Extracted from:

Agile Web Development with Rails

Third Edition

The Pragmatic Programmers

Covers Rails 2

Agile Web Development with Rails

Third Edition

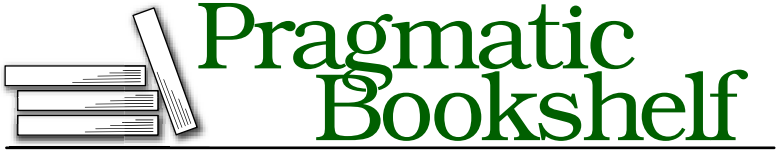
Sam Ruby

Dave Thomas

David Heinemeier Hansson

with Leon Breedt, Mike Clark, Justin Gehtland, James Duncan Davidson, and Andreas Schwarz

The Facets of Ruby Series



Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at

http://www.pragprog.com

Printed in the United States of America.

ISBN-10: 1-934356-16-6

ISBN-13: 978-1-9343561-6-6

Printed on acid-free paper.

P2.0 printing, April 2009

Version: 2009-4-28

Chapter 21

Action Controller: Routing and URLs

Action Pack lies at the heart of Rails applications. It consists of two Ruby modules, ActionController and ActionView. Together, they provide support for processing incoming requests and generating outgoing responses. In this chapter and the next, we'll look at ActionController and how it works within Rails. In the chapter that follows these two, we'll take on ActionView.

When we looked at Active Record, we treated it as a freestanding library; you can use Active Record as a part of a nonweb Ruby application. Action Pack is different. Although it is possible to use it directly as a framework, you probably won't. Instead, you'll take advantage of the tight integration offered by Rails. Components such as Action Controller, Action View, and Active Record handle the processing of requests, and the Rails environment knits them together into a coherent (and easy-to-use) whole. For that reason, we'll describe Action Controller in the context of Rails. Let's start by looking at how Rails applications handle requests. We'll then dive down into the details of routing and URL handling. Chapter 22, *Action Controller and Rails*, then looks at how you write code in a controller.

21.1 The Basics

At its simplest, a web application accepts an incoming request from a browser, processes it, and sends a response.

The first question that springs to mind is, how does the application know what to do with the incoming request? A shopping cart application will receive requests to display a catalog, add items to a cart, check out, and so on. How does it route these requests to the appropriate code?

It turns out that Rails provides two ways to define how to route a request: a comprehensive way that you will use when you need to and a convenient way that you will generally use whenever you can.

The comprehensive way lets you define a direct mapping of URLs to actions based on pattern matching, requirements, and conditions. The convenient way lets you define routes based on resources, such as the models that you define. And because the convenient way is built on the comprehensive way, you can freely mix and match the two approaches.

In both cases, Rails encodes information in the request URL and uses a subsystem called *routing* to determine what should be done with that request. The actual process is very flexible, but at the end of it Rails has determined the name of the *controller* that handles this particular request, along with a list of any other request parameters. In the process, either one of these additional parameters or the HTTP method itself is used to identify the *action* to be invoked in the target controller.

For example, an incoming request to our shopping cart application might look like http://my.shop.com/store/show_product/123. This is interpreted by the application as a request to invoke the show_product method in class StoreController, requesting that it display details of the product with the id 123.

You don't have to use the controller/action/id style of URL. For resources, most URLs are simply controller/id, where the action is supplied by HTTP. And a blogging application could be configured so that article dates could be encoded in the request URLs. Access it at http://my.blog.com/blog/2005/07/04, for example, and it might invoke the display action of the Articles controller to show the articles for July 4, 2005. We'll describe just how this kind of magic mapping occurs shortly.

Once the controller is identified, a new instance is created, and its process method is called, passing in the request details and a response object. The controller then calls a method with the same name as the action (or a method called method_missing, if a method named for the action can't be found). This is the dispatching mechanism we first saw in Figure 4.3, on page 51. The action method orchestrates the processing of the request. If the action method returns without explicitly rendering something, the controller attempts to render a template named after the action. If the controller can't find an action method to call, it immediately tries to render the template—you don't need an action method in order to display a template.

21.2 Routing Requests

So far in this book we haven't worried about how Rails maps a request such as store/add_to_cart/123 to a particular controller and action. Let's dig into that now. We will start with the comprehensive approach because that will provide the foundation for understanding the more convenient approach based on resources.

The rails command generates the initial set of files for an application. One of these files is config/routes.rb. It contains the routing information for that application. If you look at the default contents of the file, ignoring comments, you'll see the following:

```
ActionController::Routing::Routes.draw do |map|
  map.connect ':controller/:action/:id'
  map.connect ':controller/:action/:id.:format'
end
```

The Routing component draws a map that lets Rails connect external URLs to the internals of the application. Each map.connect declaration specifies a route connecting external URLs and internal program code. Let's look at the first map.connect line. The string ':controller/:action/:id' acts as a pattern, matching against the path portion of the request URL. In this case, the pattern will match any URL containing three components in the path. (This isn't actually true, but we'll clear that up in a minute.) The first component will be assigned to the parameter :controller, the second to :action, and the third to :id. Feed this pattern the URL with the path store/add_to_cart/123, and you'll end up with these parameters:

```
@params = {  :controller => 'store',
             :action     => 'add_to_cart',
             :id         => 123 }
```

Based on this, Rails will invoke the add_to_cart method in the store controller. The :id parameter will have a value of 123.

Playing with Routes

Initially, routes can be somewhat intimidating. As you start to define more and more complex routes, you'll start to encounter a problem—how do you know that your routes work the way you expect?

Clearly, one approach is to fire up your application and enter URLs into a browser. However, we can do better than that. For ad hoc experimentation with routes, we can use the script/console command. (For more formal verification we can write unit tests, as we'll see starting on page 460.) We're going to look at how to play with routes now, because it'll come in handy when we look at all the features of routing later.

The routing definition for an application is loaded into a RouteSet object in the ActionController::Routing module. Somewhat confusingly, we can access this via the Routes constant (which turns out not to be that constant). In particular, we can get to the routing definition using script/console, which lets us play with them interactively.

To save ourselves some typing, we'll assign a reference to this RouteSet object to a new local variable, rs:

```
depot> ruby script/console
>> rs = ActionController::Routing::Routes
=> #<ActionController::Routing::RouteSet:0x13cfb70....
```

Ignore the many lines of output that will be displayed—the RouteSet is a fairly complex object. Fortunately, it has a simple (and powerful) interface. Let's start by examining the routes that are defined for our application. We do that by asking the route set to convert each of its routes to a string, which formats them nicely. By using puts to display the result, we'll have each route displayed on a separate line:

```
>> puts rs.routes
ANY    /:controller/:action/:id/               {}
ANY    /:controller/:action/:id.:format/       {}
=> nil
```

The lines starting with ANY show the two default routes that come with any new Rails application (including Depot, which has considerably more routes defined). The final line, => nil, is the script/console command showing the return value of the puts method.

Each displayed route has three components. The first tells routing what HTTP verb this routing applies to. The default, ANY, means that the routing will be applied regardless of the verb. We'll see later how we can create different routing for GET, POST, HEAD, and so on.

The next element is the pattern matched by the route. It corresponds to the string we passed to the map.connect call in our routes.rb file.

The last element shows the optional parameters that modify the behavior of the route. We'll be talking about these parameters shortly.

