CHAPTER 3
Routing

I dreamed a thousand new paths. . . I woke and walked my old one.
—Chinese proverb

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

This chapter will introduce you to the principal techniques for defining and manipulating routes. The next chapter will build on this knowledge to explore the facilities Rails offers in support of writing applications that comply with the principles of Representational State Transfer (REST). As you'll see, those facilities can be of tremendous use to you even if you're not planning to scale the heights of REST theorization.

Many of the examples in these two chapters are based on a small auction application. The examples are kept simple enough that they should be comprehensible on their own. The basic idea is that there are auctions; each auction involves auctioning off an item; there are users; and users submit bids. That's most of it.

The triggering of a controller action is the main event in the life cycle of a connection to a Rails application. So it makes sense that the process by which Rails determines *which* controller and *which* action to execute must be very important. That process is embodied in the routing system.

The routing system maps URLs to actions. It does this by applying rules—rules that you specify, using Ruby commands, in the configuration file `config/routes.rb`. If you don't override the file's default rules, you'll get some reasonable behavior. But it doesn't take much work to write some custom rules and reap the benefits of the flexibility of the routing system.

Moreover, the routing system actually does two things: It maps requests to actions, and it writes URLs for you for use as arguments to methods like `link_to`, `redirect_to`, and `form_tag`. The routing system knows how to turn a visitor's request URL into a controller/action sequence. It also knows how to manufacture URL strings based on your specifications.

When you do this:

```
<%= link_to "Items", :controller => "items", :action => "list" %>
```

the routing system provides the following URL to the `link_to` helper:

```
http://localhost:3000/items/list
```

The routing system is thus a powerful, two-way routing complex. It *recognizes* URLs, routing them appropriately; and it *generates* URLs, using the routing rules as a template or blueprint for the generated string. We'll keep an eye on both of these important purposes of the routing system as we proceed.

The Two Purposes of Routing

Recognizing URLs is useful because it's how your application decides what it's supposed to do when a particular request comes in:

```
http://localhost:3000/myrecipes/apples   What do we do now?!
```

Generating URLs is useful because it allows you to use relatively high-level syntax in your view templates and controllers when you need to insert a URL—so you don't have to do this:

```
<a href="http://localhost:3000/myrecipes/apples">My Apple Recipes</a>
Not much fun having to type this out by hand!
```

The routing system deals with both of these issues: how to interpret (recognize) a request URL and how to write (generate) a URL. It performs both of these functions based on rules that you provide. The rules are inserted into the file `config/routes.rb`, using a special syntax. (Actually it's just Ruby program code, but it uses special methods and parameters.)

Each rule—or, to use the more common term, simply each *route*—includes a pattern string, which will be used both as a template for matching URLs and as a blueprint for writing them. The pattern string contains a mixture of static substrings, forward slashes (it's mimicking URL syntax), and wildcard positional parameters that serve as "receptors" for corresponding values in a URL, for purposes of both recognition and generation.

A route can also include one or more bound parameters, in form of key/value pairs in a hash. The fate of these key/value pairs depends on what the key is. A couple of keys (`:controller` and `:action`) are "magic," and determine what's actually going to happen. Other keys (`:blah`, `:whatever`, etc.) get stashed for future reference.

Putting some flesh on the bones of this description, here's a sample route, related to the preceding examples:

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

In this example, you can see:

- A static string (`myrecipes`)
- A wildcard URL component (`:ingredient`)
- Bound parameters (`:controller => "recipes"`, `:action => "show"`)

Routes have a pretty rich syntax—this one isn't by any means the most complex (nor the most simple)—because they have to do so much. A single route, like the one in this example, has to provide enough information both to match an existing URL *and* to manufacture a new one. The route syntax is engineered to address both of these processes.

It's actually not hard to grasp, if you take each type of field in turn. We'll do a run-through using the "ingredient" route. Don't worry if it doesn't all sink in the first time

through. We'll be unpacking and expanding on the techniques and details throughout the chapter.

As we go through the route anatomy, we'll look at the role of each part in both URL recognition and URL generation. Keep in mind that this is just an introductory example. You can do lots of different things with routes, but examining this example will give you a good start in seeing how it all works.

