversed the order of these two routes:

```
map.connect ':controller/help', :controller => "main"
map.connect 'users/help', :controller => "users"
```

If someone connects to `/users/help`, that first route is going to match—because the more specific case, handling “users” differently, is defined later in the file.

## 4.4 Using Regular Expressions in Routes

It's very similar to other kinds of matching operations, such as case statements:

```
case string
when /. /
  puts "Matched any character!"
when /x/
  puts "Matched 'x'!"
end
```

The second `when` will never be reached, because the first one will match `'x'`. You always want to go *from* the specific or special cases, *to* the general case:

```
case string
when /x/
  puts "Matched 'x'!"
when /. /
  puts "Matched any character!"
end
```

These case examples use regular expressions—`/x/` and so forth—to embody patterns against which a string can be tested for a match. Regular expressions actually play a role in the routing syntax too.

## 4.4 Using Regular Expressions in Routes

Sometimes you want not only to recognize a route, but also to recognize it at a finer-grained level than just what components or fields it has. You can do this through the use of regular expressions.<sup>1</sup>

---

<sup>1</sup> For more on regular expressions in Ruby, see *The Ruby Way* by Hal Fulton, part of this series.

## SECTION 4

### 4.5 Default Parameters and the `url_for` Method

For example, you could route all "show" requests so that they went to an error action if their `id` fields were non-numerical. You'd do this by creating two routes: one that handled numerical `ids`, and a fall-through route that handled the rest:

```
map.connect ':controller/show/:id',  
  :id => /\d+/, :action => "show"
```

```
map.connect ':controller/show/:id',  
  :action => "alt_show"
```

You can also wrap your regular expression-based constraints in a special hash parameter, `:requirements`:

```
map.connect ':controller/show/:id',  
  :action => "show", :requirements => { :id => /\d+/ }
```

Regular expressions in routes can be useful, especially when you have routes that differ from each other *only* with respect to the patterns of their components. But they're not a full-blown substitute for data-integrity checking. A URL that matches a route with regular expressions is like a job candidate who's passed a first interview. You still want to make sure the values you're dealing with are usable and appropriate for your application's domain.

## 4.5 Default Parameters and the `url_for` Method

The URL generation techniques you're likely to use—`link_to`, `redirect_to`, and `friends`—are actually wrappers around a lower-level method called `url_for`. It's worth looking at `url_for` on its own terms, because you'll learn something about how Rails generates URLs. (And you might want to call `url_for` on its own at some point.)

4.5 Default Parameters and the `url_for` Method

The `url_for` method's job is to generate a URL from your specifications, married to the rules in the route it finds to be a match. This method abhors a vacuum: In generating a URL, it likes to fill in as many fields as possible. To that end, if it can't find a value for a particular field from the information in the hash you've given it, it looks for a value in the current request parameters.

In other words, in the face of missing values for URL segments, `url_for` defaults to the current values for `:controller`, `:action`, and, where appropriate, other parameters required by the route.

This means that you can economize on repeating information, if you're staying inside the same controller. For example, inside a *show* view for a template, you could create a link to the *edit* action like this:

```
<%= link_to "Edit this auction", :action => "edit", :id => @auction.id %>
```

Assuming that this view is only ever rendered by actions in the auctions controller, the current controller at the time of the rendering will always be auctions. Therefore, because there's no `:controller` specified in the URL hash, the generator will fall back on auctions, and will come up with (for auction 5) this:

```
<a href="http://localhost:3000/auctions/edit/5">Edit this auction</a>
```

based on the default route (`:controller/:action/:id`).

The same is true of the action. If you don't supply an `:action` key, the current action will be interpolated. Keep in mind, though, that it's pretty common for one action to render a template that belongs to another. So, it's less likely that you'll want to let the URL generator fall back on the current action than on the current controller.

### 4.5.1 What Happened to `:id`?

Note that in that last example, we defaulted on `:controller` but we had to provide a value for `:id`.

That's because of how the use of defaults works in `url_for`. What happens is that the route generator marches along the template segments, from left to right—in the default case, like this:

```
:controller/:action/:id
```

and fills in the fields based on the parameters from the current request until it hits one where you've provided a value:

```
:controller/:action/:id  
  default!      provided!
```

When it hits one that you've provided, it checks to see whether what you've provided is the default it would have used anyway. Because we're using a *show* template as our example, and the link is to an edit action, we're not using the default value for `:action`.

Once it hits a nondefault value, `url_for` stops using defaults entirely. It figures that once you've branched away from the defaults, you want to keep branching. So the nondefault field and *all fields to its right* cease to fall back on the current request for default values.

That's why there's a specific value for `:id`, even though it may well be the same as the `params[:id]` value left over from the previous request.

Pop quiz: What would happen if you switched the default route to this:

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

and then did this in the `show.rhtml` template:

```
<%= link_to "Edit this auction", :action => "edit" %>
```

Answer: Because `:id` is no longer to the right of `:action`, but to its left, the generator would happily fill in both `:controller` and `:id` from their values in the current request. It would then use `"edit"` in the `:action` field because we've hard-coded that. There's nothing to the right of `:action`, so at that point everything's done.

So, if this is the show view for auction 5, we'd get the same hyperlink as before—*almost*. Because the default route changed, so would the ordering of the URL fields:

```
<a href="http://localhost:3000/auctions/5/edit">Edit this auction</a>
```

There's no advantage to actually doing this. The point, rather, is to get a feel for how the routing system works by seeing what happens when you tweak it.

## 4.6 Using Literal URLs

If you want to, you can hard-code your paths and URLs as string arguments to `link_to`, `redirect_to`, and `friends`. For example, instead of this:

```
<%= link_to "Help", :controller => "main", :action => "help" %>
```

you can write this:

```
<%= link_to "Help", "/main/help" %>
```