Use the recognize_path method to see how routing would parse a particular incoming path. The following examples are based on the Depot application:

```
>> rs.recognize_path "/store"
=> {:action=>"index", :controller=>"store"}

>> rs.recognize_path "/store/add_to_cart/1"
=> {:action=>"add_to_cart", :controller=>"store", :id=>"1"}

>> rs.recognize_path "/store/add_to_cart/1.xml"
=> {:action=>"add_to_cart", :controller=>"store", :format=>"xml", :id=>"1"}
```

You can also use the generate method to see what URL routing will create for a particular set of parameters. This is like using the url_for method inside your application.[1]

1. It's worth stressing this point. Inside an application, you'll use methods such as url_for and

```
>> rs.generate :controller => :store
=> "/store"
>> rs.generate :controller => :store, :id => 123
=> "/store/index/123"
```

All of these examples used our application's routing and relied on our application having implemented all the controllers referenced in the request path—routing checks that the controller is valid and so won't parse a request for a controller it can't find. For example, our Depot application doesn't have a coupon controller. If we try to parse an incoming route that uses this controller, the path won't be recognized:

```
>> rs.recognize_path "/coupon/show/1"
ActionController::RoutingError: no route found to match
    "/coupon/show/1" with {}
```

We can tell routing to pretend that our application contains controllers that have not yet been written with the use_controllers! method:

```
>> ActionController::Routing.use_controllers! ["store", "admin", "coupon"]
=> ["store", "admin", "coupon"]
```

However, for this change to take effect, we need to reload the definition of the routes:

```
>> load "config/routes.rb"
=> true
>> rs.recognize_path "/coupon/show/1"
=> {:action=>"show", :controller=>"coupon", :id=>"1"}
```

We can use this trick to test routing schemes that are not yet part of our application: create a new Ruby source file containing the Routes.draw block that would normally be in the routes.rb configuration file, and load this new file using load.

Defining Routes with map.connect

The patterns accepted by map.connect are simple but powerful:

- Components are separated by forward slash characters and periods. Each component in the pattern matches one or more components in the URL. Components in the pattern match in order against the URL.
- A pattern component of the form *:name* sets the parameter *name* to whatever value is in the corresponding position in the URL.
- A pattern component of the form **name* accepts all remaining components in the incoming URL. The parameter *name* will reference an array containing their values. Because it swallows all remaining components of the URL, **name* must appear at the end of the pattern.

link_to to generate route-based URLs. The only reason we're using the generate method here is that it works in the context of a console session.

- Anything else as a pattern component matches exactly itself in the corresponding position in the URL. For example, a routing pattern containing store/:controller/buy/:id would map if the URL contains the text store at the front and the text buy as the third component of the path.

map.connect accepts additional parameters.

:defaults => { :name => "value", ...}

Sets default values for the named parameters in the pattern. Trailing components in the pattern that have default values can be omitted in the incoming URL, and their default values will be used when setting the parameters. Parameters with a default of nil will not be added to the params hash if they do not appear in the URL. If you don't specify otherwise, routing will automatically supply the following defaults:

```
defaults => { :action => "index", :id => nil }
```

This explains the parsing of the default route, specified in routes.rb as follows:

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

Because the action defaults to "index" and the id may be omitted (because it defaults to nil), routing recognizes the following styles of incoming URL for the default Rails application:

```
>> rs.recognize_path "/store"
=> {:action=>"index", :controller=>"store"}
>> rs.recognize_path "/store/show"
=> {:action=>"show", :controller=>"store"}
>> rs.recognize_path "/store/show/1"
=> {:action=>"show", :controller=>"store", :id=>"1"}
```

:requirements => { :name =>*/regexp/*, ...}

Specifies that the given components, if present in the URL, must each match the specified regular expressions in order for the map as a whole to match. In other words, if any component does not match, this map will not be used.

:conditions => { :name =>*/regexp/*or*string*, ...}

Introduced in Rails 1.2, :conditions allows you to specify that routes are matched only in certain circumstances. The set of conditions that may be tested may be extended by plug-ins—out of the box, routing supports a single condition. This allows you to write routes that are conditional on the HTTP verb used to submit the incoming request.

In the following example, Rails will invoke the display_checkout_form action when it receives a GET request to /store/checkout, but it will call the action save_checkout_form if it sees a POST request to that same URL:

Download e1/routing/config/routes_with_conditions.rb

```
ActionController::Routing::Routes.draw do |map|
  map.connect 'store/checkout',
        :conditions => { :method => :get },
        :controller => "store",
        :action     => "display_checkout_form"

  map.connect 'store/checkout',
        :conditions => { :method => :post },
        :controller => "store",
        :action     => "save_checkout_form"
end
```

:name => value

Sets a default value for the component *:name*. Unlike the values set using :defaults, the name need not appear in the pattern itself. This allows you to add arbitrary parameter values to incoming requests. The value will typically be a string or nil.

:name => /regexp/

Equivalent to using :requirements to set a constraint on the value of *:name*.

There's one more rule: routing tries to match an incoming URL against each rule in routes.rb in turn. The first match that succeeds is used. If no match succeeds, an error is raised.

Now let's look at a more complex example. In your blog application, you'd like all URLs to start with the word blog. If no additional parameters are given, you'll display an index page. If the URL looks like blog/show/*nnn*, you'll display article *nnn*. If the URL contains a date (which may be year, year/month, or year/month/day), you'll display articles for that date. Otherwise, the URL will contain a controller and action name, allowing you to edit articles and otherwise administer the blog. Finally, if you receive an unrecognized URL pattern, you'll handle that with a special action.

The routing for this contains a line for each individual case:

Download e1/routing/config/routes_for_blog.rb

```
ActionController::Routing::Routes.draw do |map|

  # Straight 'http://my.app/blog/' displays the index
  map.connect "blog/",
              :controller => "blog",
              :action => "index"
```

```
  # Return articles for a year, year/month, or year/month/day
  map.connect "blog/:year/:month/:day",
              :controller => "blog",
              :action => "show_date",
              :requirements => { :year => /(19|20)\d\d/,
                                 :month => /[01]?\d/,
                                 :day => /[0-3]?\d/},
              :day => nil,
              :month => nil

  # Show an article identified by an id
  map.connect "blog/show/:id",
              :controller => "blog",
              :action => "show",
              :id => /\d+/

  # Regular Rails routing for admin stuff
  map.connect "blog/:controller/:action/:id"

  # Catchall so we can gracefully handle badly formed requests
  map.connect "*anything",
              :controller => "blog",
              :action => "unknown_request"
end
```

Note two things in this code. First, we constrained the date-matching rule to look for reasonable-looking year, month, and day values. Without this, the rule would also match regular controller/action/id URLs. Second, notice how we put the catchall rule ("*anything") at the end of the list. Because this rule matches any request, putting it earlier would stop subsequent rules from being examined.

We can see how these rules handle some request URLs:

```
>> ActionController::Routing.use_controllers! [ "article", "blog" ]
=> ["article", "blog"]

>> load "config/routes_for_blog.rb"
=> []

>> rs.recognize_path "/blog"
=> {:controller=>"blog", :action=>"index"}

>> rs.recognize_path "/blog/show/123"
=> {:controller=>"blog", :action=>"show", :id=>"123"}

>> rs.recognize_path "/blog/2004"
=> {:year=>"2004", :controller=>"blog", :action=>"show_date"}

>> rs.recognize_path "/blog/2004/12"
=> {:month=>"12", :year=>"2004", :controller=>"blog", :action=>"show_date"}
```

```
>> rs.recognize_path "/blog/2004/12/25"
=> {:month=>"12", :year=>"2004", :controller=>"blog", :day=>"25",
    :action=>"show_date"}

>> rs.recognize_path "/blog/article/edit/123"
=> {:controller=>"article", :action=>"edit", :id=>"123"}

>> rs.recognize_path "/blog/article/show_stats"
=> {:controller=>"article", :action=>"show_stats"}

>> rs.recognize_path "/blog/wibble"
=> {:controller=>"blog", :anything=>["blog", "wibble"], :action=>"unknown_request"}

>> rs.recognize_path "/junk"
=> {:controller=>"blog", :anything=>["junk"], :action=>"unknown_request"}
```

We're not quite done with specifying routes yet, but before we look at creating named routes, let's first see the other side of the coin—generating a URL from within our application.