Bound Parameters

If we're speaking about route recognition, the bound parameters—key/value pairs in the hash of options at the end of the route's argument list—determine what's going to happen if and when this route matches an incoming URL. Let's say someone requests this URL from their web browser:

```
http://localhost:3000/myrecipes/apples
```

This URL will match the ingredient route. The result will be that the `show` action of the recipes controller will be executed. To see why, look at the route again:

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

The `:controller` and `:action` keys are *bound*: This route, when matched by a URL, will always take the visitor to exactly that controller and that action. You'll see techniques for matching controller and action based on *wildcard* matching shortly. In this example, though, there's no wildcard involved. The controller and action are hard-coded.

Now, when you're generating a URL for use in your code, you provide values for all the necessary bound parameters. That way, the routing system can do enough match-ups to find the route you want. (In fact, Rails will complain by raising an exception if you don't supply enough values to satisfy a route.)

The parameters are usually bundled in a hash. For example, to generate a URL from the ingredient route, you'd do something like this:

```
<%= link_to "Recipe for apples",
      :controller => "recipes",
      :action     => "show",
      :ingredient => "apples" %>
```

The values `"recipes"` and `"show"` for `:controller` and `:action` will match the ingredient route, which contains the same values for the same parameters. That means that the pattern string in that route will serve as the template—the blueprint—for the generated URL.

The use of a hash to specify URL components is common to all the methods that produce URLs (`link_to`, `redirect_to`, `form_for`, etc.). Underneath, these methods are making their own calls to `url_for`, a lower-level URL generation method that we'll talk about more a little further on.

We've left `:ingredient` hanging. It's a wildcard component of the pattern string.

Wildcard Components ("Receptors")

The symbol `:ingredient` inside the quoted pattern in the route is a wildcard parameter (or variable). You can think of it as a *receptor*: Its job is to be latched onto by a value. Which value latches onto which wildcard is determined positionally, lining the URL up with the pattern string:

```
http://localhost:3000/myrecipes/apples          Someone connects to this URL...
                        'myrecipes/:ingredient'  which matches this pattern string
```

The `:ingredient` receptor, in this example, receives the value `apples` from the URL. What that means for you is that the value `params[:ingredient]` will be set to the string `"apples"`. You can access that value inside your `recipes/show` action. When you generate a URL, you have to supply values that will attach to the receptors—the wildcard symbols inside the pattern string. You do this using key => value syntax. That's the meaning of the last line in the preceding example:

```
<%= link_to "My Apple Recipes",
      :controller => "recipes",
      :action     => "show",
      :ingredient => "apples" %>
```

In this call to `link_to`, we've provided values for three parameters. Two of them are going to match hard-coded, bound parameters in the route; the third, `:ingredient`, will be assigned to the slot in the URL corresponding to the `:ingredient` slot in the pattern string.

But they're all just hash key/value pairs. The call to `link_to` doesn't "know" whether it's supplying hard-coded or wildcard values. It just knows (or hopes!) that these three values, tied to these three keys, will suffice to pinpoint a route—and therefore a pattern string, and therefore a blueprint for a URL.

Static Strings

Our sample route contains a static string inside the pattern string: `recipes`.

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

This string anchors the recognition process. When the routing system sees a URL that starts `/recipes`, it will match that to the static string in the ingredient route. Any URL that does not contain the static string `recipes` in the leftmost slot will not match this route.

As for URL generation, static strings in the route simply get placed, positionally, in the URL that the routing system generates. Thus the `link_to` example we've been considering

```
<%= link_to "My Apple Recipes",
      :controller => "recipes",
      :action     => "show",
      :ingredient => "apples" %>
```

will generate the following HTML:

```
<a href="http://localhost:3000/myrecipes/apples">My Apple Recipes</a>
```

The string `myrecipes` did not appear in the `link_to` call. The *parameters* of the `link_to` call triggered a match to the ingredients route. The URL generator then used that route's pattern string as the blueprint for the URL it generated. The pattern string stipulates the substring `myrecipes`.

URL *recognition* and URL *generation*, then, are the two jobs of the routing system. It's a bit like the address book stored in a cell phone. When you select "Gavin" from your contact list, the phone looks up the phone number. And when Gavin calls you, the phone figures out *from* the number provided by caller ID that the caller is

Gavin; that is, it recognizes the number and maps it to the value "Gavin", which is displayed on the phone's screen.

Rails routing is a bit more complex than cell phone address book mapping, because there are variables involved. It's not just a one-to-one mapping. But the basic idea is the same: recognize what comes in as requests, and generate what goes into the code as HTML.

We're going to turn next to the routing rules themselves. As we go, you should keep the dual purpose of recognition/generation in mind. There are two principles that are particularly useful to remember:

- *The same rule* governs both recognition and generation. The whole system is set up so that you don't have to write rules twice. You write each rule once, and the logic flows through it in both directions.
- The URLs that are generated by the routing system (via link_to and friends) *only make sense to the routing system.* The path recipes/apples, which the system generates, contains not a shred of a clue as to what's supposed to happen—except insofar as it maps to a routing rule. The routing rule then provides the necessary information to trigger a controller action. Someone looking at the URL without knowing the rules won't know what the URL means.

You'll see how these play out in detail as we proceed.

The routes.rb File

Routes are defined in the file config/routes.rb, as shown (with some extra comments) in Listing 3.1. This file is created when you first create your Rails application. It comes with a few routes already written and in most cases you'll want to change and/or add to the routes defined in it.

Listing 3.1 The Default routes.rb File

```
ActionController::Routing::Routes.draw do |map|
  # The priority is based upon order of creation
  # First created gets highest priority.

  # Sample of regular route:
  # map.connect 'products/:id', :controller => 'catalog',
                                :action => 'view'
```

```
  # Keep in mind you can assign values other than
  # :controller and :action

  # Sample of named route:
  # map.purchase 'products/:id/purchase', :controller => 'catalog',
                                          :action => 'purchase'
  # This route can be invoked with purchase_url(:id => product.id)

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

  # Allow downloading Web Service WSDL as a file with an extension

  # instead of a file named 'wsdl'
  map.connect ':controller/service.wsdl', :action => 'wsdl'

  # Install the default route as the lowest priority.
  map.connect ':controller/:action/:id.:format'
  map.connect ':controller/:action/:id'
end
```

The whole thing consists of a single call to the method `ActionController::Routing::Routes.draw`. That method takes a block, and everything from the second line of the file to the second-to-last line is body of that block.

Inside the block, you have access to a variable called `map`. It's an instance of the class `ActionController::Routing::RouteSet::Mapper`. Through it you configure the Rails routing system: You define routing rules by calling methods on your mapper object. In the default `routes.rb` file you see several calls to `map.connect`. Each such call (at least, those that aren't commented out) creates a new route by registering it with the routing system.

The routing system has to find a pattern match for a URL it's trying to recognize, or a parameters match for a URL it's trying to generate. It does this by going through the rules—the routes—in the order in which they're defined; that is, the order in which they appear in `routes.rb`. If a given route fails to match, the matching routine falls through to the next one. As soon as any route succeeds in providing the necessary match, the search ends.

Courtenay Says...

Routing is probably one of the most complex parts of Rails. In fact, for much of Rails' history, only one person could make any changes to the source, due to its labrynthine implementation. So, don't worry too much if you don't grasp it immediately. Most of us still don't.

That being said, the `routes.rb` syntax is pretty straightforward if you follow the rules. You'll likely spend less than 5 minutes setting up routes for a vanilla Rails project.

The Default Route

If you look at the very bottom of `routes.rb` you'll see the *default route*:

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

The default route is in a sense the end of the journey; it defines what happens when nothing else happens. However, it's also a good place to start. If you understand the default route, you'll be able to apply that understanding to the more intricate examples as they arise.

The default route consists of just a pattern string, containing three wildcard "receptors." Two of the receptors are `:controller` and `:action`. That means that this route determines what it's going to do based entirely on wildcards; there are no bound parameters, no hard-coded controller or action.