URL Generation

Routing takes an incoming URL and decodes it into a set of parameters that are used by Rails to dispatch to the appropriate controller and action (potentially setting additional parameters along the way). But that's only half the story. Our application also needs to create URLs that refer back to itself. Every time it displays a form, for example, that form needs to link back to a controller and action. But the application code doesn't necessarily know the format of the URLs that encode this information; all it sees are the parameters it receives once routing has done its work.

We could hard-code all the URLs into the application, but sprinkling knowledge about the format of requests in multiple places would make our code more brittle. This is a violation of the DRY principle;[2] change the application's location or the format of URLs, and we'd have to change all those strings.

Fortunately, we don't have to worry about this, because Rails also abstracts the generation of URLs using the url_for method (and a number of higher-level friends that use it). To illustrate this, let's go back to a simple mapping:

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

The url_for method generates URLs by applying its parameters to a mapping. It works in controllers and in views. Let's try it:

```
@link = url_for(:controller => "store", :action => "display", :id => 123)
```

2. DRY stands for *don't repeat yourself*, an acronym coined in *The Pragmatic Programmer* [HT00].

This code will set @link to something like this:

```
http://pragprog.com/store/display/123
```

The url_for method took our parameters and mapped them into a request that is compatible with our own routing. If the user selects a link that has this URL, it will invoke the expected action in our application.

The rewriting behind url_for is fairly clever. It knows about default parameters and generates the minimal URL that will do what you want. And, as you might have suspected, we can play with it from within script/console. We can't call url_for directly, because it is available only inside controllers and views. We can, however, do the next best thing and call the generate method inside routings. Again, we'll use the route set that we used previously. Let's look at some examples:

```
# No action or id, the rewrite uses the defaults
>> rs.generate :controller => "store"
=> "/store"

# If the action is missing, the rewrite inserts the default (index) in the URL
>> rs.generate :controller => "store", :id => 123
=> "/store/index/123"

# The id is optional
>> rs.generate :controller => "store", :action => :list
=> "/store/list"

# A complete request
>> rs.generate :controller => "store", :action => :list, :id => 123
=> "/store/list/123"

# Additional parameters are added to the end of the URL
>> rs.generate :controller => "store", :action => :list, :id => 123,
?>             :extra => "wibble"
=> "/store/list/123?extra=wibble"
```

The defaulting mechanism uses values from the current request if it can. This is most commonly used to fill in the current controller's name if the :controller parameter is omitted. We can demonstrate this inside script/console by using the optional second parameter to generate. This parameter gives the options that were parsed from the currently active request. So, if the current request is to /store/index and we generate a new URL giving just an action of show, we'll still see the store part included in the URL's path:

```
>> rs.generate({:action => "show"},
?>             {:controller => "store", :action => "index"})
=> "/store/show"
```

To make this more concrete, we can see what would happen if we used url_for in (say) a view in these circumstances. Note that the url_for method is normally

available only to controllers, but script/console provides us with an app object with a similar method, the key difference being that the method on the app object doesn't have a default request. This means that we can't rely on defaults being provided for :controller and :action:

```
>> app.url_for :controller => :store, :action => :display, :id => 123
=> http://example.com/store/status
```

URL generation works for more complex routings as well. For example, the routing for our blog includes the following mappings:

Download e1/routing/config/routes_for_blog.rb

```
# Return articles for a year, year/month, or year/month/day
map.connect "blog/:year/:month/:day",
            :controller => "blog",
            :action => "show_date",
            :requirements => { :year => /(19|20)\d\d/,
                               :month => /[01]?\d/,
                               :day => /[0-3]?\d/},
            :day => nil,
            :month => nil

# Show an article identified by an id
map.connect "blog/show/:id",
            :controller => "blog",
            :action => "show",
            :id => /\d+/

# Regular Rails routing for admin stuff
map.connect "blog/:controller/:action/:id"
```

Imagine the incoming request was http://pragprog.com/blog/2006/07/28. This will have been mapped to the show_date action of the Blog controller by the first rule:

```
>> ActionController::Routing.use_controllers! [ "blog" ]
=> ["blog"]
>> load "config/routes_for_blog.rb"
=> true
>> last_request = rs.recognize_path "/blog/2006/07/28"
=> {:month=>"07", :year=>"2006", :controller=>"blog", :day=>"28", :action=>"show_date"}
```

Let's see what various url_for calls will generate in these circumstances.

If we ask for a URL for a different day, the mapping call will take the values from the incoming request as defaults, changing just the day parameter:

```
>> rs.generate({:day => 25}, last_request)
=> "/blog/2006/07/25"
```

Now let's see what happens if we instead give it just a year:

```
>> rs.generate({:year => 2005}, last_request)
=> "/blog/2005"
```

That's pretty smart. The mapping code assumes that URLs represent a hierarchy of values.[3] Once we change something away from the default at one level in that hierarchy, it stops supplying defaults for the lower levels. This is reasonable: the lower-level parameters really make sense only in the context of the higher-level ones, so changing away from the default invalidates the lower-level ones. By overriding the year in this example, we implicitly tell the mapping code that we don't need a month and day.

Note also that the mapping code chose the first rule that could reasonably be used to render the URL. Let's see what happens if we give it values that can't be matched by the first, date-based rule:

```
>> rs.generate({:action => "edit", :id => 123}, last_request)
=> "/blog/blog/edit/123"
```

Here the first blog is the fixed text, the second blog is the name of the controller, and edit is the action name—the mapping code applied the third rule. If we'd specified an action of show, it would use the second mapping:

```
>> rs.generate({:action => "show", :id => 123}, last_request)
=> "/blog/show/123"
```

Most of the time the mapping code does just what you want. However, it is sometimes too smart. Say you wanted to generate the URL to view the blog entries for 2006. You could write this:

```
>> rs.generate({:year => 2006}, last_request)
```

You might be surprised when the mapping code spat out a URL that included the month and day as well:

```
=> "/blog/2006/07/28"
```

The year value you supplied was the same as that in the current request. Because this parameter hadn't changed, the mapping carried on using default values for the month and day to complete the rest of the URL. To get around this, set the month parameter to nil:

```
>> rs.generate({:year => 2006, :month => nil}, last_request)
=> "/blog/2006"
```

In general, if you want to generate a partial URL, it's a good idea to set the first of the unused parameters to nil; doing so prevents parameters from the incoming request leaking into the outgoing URL.

Sometimes you want to do the opposite, changing the value of a parameter higher in the hierarchy and forcing the routing code to continue to use values at lower levels. In our example, this would be like specifying a different year and having it add the existing default month and day values after it in the

3. This is natural on the Web, where static content is stored within folders (directories), which themselves may be within folders, and so on.

URL. To do this, we can fake out the routing code—we use the :overwrite_params option to tell url_for that the original request parameters contained the new year that we want to use. Because it thinks that the year hasn't changed, it continues to use the rest of the defaults. (Note that this option doesn't work down within the routing API, so we can't demonstrate it directly in script/console.)

```
url_for(:year => "2002")
=> http://example.com/blog/2002

url_for(:overwrite_params => {:year => "2002"})
=> http://example.com/blog/2002/4/15
```

One last gotcha. Say a mapping has a requirement such as this:

```
map.connect "blog/:year/:month/:day",
            :controller   => "blog",
            :action       => "show_date",
            :requirements => { :year  => /(19|20)\d\d/,
                               :month => /[01]\d/,
                               :day   => /[0-3]\d/},
```

Note that the :day parameter is required to match /[0-3]\d/; it must be two digits long. This means that if you pass in a Fixnum value less than 10 when creating a URL, this rule will not be used.

```
url_for(:year => 2005, :month => 12, :day => 8)
```

Because the number 8 converts to the string "8" and that string isn't two digits long, the mapping won't fire. The fix is either to relax the rule (making the leading zero optional in the requirement with [0-3]?\d) or to make sure you pass in two-digit numbers:

```
url_for(:year=>year, :month=>sprintf("%02d", month), :day=>sprintf("%02d", day))
```

The url_for Method

Now that we've looked at how mappings are used to generate URLs, we can look at the url_for method in all its glory:

url_for

Create a URL that references this application

```
url_for(option => value, ...)
```

Creates a URL that references a controller in this application. The *options* hash supplies parameter names and their values that are used to fill in the URL (based on a mapping). The parameter values must match any constraints imposed by the mapping that is used. Certain parameter names, listed in the *Options:* section that follows, are reserved and are used to fill in the nonpath part of the URL. If you use an Active Record model object as a value in url_for (or any related method), that object's database id will be used.