Here's a sample scenario. A request comes in with the URL:

```
http://localhost:3000/auctions/show/1
```

Let's say it doesn't match any other pattern. It hits the last route in the file—the default route. There's definitely a congruency, a match. We've got a route with three receptors, and a URL with three values, and therefore three positional matches:

```
:controller/:action/:id
 auctions  / show  / 1
```

We end up, then, with the auctions controller, the `show` action, and "1" for the id value (to be stored in `params[:id]`). The dispatcher now knows what to do.

The behavior of the default route illustrates some of the specific default behaviors of the routing system. The default action for any request, for example, is index. And, given a wildcard like `:id` in the pattern string, the routing system prefers to find a value for it, but will go ahead and assign it *nil* rather than give up and conclude that there's no match.

Table 3.1 shows some examples of URLs and how they will map to this rule, and with what results.

Table 3.1 Default Route Examples

URL	Result		Value of `id`
	Controller	**Action**	
`/auctions/show/3`	`auctions`	`show`	`3`
`/auctions/index`	`auctions`	`index`	*nil*
`/auctions`	`auctions`	`index (default)`	*nil*
`/auctions/show`	`auctions`	`show`	*nil*—probably an error!

The *nil* in the last case is probably an error because a `show` action with no id is usually not what you'd want!

Spotlight on the `:id` Field

Note that the treatment of the `:id` field in the URL is not magic; it's just treated as a value with a name. If you wanted to, you could change the rule so that `:id` was `:blah`—but then you'd have to remember to do this in your controller action:

```
@auction = Auction.find(params[:blah])
```

The name `:id` is simply a convention. It reflects the commonness of the case in which a given action needs access to a particular database record. The main business of the router is to determine the controller and action that will be executed. The `id` field is a bit of an extra; it's an opportunity for actions to hand a data field off to each other.

The `id` field ends up in the `params` hash, which is automatically available to your controller actions. In the common, classic case, you'd use the value provided to dig a record out of the database:

```
class ItemsController < ApplicationController
  def show
    @item = Item.find(params[:id])
  end
end
```

Default Route Generation

In addition to providing the basis for recognizing URLs, and triggering the correct behavior, the default route also plays a role in URL generation. Here's a `link_to` call that will use the default route to generate a URL:

```
<%= link_to item.description,
    :controller => "item",
    :action => "show",
    :id => item.id %>
```

This code presupposes a local variable called `item`, containing (we assume) an `Item` object. The idea is to create a hyperlink to the `show` action for the item controller, and to include the `id` of this particular item. The hyperlink, in other words, will look something like this:

```
<a href="localhost:3000/item/show/3">A signed picture of Houdini</a>
```

This URL gets created courtesy of the route-generation mechanism. Look again at the default route:

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

In our `link_to` call, we've provided values for all three of the fields in the pattern. All that the routing system has to do is plug in those values and insert the result into the URL:

```
item/show/3
```

When someone clicks on the link, that URL will be recognized—courtesy of the other half of the routing system, the recognition facility—and the correct controller and action will be executed, with `params[:id]` set to 3.

The generation of the URL, in this example, uses wildcard logic: We've supplied three symbols, `:controller`, `:action`, and `:id`, in our pattern string, and those symbols will be replaced, in the generated URL, by whatever values we supply. Contrast this with our earlier example:

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

To get the URL generator to choose this route, you have to specify "recipes" and "show" for `:controller` and `:action` when you request a URL for `link_to`. In the default route—and, indeed, any route that has symbols embedded in its pattern—you still have to match, but you can use any value.

Modifying the Default Route

A good way to get a feel for the routing system is by changing things and seeing what happens. We'll do this with the default route. You'll probably want to change it back… but changing it will show you something about how routing works.

Specifically, swap `:controller` and `:action` in the pattern string:

```
# Install the default route as the lowest priority.
map.connect ':action/:controller/:id'
```

You've now set the default route to have actions first. That means that where previously you might have connected to http://localhost:3000/auctions/show/3, you'll now need to connect to http://localhost:3000/show/auctions/3. And when you generate a URL from this route, it will come out in the /show/auctions/3 order.