The two redirect calls in the following code fragment have an identical effect:

```
user = User.find_by_name("dave thomas")
redirect_to(:action => 'delete', :id => user.id)

# can be written as
redirect_to(:action => 'delete', :id => user)
```

url_for also accepts a single string or symbol as a parameter. Rails uses this internally.

You can override the default values for the parameters in the following table by implementing the method default_url_options in your controller. This should return a hash of parameters that could be passed to url_for.

Options:

:anchor	*string*	An anchor name to be appended to the URL. Rails automatically prepends the # character.
:host	*string*	Sets the host name and port in the URL. Use a string such as store.pragprog.com or helper.pragprog.com:8080. Defaults to the host in the incoming request.
:only_path	*boolean*	Only the path component of the URL is generated; the protocol, host name, and port are omitted.
:protocol	*string*	Sets the protocol part of the URL. Use a string such as "https://". Defaults to the protocol of the incoming request.
:overwrite_params	*hash*	The options in hash are used to create the URL, but no default values are taken from the current request.
:skip_relative_url_root	*boolean*	If true, the relative URL root is not prepended to the generated URL. See Section 21.2, *Rooted URLs*, on page 443 for more details.
:trailing_slash	*boolean*	Appends a slash to the generated URL. Use :trailing_slash with caution if you also use page or action caching (described starting on page 498). The extra slash reportedly confuses the caching algorithm.
:port	*integer*	Sets the port to connect to. Defaults based on the protocol.
:user	*string*	Sets the user. Used for inline authentication. Used only if :password is also specified.
:password	*string*	Sets the password. Used for inline authentication. Used only if :user is also specified.

Named Routes

So far we've been using anonymous routes, created using map.connect in the routes.rb file. Often this is enough; Rails does a good job of picking the URL to generate given the parameters we pass to url_for and its friends. However, we can make our application easier to understand by giving the routes names. This doesn't change the parsing of incoming URLs, but it lets us be explicit about generating URLs using specific routes in our code.

You create a named route simply by using a name other than connect in the routing definition. The name you use becomes the name of that particular route.

For example, we might recode our blog routing as follows:

Download e1/routing/config/routes_with_names.rb

```
ActionController::Routing::Routes.draw do |map|

  # Straight 'http://my.app/blog/' displays the index
  map.index "blog/",
            :controller => "blog",
            :action => "index"

  # Return articles for a year, year/month, or year/month/day
  map.date "blog/:year/:month/:day",
           :controller => "blog",
           :action => "show_date",
           :requirements => { :year => /(19|20)\d\d/,
                              :month => /[01]?\d/,
                              :day => /[0-3]?\d/},
           :day => nil,
           :month => nil

  # Show an article identified by an id
  map.show_article "blog/show/:id",
                   :controller => "blog",
                   :action => "show",
                   :id => /\d+/

  # Regular Rails routing for admin stuff
  map.blog_admin "blog/:controller/:action/:id"

  # Catchall so we can gracefully handle badly formed requests
  map.catch_all "*anything",
                :controller => "blog",
                :action => "unknown_request"
end
```

Here we've named the route that displays the index as index, the route that accepts dates is called date, and so on. We can use these to generate URLs by appending _url to their names and using them in the same way we'd otherwise use url_for. Thus, to generate the URL for the blog's index, we could use this:

```
@link = index_url
```

This will construct a URL using the first routing, resulting in the following:

```
http://pragprog.com/blog/
```

You can pass additional parameters as a hash to these named routes. The parameters will be added into the defaults for the particular route. This is illustrated by the following examples:

```
index_url
  #=>   http://pragprog.com/blog

date_url(:year => 2005)
  #=>   http://pragprog.com/blog/2005
```

```
date_url(:year => 2003, :month => 2)
  #=>   http://pragprog.com/blog/2003/2

show_article_url(:id => 123)
  #=>   http://pragprog.com/blog/show/123
```

You can use an *xxx*_url method wherever Rails expects URL parameters. Thus, you could redirect to the index page with the following code:

```
redirect_to(index_url)
```

In a view template, you could create a hyperlink to the index using this:

```
<%= link_to("Index", index_url) %>
```

As well as the *xxx*_url methods, Rails also creates *xxx*_path forms. These construct just the path portion of the URL (ignoring the protocol, host, and port).

Finally, if the only parameters to a named URL generation method are used to fill in values for named fields in the URL, you can pass them as regular parameters, rather than as a hash. For example, our sample routes.rb file defined a named URL for blog administration:

Download e1/routing/config/routes_with_names.rb

```
map.blog_admin "blog/:controller/:action/:id"
```

We've already seen how we can link to the *list users* action with a named URL generator:

```
blog_admin_url :controller => 'users', :action => 'list'
```

Because we're using options only to give the named parameters values, we could also have used this:

```
blog_admin_url  'users', 'list'
```

Perhaps surprisingly, this form is less efficient than passing a hash of values:

Controller Naming

Back on page 271 we said that controllers could be grouped into modules and that incoming URLs identified these controllers using a path-like convention. An incoming URL of http://my.app/admin/book/edit/123 would invoke the edit action of BookController in the Admin module.

This mapping also affects URL generation:

- If you do not pass a :controller parameter to url_for, it uses the current controller.
- If you pass a controller name starting with /, then that name is absolute.
- All other controller names are relative to the module of the controller issuing the request.

To illustrate this, let's assume an incoming request of this format:

```
http://my.app/admin/book/edit/123
```

```
url_for(:action => "edit", :id => 123)
   #=> http://my.app/admin/book/edit/123

url_for(:controller => "catalog", :action => "show", :id => 123)
   #=> http://my.app/admin/catalog/show/123

url_for(:controller => "/store", :action => "purchase", :id => 123)
   #=> http://my.app/store/purchase/123

url_for(:controller => "/archive/book", :action => "record", :id => 123)
   #=> http://my.app/archive/book/record/123
```

Rooted URLs

Sometimes you want to run multiple copies of the same application. Perhaps you're running a service bureau and have multiple customers. Or maybe you want to run both staging and production versions of your application.

If possible, the easiest way of doing this is to run multiple (sub)domains with an application instance in each. However, if this is not possible, you can also use a prefix in your URL path to distinguish your application instances. For example, you might run multiple users' blogs on URLs such as this:

```
http://megablogworld.com/dave/blog
http://megablogworld.com/joe/blog
http://megablogworld.com/sue/blog
```

In these cases, the prefixes dave, joe, and sue identify the application instance. The application's routing starts after this. You can tell Rails to ignore this part of the path on URLs it receives, and to prepend it on URLs it generates, by setting the environment variable RAILS_RELATIVE_URL_ROOT. If your Rails application is running on Apache, this feature is automatically enabled.

21.3 Resource-Based Routing

Rails routes support the mapping between URLs and actions based on the contents of the URL and on the HTTP method used to invoke the request. We've seen how to do this on a URL-by-URL basis using anonymous or named routes. Rails also supports a higher-level way of creating groups of related routes. To understand the motivation for this, we need to take a little diversion into the world of Representational State Transfer.