It's not particularly logical; the original default (the default default) route is better. But it shows you a bit of what's going on, specifically with the magic symbols `:controller` and `:action`. Try a few more changes, and see what effect they have. (And then put it back the way it was!)

The Ante-Default Route and `respond_to`

The route just before the default route (thus the "ante-default" route) looks like this:

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

The `.:format` at the end matches a literal dot and a wildcard "format" value after the id field. That means it will match, for example, a URL like this:

```
http://localhost:3000/recipe/show/3.xml
```

Here, `params[:format]` will be set to `xml`. The `:format` field is special; it has an effect inside the controller action. That effect is related to a method called `respond_to`.

The `respond_to` method allows you to write your action so that it will return different results, depending on the requested format. Here's a `show` action for the items controller that offers either HTML or XML:

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

The `respond_to` block in this example has two clauses. The HTML clause just consists of `format.html`. A request for HTML will be handled by the usual rendering of the RHTML view template. The XML clause includes a code block; if XML is requested, the block will be executed and the result of its execution will be returned to the client.

Here's a command-line illustration, using `wget` (slightly edited to reduce line noise):

```
$ wget http://localhost:3000/items/show/3.xml -O -
Resolving localhost... 127.0.0.1, ::1
Connecting to localhost|127.0.0.1|:3000... connected.
HTTP request sent, awaiting response... 200 OK
Length: 295 [application/xml]
<item>
```

```
  <created-at type="datetime">2007-02-16T04:33:00-05:00</created-at>
  <description>Violin treatise</description>
  <id type="integer">3</id>
  <maker>Leopold Mozart</maker>
  <medium>paper</medium>
  <modified-at type="datetime"></modified-at>
  <year type="integer">1744</year>
</item>
```

The `.xml` on the end of the URL results in `respond_to` choosing the "xml" branch, and the returned document is an XML representation of the item.

`respond_to` and the `HTTP-Accept` Header

You can also trigger a branching on `respond_to` by setting the `HTTP-Accept` header in the request. When you do this, there's no need to add the `.:format` part of the URL.

Here's a `wget` example that does not use `.xml` but does set the `Accept` header:

```
wget http://localhost:3000/items/show/3 -O - —header="Accept:
  text/xml"
Resolving localhost... 127.0.0.1, ::1
Connecting to localhost|127.0.0.1|:3000... connected.
HTTP request sent, awaiting response...
200 OK
Length: 295 [application/xml]
<item>
  <created-at type="datetime">2007-02-16T04:33:00-05:00</created-at>
  <description>Violin treatise</description>
  <id type="integer">3</id>
  <maker>Leopold Mozart</maker>
  <medium>paper</medium>
  <modified-at type="datetime"></modified-at>
  <year type="integer">1744</year>
</item>
```

The result is exactly the same as in the previous example.

The Empty Route

Except for learning-by-doing exercises, you're usually safe leaving the default route alone. But there's another route in `routes.rb` that plays something of a default role and you will probably want to change it: the empty route.

A few lines up from the default route (refer to Listing 3.1) you'll see this:

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

What you're seeing here is the empty route—that is, a rule specifying what should happen when someone connects to

```
http://localhost:3000
```
Note the lack of "/anything" at the end!

The empty route is sort of the opposite of the default route. Instead of saying, "I need any three values, and I'll use them as controller, action, and id," the empty route says, "I don't want *any* values; I want *nothing*, and I already know what controller and action I'm going to trigger!"

In a newly generated `routes.rb` file, the empty route is commented out, because there's no universal or reasonable default for it. You need to decide what this "nothing" URL should do for each application you write.

Here are some examples of fairly common empty route rules:

```
map.connect '', :controller => "main", :action => "welcome"
map.connect '', :controller => "top", :action => "login"
map.connect '', :controller => "main"
```

That last one will connect to `main/index`—`index` being the default action when there's none specified.

Note that Rails 2.0 introduces a mapper method named `root` which becomes the proper way to define the empty route for a Rails application, like this:

```
map.root :controller => "homepage"
```

Defining the empty route gives people something to look at when they connect to your site with nothing but the domain name.