REST: Representational State Transfer

REST is a way of thinking about the architecture of distributed hypermedia systems. This is relevant to us because many web applications can be categorized this way.

The ideas behind REST were formalized in Chapter 5 of Roy Fielding's 2000 PhD dissertation.[4] In a REST approach, servers communicate with clients using stateless connections. All the information about the state of the interaction between the two is encoded into the requests and responses between them. Long-term state is kept on the server as a set of identifiable *resources*. Clients access these resources using a well-defined (and severely constrained) set of resource identifiers (URLs in our context). REST distinguishes the content of resources from the presentation of that content. REST is designed to support highly scalable computing while constraining application architectures to be decoupled by nature.

There's a lot of abstract stuff in this description. What does REST mean in practice?

First, the formalities of a RESTful approach mean that network designers know when and where they can cache responses to requests. This enables load to be pushed out through the network, increasing performance and resilience while reducing latency.

Second, the constraints imposed by REST can lead to easier-to-write (and maintain) applications. RESTful applications don't worry about implementing remotely accessible services. Instead, they provide a regular (and simple) interface to a set of resources. Your application implements a way of listing, creating, editing, and deleting each resource, and your clients do the rest.

Let's make this more concrete. In REST, we use a simple set of verbs to operate on a rich set of nouns. If we're using HTTP, the verbs correspond to HTTP methods (GET, PUT, POST, and DELETE, typically). The nouns are the resources in our application. We name those resources using URLs.

A content management system might contain a set of articles. There are implicitly two resources here. First, there are the individual articles. Each constitutes a resource. There's also a second resource: the collection of articles.

To fetch a list of all the articles, we could issue an HTTP GET request against this collection, say on the path /articles. To fetch the contents of an individual resource, we have to identify it. The Rails way would be to give its primary key value (that is, its id). Again we'd issue a GET request, this time against the URL /articles/1. So far, this is all looking quite familiar. But what happens when we want to add an article to our collection?

In non-RESTful applications, we'd probably invent some action with a verb phrase as a name: articles/add_article/1. In the world of REST, we're not supposed to do this. We're supposed to tell resources what to do using a standard set of verbs. To create a new article in our collection using REST, we'd use an

4. http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

HTTP POST request directed at the /articles path, with the post data containing the article to add. Yes, that's the same path we used to get a list of articles. If you issue a GET to it, it responds with a list, and if you do a POST to it, it adds a new article to the collection.

Take this a step further. We've already seen you can retrieve the content of an article—you just issue a GET request against the path /articles/1. To update that article, you'd issue an HTTP PUT request against the same URL. And, to delete it, you could issue an HTTP DELETE request, again using the same URL.

Take this further. Maybe our system also tracks users. Again, we have a set of resources to deal with. REST tells us to use the same set of verbs (GET, POST, PUT, and DELETE) against a similar-looking set of URLS (/users, /user/1, ...).

Now we see some of the power of the constraints imposed by REST. We're already familiar with the way Rails constrains us to structure our applications a certain way. Now the REST philosophy tells us to structure the interface to our applications too. Suddenly our world gets a lot simpler.

REST and Rails

Rails 1.2 added direct support for this type of interface; it adds a kind of macro route facility, called *resources*. Let's create a set of RESTful routes for our articles example:

```
ActionController::Routing::Routes.draw do |map|
  map.resources :articles
end
```

The map.resources line has added seven new routes and four new route helpers to our application. Along the way, it assumed that the application will have a controller named ArticlesController containing seven actions with given names. It's up to us to write that controller.

Before we do, take a look at the routes that were generated for us. We do this by making use of the handy rake routes command:[5]

```
            articles GET    /articles
                             {:controller=>"articles", :action=>"index"}
formatted_articles GET    /articles.:format
                             {:controller=>"articles", :action=>"index"}
                   POST   /articles
                             {:controller=>"articles", :action=>"create"}
                   POST   /articles.:format
                             {:controller=>"articles", :action=>"create"}
```

5. Depending on what release of Rails you are running, you might not see route names that begin with the prefix formatted_. Such routes were found to be rarely used and consumed both CPU and memory resources and therefore are scheduled to be removed in Rails 2.3, to be replaced by an optional :format argument.

```
          new_article GET    /articles/new
                             {:controller=>"articles", :action=>"new"}
 formatted_new_article GET    /articles/new.:format
                             {:controller=>"articles", :action=>"new"}
         edit_article GET    /articles/:id/edit
                             {:controller=>"articles", :action=>"edit"}
formatted_edit_article GET    /articles/:id/edit.:format
                             {:controller=>"articles", :action=>"edit"}
              article GET    /articles/:id
                             {:controller=>"articles", :action=>"show"}
    formatted_article GET    /articles/:id.:format
                             {:controller=>"articles", :action=>"show"}
                      PUT    /articles/:id
                             {:controller=>"articles", :action=>"update"}
                      PUT    /articles/:id.:format
                             {:controller=>"articles", :action=>"update"}
                      DELETE /articles/:id
                             {:controller=>"articles", :action=>"destroy"}
                      DELETE /articles/:id.:format
                             {:controller=>"articles", :action=>"destroy"}
                             /:controller/:action/:id

                             /:controller/:action/:id.:format
```

All the routes defined are spelled out in a columnar format. The lines will generally wrap on your screen; in fact, they had to be broken into two lines per route to fit on this page. The columns are route name, HTTP method, route path, and (on a separate line on this page) route requirements. By now, these should all be familiar to you, because you can define the same mapping using the comprehensive method defined on page 428.

The last two entries represent the default routes that were present before we generated the scaffold for the articles model.

Now let's look at the seven controller actions that these routes reference. Although we created our routes to manage the articles in our application, let's broaden this out in these descriptions and talk about resources—after all, the same seven methods will be required for all resource-based routes:

index
: Returns a list of the resources.

create
: Creates a new resource from the data in the POST request, adding it to the collection.

new
: Constructs a new resource and passes it to the client. This resource will not have been saved on the server. You can think of the new action as creating an empty form for the client to fill in.

show
: Returns the contents of the resource identified by params[:id].

update
: Updates the contents of the resource identified by params[:id] with the data associated with the request.

edit
: Returns the contents of the resource identified by params[:id] in a form suitable for editing.

destroy
: Destroys the resource identified by params[:id].

You can see that these seven actions contain the four basic CRUD operations (create, read, update, and delete). They also contain an action to list resources and two auxiliary actions that return new and existing resources in a form suitable for editing on the client.

If for some reason you don't need or want all seven actions, you can limit the actions produced using :only or :except options on your map.resource:

```
map.resources :comments, :except => [:update, :destroy]
```

Several of the routes are named routes—as described in Section 21.2, *Named Routes*, on page 440—enabling you to use helper functions such as articles_url and edit_article_url(:id=>1). Two routes are defined for each controller action: one with a default format and one with an explicit format. We will cover formats in more detail in Section 21.3, *Selecting a Data Representation*, on page 459.

But first, let's create a simple application to play with this. By now, you know the drill, so we'll take it quickly. We'll create an application called *restful*:

```
work> rails restful
```

So, now we'll start creating our model, controller, views, and so on. We could do this manually, but Rails comes ready-made to produce scaffolding that uses resource-based routing, so let's save ourselves some typing. The generator takes the name of the model (the resource) and optionally a list of field names and types. In our case, the article model has three attributes: a title, a summary, and the content:

```
restful> ruby script/generate scaffold article \
    title:string summary:text content:text
    exists  app/models/
    exists  app/controllers/
    exists  app/helpers/
    create  app/views/articles
    exists  app/views/layouts/
    exists  test/functional/
    exists  test/unit/
    exists  public/stylesheets/
```

```
    create  app/views/articles/index.html.erb
    create  app/views/articles/show.html.erb
    create  app/views/articles/new.html.erb
    create  app/views/articles/edit.html.erb
    create  app/views/layouts/articles.html.erb
    create  public/stylesheets/scaffold.css
    create  app/controllers/articles_controller.rb
    create  test/functional/articles_controller_test.rb
    create  app/helpers/articles_helper.rb
     route  map.resources :articles
dependency  model
    exists    app/models/
    exists    test/unit/
    exists    test/fixtures/
    create    app/models/article.rb
    create    test/unit/article_test.rb
    create    test/fixtures/articles.yml
    create    db/migrate
    create    db/migrate/20080601000001_create_articles.rb
```

Take a look at the line that starts with route in the output of this command. It's telling us that the generator has automatically added the appropriate mapping to our application's routes. Let's take a look at what it did. Look at the top of the file routes.rb in the config/ directory:

Download restful/config/routes.rb

```
ActionController::Routing::Routes.draw do |map|
  map.resources :articles

  # ...

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

The migration file was automatically created and populated with the information we gave the generator:

Download restful/db/migrate/20080601000001_create_articles.rb

```
class CreateArticles < ActiveRecord::Migration
  def self.up
    create_table :articles do |t|
      t.string :title
      t.text :summary
      t.text :content

      t.timestamps
    end
  end

  def self.down
    drop_table :articles
  end
end
```

So, all we have to do is run the migration:

```
restful> rake db:migrate
```

Now we can start the application (by running script/server) and play. The index page lists existing articles; you can add an article, edit an existing article, and so on. But, as you're playing, take a look at the URLs that are generated.

Let's take a look at the controller code:

Download restful/app/controllers/articles_controller.rb

```
class ArticlesController < ApplicationController
  # GET /articles
  # GET /articles.xml
  def index
    @articles = Article.find(:all)

    respond_to do |format|
      format.html # index.html.erb
      format.xml  { render :xml => @articles }
    end
  end

  # GET /articles/1
  # GET /articles/1.xml
  def show
    @article = Article.find(params[:id])

    respond_to do |format|
      format.html # show.html.erb
      format.xml  { render :xml => @article }
    end
  end

  # GET /articles/new
  # GET /articles/new.xml
  def new
    @article = Article.new

    respond_to do |format|
      format.html # new.html.erb
      format.xml  { render :xml => @article }
    end
  end

  # GET /articles/1/edit
  def edit
    @article = Article.find(params[:id])
  end

  # POST /articles
  # POST /articles.xml
  def create
    @article = Article.new(params[:article])
```

```
    respond_to do |format|
      if @article.save
        flash[:notice] = 'Article was successfully created.'
        format.html { redirect_to(@article) }
        format.xml  { render :xml => @article, :status => :created,
                             :location => @article }
      else
        format.html { render :action => "new" }
        format.xml  { render :xml => @article.errors,
                             :status => :unprocessable_entity }
      end
    end
  end

  # PUT /articles/1
  # PUT /articles/1.xml
  def update
    @article = Article.find(params[:id])

    respond_to do |format|
      if @article.update_attributes(params[:article])
        flash[:notice] = 'Article was successfully updated.'
        format.html { redirect_to(@article) }
        format.xml  { head :ok }
      else
        format.html { render :action => "edit" }
        format.xml  { render :xml => @article.errors,
                             :status => :unprocessable_entity }
      end
    end
  end

  # DELETE /articles/1
  # DELETE /articles/1.xml
  def destroy
    @article = Article.find(params[:id])
    @article.destroy

    respond_to do |format|
      format.html { redirect_to(articles_url) }
      format.xml  { head :ok }
    end
  end
end
```

Notice how we have one action for each of the RESTful actions. The comment before each shows the format of the URL that invokes it.

Notice also that many of the actions contain a respond_to block. As we saw back on page 187, Rails uses this to determine the type of content to send in a response. The scaffold generator automatically creates code that will respond appropriately to requests for HTML or XML content. We'll play with that in a little while.

The views created by the generator are fairly straightforward. The only tricky thing is the need to use the correct HTTP method to send requests to the server. For example, the view for the index action looks like this:

Download restful/app/views/articles/index.html.erb

```
<h1>Listing articles</h1>

<table>
  <tr>
    <th>Title</th>
    <th>Summary</th>
    <th>Content</th>
  </tr>

<% for article in @articles %>
  <tr>
    <td><%=h article.title %></td>
    <td><%=h article.summary %></td>
    <td><%=h article.content %></td>
    <td><%= link_to 'Show', article %></td>
    <td><%= link_to 'Edit', edit_article_path(article) %></td>
    <td><%= link_to 'Destroy', article, :confirm => 'Are you sure?',
                                        :method => :delete %></td>
  </tr>
<% end %>
</table>

<br />

<%= link_to 'New article', new_article_path %>
```

The links to the actions that edit an article and add a new article should both use regular GET methods, so a standard link_to works fine.[6] However, the request to destroy an article must issue an HTTP DELETE, so the call includes the :method => :delete option to link_to.[7]

For completeness, here are the other views:

Download restful/app/views/articles/edit.html.erb

```
<h1>Editing article</h1>

<% form_for(@article) do |f| %>
  <%= f.error_messages %>
```

6. Note how we're using named routes as the parameters to these calls. Once you go RESTful, named routes are *de rigueur*.

7. And here the implementation gets messy. Browsers cannot issue HTTP DELETE requests, so Rails fakes it out. If you look at the generated HTML, you'll see that Rails uses JavaScript to generate a dynamic form. The form will post to the action you specify. But it also contains an extra hidden field named _method whose value is delete. When a Rails application receives a _method parameter, it ignores the real HTTP method and pretends the parameter's value (delete in this case) was used.

```
  <p>
    <%= f.label :title %><br />
    <%= f.text_field :title %>
  </p>
  <p>
    <%= f.label :summary %><br />
    <%= f.text_area :summary %>
  </p>
  <p>
    <%= f.label :content %><br />
    <%= f.text_area :content %>
  </p>
  <p>
    <%= f.submit "Update" %>
  </p>
<% end %>

<%= link_to 'Show', @article %> |
<%= link_to 'Back', articles_path %>
```

Download restful/app/views/articles/new.html.erb

```
<h1>New article</h1>

<% form_for(@article) do |f| %>
  <%= f.error_messages %>

  <p>
    <%= f.label :title %><br />
    <%= f.text_field :title %>
  </p>
  <p>
    <%= f.label :summary %><br />
    <%= f.text_area :summary %>
  </p>
  <p>
    <%= f.label :content %><br />
    <%= f.text_area :content %>
  </p>
  <p>
    <%= f.submit "Create" %>
  </p>
<% end %>

<%= link_to 'Back', articles_path %>
```

Download restful/app/views/articles/show.html.erb

```
<p>
  <b>Title:</b>
  <%=h @article.title %>
</p>

<p>
  <b>Summary:</b>
  <%=h @article.summary %>
</p>
```

```
<p>
  <b>Content:</b>
  <%=h @article.content %>
</p>


<%= link_to 'Edit', edit_article_path(@article) %> |
<%= link_to 'Back', articles_path %>
```

Adding Your Own Actions

In an ideal world you'd use a consistent set of actions across all your application's resources, but this isn't always practical. You sometimes need to add special processing to a resource. For example, we may need to create an interface to allow people to fetch just recent articles. To do that with Rails, we use an extension to the map.resources call:

```
ActionController::Routing::Routes.draw do |map|
  map.resources :articles, :collection => { :recent => :get }
end
```

That syntax takes a bit of getting used to. It says "we want to add a new action named recent, invoked via an HTTP GET. It applies to the collection of resources—in this case all the articles."

The :collection option adds the following routing to the standard set added by map.resources:

Method	URL Path	Action	Helper
GET	/articles/recent	recent	recent_articles_url

In fact, we've already seen this technique of appending special actions to a URL using a slash—the edit action uses the same mechanism.

You can also create special actions for individual resources; just use :member instead of :collection. For example, we could create actions that mark an article as embargoed or released—an embargoed article is invisible until released:

```
ActionController::Routing::Routes.draw do |map|
  map.resources :articles, :member => { :embargo => :put,
                                        :release => :put }
end
```

This adds the following routes to the standard set added by map.resources:

Method	URL Path	Action	Helper
PUT	/articles/1/embargo	embargo	embargo_article_url(:id => 1)
PUT	/articles/1/release	release	release_article_url(:id => 1)

It's also possible to create special actions that create new resources; use :new, passing it the same hash of :action => :method we used with :collection and :member. For example, we might need to create articles with just a title and a body—the summary is omitted.

We could create a special shortform action for this:

```
ActionController::Routing::Routes.draw do |map|
  map.resources :articles, :new => { :shortform => :post }
end
```

This adds the following routes to the standard set added by map.resources:

Method	URL Path	Action	Helper
POST	/articles/new/shortform	shortform	shortform_new_article_url

Nested Resources

Often our resources themselves contain additional collections of resources. For example, we may want to allow folks to comment on our articles. In this case, each comment would be a resource, and collections of comments would be associated with each article resource.

Rails provides a convenient and intuitive way of declaring the routes for this type of situation:

Download restful2/config/routes.rb

```
ActionController::Routing::Routes.draw do |map|
  map.resources :articles  do |article|
    article.resources :comments
  end

  # ...

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

There even is a shorthand for the simplest and most common case, such as this one:

```
map.resources :articles, :has_many => :comments
```

Both expressions are equivalent and define the top-level set of article routes and additionally create a set of subroutes for comments. Because the comment resources appear inside the articles block, a comment resource *must* be qualified by an article resource. This means that the path to a comment must always be prefixed by the path to a particular article. To fetch the comment with id 4 for the article with an id of 99, you'd use a path of /articles/99/comments/4.

Once again, we can see the full set of routes generated by our configuration by using the rake routes command:

```
                articles GET    /articles
                         {:controller=>"articles", :action=>"index"}
      formatted_articles GET    /articles.:format
                         {:controller=>"articles", :action=>"index"}
```

```
                                POST   /articles
                                {:controller=>"articles", :action=>"create"}
                                POST   /articles.:format
                                {:controller=>"articles", :action=>"create"}
                     new_article GET    /articles/new
                                {:controller=>"articles", :action=>"new"}
           formatted_new_article GET    /articles/new.:format
                                {:controller=>"articles", :action=>"new"}
                    edit_article GET    /articles/:id/edit
                                {:controller=>"articles", :action=>"edit"}
          formatted_edit_article GET    /articles/:id/edit.:format
                                {:controller=>"articles", :action=>"edit"}
                         article GET    /articles/:id
                                {:controller=>"articles", :action=>"show"}
              formatted_article GET    /articles/:id.:format
                                {:controller=>"articles", :action=>"show"}
                                PUT    /articles/:id
                                {:controller=>"articles", :action=>"update"}
                                PUT    /articles/:id.:format
                                {:controller=>"articles", :action=>"update"}
                                DELETE /articles/:id
                                {:controller=>"articles", :action=>"destroy"}
                                DELETE /articles/:id.:format
                                {:controller=>"articles", :action=>"destroy"}
                 article_comments GET    /articles/:article_id/comments
                                {:controller=>"comments", :action=>"index"}
       formatted_article_comments GET    /articles/:article_id/comments.:format
                                {:controller=>"comments", :action=>"index"}
                                POST   /articles/:article_id/comments
                                {:controller=>"comments", :action=>"create"}
                                POST   /articles/:article_id/comments.:format
                                {:controller=>"comments", :action=>"create"}
             new_article_comment GET    /articles/:article_id/comments/new
                                {:controller=>"comments", :action=>"new"}
   formatted_new_article_comment GET    /articles/:article_id/comments/new.:format
                                {:controller=>"comments", :action=>"new"}
            edit_article_comment GET    /articles/:article_id/comments/:id/edit
                                {:controller=>"comments", :action=>"edit"}
  formatted_edit_article_comment GET    /articles/:article_id/comments/:id/edit.:format
                                {:controller=>"comments", :action=>"edit"}
                  article_comment GET    /articles/:article_id/comments/:id
                                {:controller=>"comments", :action=>"show"}
       formatted_article_comment GET    /articles/:article_id/comments/:id.:format
                                {:controller=>"comments", :action=>"show"}
                                PUT    /articles/:article_id/comments/:id
                                {:controller=>"comments", :action=>"update"}
                                PUT    /articles/:article_id/comments/:id.:format
                                {:controller=>"comments", :action=>"update"}
                                DELETE /articles/:article_id/comments/:id
                                {:controller=>"comments", :action=>"destroy"}
                                DELETE /articles/:article_id/comments/:id.:format
                                {:controller=>"comments", :action=>"destroy"}
                                       /:controller/:action/:id
                                       /:controller/:action/:id.:format
```

Note here that the named route for /articles/:article_id/comments/:id is article_comment, not simply comment. This naming simply reflects the nesting of these resources.

We can extend our previous article's application to support these new routes. First, we'll create a model for comments and add a migration:

```
restful> ruby script/generate model comment \
         comment:text article_id:integer
```

Download restful2/db/migrate/20080601000002_create_comments.rb

```
class CreateComments < ActiveRecord::Migration
  def self.up
    create_table :comments do |t|
      t.text :comment
      t.integer :article_id

      t.timestamps
    end
  end

  def self.down
    drop_table :comments
  end
end
```

Second, we'll need to tell the article model that it now has associated comments. We'll also add a link back to articles from comments.

Download restful2/app/models/article.rb

```
class Article < ActiveRecord::Base
  has_many :comments
end
```

Download restful2/app/models/comment.rb

```
class Comment < ActiveRecord::Base
  belongs_to :article
end
```

And we run the migration:

```
restful> rake db:migrate
```

We'll create a CommentsController to manage the comments resource. We'll give it the same actions as the scaffold-generated articles controller, except we'll omit index and show, because comments are displayed only from an article's show action.

We'll update the show template for articles to display any comments, and we'll add a link to allow a new comment to be posted.

Download restful2/app/views/articles/show.html.erb

```
<p>
  <b>Title:</b>
  <%=h @article.title %>
</p>

<p>
  <b>Summary:</b>
  <%=h @article.summary %>
</p>

<p>
  <b>Content:</b>
  <%=h @article.content %>
</p>


<% unless @article.comments.empty? %>
  <%= render :partial => "/comments/comment",
                    :collection => @article.comments %>
<% end %>

<%= link_to "Add comment", new_article_comment_url(@article) %> |
<%= link_to 'Edit', edit_article_path(@article) %> |
<%= link_to 'Back', articles_path %>
```

This code illustrates a couple of interesting techniques. We use a partial template to display the comments, but that template is located in the directory app/views/comments. We tell Rails to look there by putting a leading / and the relative path in the render call. The code also uses the fact that routing helpers accept positional parameters. Rather than writing this:

```
new_article_comment_url(:article_id => @article.id)
```

we can use the fact that the :article field is the first in the route, and write this:

```
new_article_comment_url(@article)
```

However, the actions have a slightly different form; because comments are accessed only in the context of an article, we fetch the article before working on the comment itself. We also use the collection methods declared by has_many to double-check that we work only with comments belonging to the current article.