Writing Custom Routes

The default route is a very general one. Its purpose is to catch all routes that haven't matched already. Now we're going to look at that *already* part: the routes defined earlier in the `routes.rb` file, routes that match more narrowly than the general one at the bottom of the file.

You've already seen the major components that you can put into a route: static strings, bound parameters (usually including `:controller` and often including `:action`), and wildcard "receptors" that get their values either positionally from a URL, or key-wise from a URL hash in your code.

When you write your routes, you have to think like the routing system.

- On the recognition side, that means your route has to have enough information in it—either hard-coded or waiting to receive values from the URL—to decide which controller and action to choose. (Or at least a controller; it can default to `index` if that's what you want.)
- On the generation side, your need to make sure that your hard-coded parameters and wildcards, taken together, provide you with enough values to pinpoint a route to use.

As long as these things are present—and as long as your routes are listed in order of priority ("fall-through" order)—your routes should work as desired.

Using Static Strings

Keep in mind that there's no necessary correspondence between the number of fields in the pattern string, the number of bound parameters, and the fact that every connection needs a controller and an action.

For example, you *could* write a route like this:

```
map.connect ":id",  :controller => "auctions", :action => "show"
```

which would recognize a URL like this:

```
http://localhost:3000/8
```

The routing system would set `params[:id]` to `8` (based on the position of the `:id` "receptor," which matches the position of "8" in the URL), and it would execute

the `show` action of the `auctions` controller. Of course, this is a bit of a stingy route, in terms of visual information. You might want to do something more like Listing 2.2, which is a little richer semantically-speaking:

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

This version of the route would recognize this:

```
http://localhost:3000/auctions/8
```

In this route, "auctions" is a static string. It will be looked for in the URL, for recognition purposes; and it will be inserted into the URL when you generate it with the following code:

```
<%= link_to "Auction details",
    :controller => "auctions",
    :action => "show",
    :id => auction.id %>
```

Using Your Own "Receptors"

So far, we've used the two magic parameters, `:controller` and `:action`, and the nonmagic but standard `:id`. It is also possible to use your own parameters, either hard-coded or wildcard. Doing this can help you add some expressiveness and self-documentation to your routes, as well as to your application code.

The main reason you'd want to use your own parameters is so that you can use them as handles in your code. For example, you might want a controller action to look like this:

```
def show
  @auction = Auction.find(params[:id])
  @user = User.find(params[:user_id])
end
```

Here we've got the symbol `:user_id` showing up, along with `:id`, as a key to the params hash. That means it got there, somehow. In fact, it got there the same way as

the :id parameter: It appears in the pattern for the route by which we got to the show action in the first place.

Here's that route:

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

This route, when faced with a URL like this

```
/auctions/3/1
```

will cause the auctions/show action to run, and will set both :user_id and :id in the params hash. (:user_id matches 3 positionally, and :id matches 1.)

On the URL generation side, all you have to do is include a :user_id key in your URL specs:

```
<%= link_to "Auction",
    :controller => "auctions",
    :action => "show",
    :user_id => current_user.id,
    :id => ts.id %>
```

The :user_id key in the hash will match the :user_id receptor in the route pattern. The :id key will also match, and so will the :controller and :action parameters. The result will be a URL based on the blueprint 'auctions/:user_id/:id'.

You can actually arbitrarily add many specifiers to a URL hash in calls to link_to and other similar methods. Any parameters you define that aren't found in a routing rule will be added to the URL as a query string. For example, if you add:

```
:some_other_thing => "blah"
```

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

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

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, which means that 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 them. But if they connect to `/any_other_controller/help`, they get the `help` action of the main controller. Yes, it's tricky.

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.

It's very similar to other kinds of matching operations, like `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.

Using Regular Expressions in Routes

Sometimes you want not only to recognize a route, but 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.[1]

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"
```

If you want to do so, mainly for clarity, you can wrap your regular expression-based constraints in a special hash parameter named `:requirements`, like this:

```
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 that the values you're dealing with are usable and appropriate for your application's domain.