Download restful2/app/controllers/comments_controller.rb

```
class CommentsController < ApplicationController

  before_filter :find_article

  def new
    @comment = Comment.new
  end
```

```
  def edit
    @comment = @article.comments.find(params[:id])
  end

  def create
    @comment = Comment.new(params[:comment])
    if (@article.comments << @comment)
      redirect_to article_url(@article)
    else
      render :action => :new
    end
  end

  def update
    @comment = @article.comments.find(params[:id])
    if @comment.update_attributes(params[:comment])
      redirect_to article_url(@article)
    else
      render :action => :edit
    end
  end

  def destroy
    comment = @article.comments.find(params[:id])
    @article.comments.delete(comment)
    redirect_to article_url(@article)
  end

private

  def find_article
    @article_id = params[:article_id]
    return(redirect_to(articles_url)) unless @article_id
    @article = Article.find(@article_id)
  end

end
```

The full source code for this application, showing the additional views for comments, is available online.

Shallow Route Nesting

At times, nested resources can produce cumbersome URL. A solution to this is to use shallow route nesting:

```
map.resources :articles, :shallow => true do |article|
  article.resources :comments
end
```

This will enable the recognition of the following routes:

```
/articles/1          => article_path(1)
/articles/1/comments => article_comments_path(1)
/comments/2          => comment_path(2)
```

Try rake routes to see the full mapping.

Selecting a Data Representation

One of the goals of a REST architecture is to decouple data from its representation. If a human user uses the URL path /articles to fetch some articles, they should see a nicely formatted HTML. If an application asks for the same URL, it could elect to receive the results in a code-friendly format (YAML, JSON, or XML, perhaps).

We've already seen how Rails can use the HTTP Accept header in a respond_to block in the controller. However, it isn't always easy (and sometimes it's plain impossible) to set the Accept header. To deal with this, Rails allows you to pass the format of response you'd like as part of the URL. To do this, set a :format parameter in your routes to the file extension of the MIME type you'd like returned. The easiest way to do this is by adding a field called :format to your route definitions:

```
map.store "/store/:action/:id.:format", :id => nil, :format => nil
```

Because a full stop (period) is a separator character in route definitions, :format is treated as just another field. Because we give it a nil default value, it's an optional field.

Having done this, we can use a respond_to block in our controllers to select our response type depending on the requested format:

```
def show
  respond_to do |format|
    format.html
    format.xml  { render :xml => @product.to_xml }
    format.yaml { render :text => @product.to_yaml }
  end
end
```

Given this, a request to /store/show/1 or /store/show/1.html will return HTML content, while /store/show/1.xml will return XML and /store/show/1.yaml will return YAML. You can also pass the format in as an HTTP request parameter:

```
GET HTTP://pragprog.com/store/show/123?format=xml
```

The routes defined by map.resources have this facility enabled by default.

Although the idea of having a single controller that responds with different content types seems appealing, the reality is tricky. In particular, it turns out that error handling can be tough. Although it's acceptable on error to redirect a user to a form, showing them a nice flash message, you have to adopt a different strategy when you serve XML. Consider your application architecture carefully before deciding to bundle all your processing into single controllers.

Rails makes it simple to develop an application that is based on Resource-based routing. Many claim it greatly simplifies the coding of their applications. However, it isn't always appropriate. Don't feel compelled to use it if you can't find a way of making it work. And you can always mix and match. Some controllers can be resourced based, and others can be based on actions. Some controllers can even be resource based with a few extra actions.

21.4 Testing Routing

So far we've experimented with routes by poking at them manually using script/console. When it comes time to roll out an application, though, we might want to be a little more formal and include unit tests that verify our routes work as expected. Rails includes a number of test helpers that make this easy:

assert_generates(path, options, defaults={}, extras={}, message=nil)

Verifies that the given set of options generates the specified path.

Download e1/routing/test/unit/routing_test.rb

```
def test_generates
  ActionController::Routing.use_controllers! ["store"]
  load "config/routes.rb"

  assert_generates("/store", :controller => "store", :action => "index")
  assert_generates("/store/list", :controller => "store", :action => "list")
  assert_generates("/store/add_to_cart/1",
                   { :controller => "store", :action => "add_to_cart",
                     :id => "1", :name => "dave" },
                   {}, { :name => "dave"})
end
```

The extras parameter is used to tell the request the names and values of additional request parameters (in the third assertion in the previous code, this would be ?name=dave). The test framework does not add these as strings to the generated URL; instead, it tests that the values it would have added appears in the extras hash.

The defaults parameter is unused.

assert_recognizes(options, path, extras={}, message=nil)

Verifies that routing returns a specific set of options given a path.

Download e1/routing/test/unit/routing_test.rb

```
def test_recognizes
  ActionController::Routing.use_controllers! ["store"]
  load "config/routes.rb"
```

```
  # Check the default index action gets generated
  assert_recognizes({"controller" => "store", "action" => "index"}, "/store")

  # Check routing to an action
  assert_recognizes({"controller" => "store", "action" => "list"},
                    "/store/list")

  # And routing with a parameter
  assert_recognizes({ "controller" => "store",
                      "action" => "add_to_cart",
                     "id" => "1" },
                    "/store/add_to_cart/1")

  # And routing with a parameter
  assert_recognizes({ "controller" => "store",
                      "action" => "add_to_cart",
                      "id" => "1",
                      "name" => "dave" },
                    "/store/add_to_cart/1",
                    { "name" => "dave" } ) # like having ?name=dave after the URL

  # Make it a post request
  assert_recognizes({ "controller" => "store",
                      "action" => "add_to_cart",
                      "id" => "1" },
                    { :path => "/store/add_to_cart/1", :method => :post })
end
```

The extras parameter again contains the additional URL parameters. In the fourth assertion in the preceding code example, we use the extras parameter to verify that, had the URL ended ?name=dave, the resulting params hash would contain the appropriate values.[8]

The :conditions parameter lets you specify routes that are conditional on the HTTP verb of the request. You can test these by passing a hash, rather than a string, as the second parameter to assert_recognizes. The hash should contain two elements: :path will contain the incoming request path, and :method will contain the HTTP verb to be used.

Download e1/routing/test/unit/routing_conditions_test.rb

```
def test_method_specific_routes
  assert_recognizes({"controller" => "store", "action" => "display_checkout_form"},
                    :path => "/store/checkout", :method => :get)
  assert_recognizes({"controller" => "store", "action" => "save_checkout_form"},
                    :path => "/store/checkout", :method => :post)
end
```

8. Yes, it is strange that you can't just put ?name=dave on the URL itself.

assert_routing(path, options, defaults={}, extras={}, message=nil)

Combines the previous two assertions, verifying that the path generates the options and then that the options generate the path.

Download e1/routing/test/unit/routing_test.rb

```
def test_routing
  ActionController::Routing.use_controllers! ["store"]
  load "config/routes.rb"

  assert_routing("/store", :controller => "store", :action => "index")
  assert_routing("/store/list", :controller => "store", :action => "list")
  assert_routing("/store/add_to_cart/1",
                 :controller => "store", :action => "add_to_cart", :id => "1")
end
```

It's important to use symbols as the keys and use strings as the values in the options hash. If you don't, asserts that compare your options with those returned by routing will fail.

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style, and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

Agile Web Development with Rails
http://pragprog.com/titles/rails3
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragprog.com/news
Check out the latest pragmatic developments in the news.

Buy the Book

If you liked this PDF, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragmaticprogrammer.com/titles/rails3.

Contact Us

Phone Orders:	1-800-699-PROG (+1 919 847 3884)
Online Orders:	www.pragmaticprogrammer.com/catalog
Customer Service:	orders@pragmaticprogrammer.com
Non-English Versions:	translations@pragmaticprogrammer.com
Pragmatic Teaching:	academic@pragmaticprogrammer.com
Author Proposals:	proposals@pragmaticprogrammer.com