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 learn something about how

Rails generates URLs. (And you might want to call `url_for` on its own at some point.)

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 belonging to the auctions controller, you could create a link to the `edit` action like this:

```
<%= link_to "Edit 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. Because there's no `:controller` specified in the URL hash, the generator will fall back on auctions, and based on the default route (`:controller/:action/:id`), it will come up with this (for auction 5):

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

The same is true of the action. If you don't supply an `:action` key, then 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.

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 the way defaults work in the `url_for` method. 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 it 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. Since 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 non-default 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 you did this in the `show.rhtml` template:

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

Answer: Since `: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, since 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*. Since 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.

Using Literal URLs

You can, if you wish, 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" %>
```

However, using a literal path or URL bypasses the routing system. If you write literal URLs, you're on your own to maintain them. (You can of course use Ruby's string interpolation techniques to insert values, if that's appropriate for what you're doing, but really stop and think about whether you are reinventing Rails functionality if you go down that path.)

Route Globbing

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

```
/files/list/base/books/fiction/dickens
```

you want the `files/list` action to have access to all four remaining fields. But sometimes there might be only three fields:

```
/files/list/base/books/fiction
```

or five:

```
/files/list/base/books/fiction/dickens/little_dorrit
```

So you need a route that will match (in this particular case) *everything after the second URI component.*

You can do that with a *route glob*. You "glob" the route with an asterisk:

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

Now, the `files/list` action will have access to an array of URI fields, accessible via `params[:specs]`:

```
def list
  specs = params[:specs] # e.g, ["base", "books", "fiction", "dickens"]
end
```

The glob has to come at the end of the pattern string in the route. You *cannot* do this:

```
map.connect 'files/list/*specs/dickens'  # Won't work!
```

The glob sponges up all the remaining URI components, and its semantics therefore require that it be the last thing in the pattern string.

Globbing Key-Value Pairs

Route globbing might provide the basis for a general mechanism for fielding queries about items up for auction. Let's say you devise a URI scheme that takes the following form:

```
http://localhost:3000/items/field1/value1/field2/value2/...
```

Making requests in this way will return a list of all items whose fields match the values, based on an unlimited set of pairs in the URL.

In other words, `http://localhost:3000/items/year/1939/medium/wood` would generate a list of all wood items made in 1939.

The route that would accomplish this would be:

```
map.connect 'items/*specs', :controller => "items", :action => "specify"
```

Of course, you'll have to write a `specify` action like the one in Listing 3.2 to support this route.

Listing 3.2 The `specify` Action

```
def specify
  @items = Item.find(:all, :conditions => Hash[params[:specs]])
  if @items.any?
    render :action => "index"
  else
    flash[:error] = "Can't find items with those properties"
    redirect_to :action => "index"
  end
end
```

How about that square brackets class method on `Hash`, eh? It converts a one-dimensional array of key/value pairs into a hash! Further proof that in-depth knowledge of Ruby is a prerequisite for becoming an expert Rails developer.

Next stop: Named routes, a way to encapsulate your route logic in made-to-order helper methods.

Named Routes

The topic of named routes almost deserves a chapter of its own. What you learn here will feed directly into our examination of REST-related routing in Chapter 4.

The idea of naming a route is basically to make life easier on you, the programmer. There are no outwardly visible effects as far as the application is concerned. When you name a route, a new method gets defined for use in your controllers and views; the method is called *name*`_url` (with *name* being the name you gave the route), and calling the method, with appropriate arguments, results in a URL being generated for the route. In addition, a method called *name*`_path` also gets created; this method generates just the path part of the URL, without the protocol and host components.

Creating a Named Route

The way you name a route is by calling a method on your mapper object with the name you want to use, instead of the usual connect:

```
map.help 'help',
  :controller => "main",
  :action     => "show_help"
```

In this example, you'll get methods called `help_url` and `help_path`, which you can use wherever Rails expects a URL or URL components:

```
<%= link_to "Help!", help_path %>
```

And, of course, the usual recognition and generation rules are in effect. The pattern string consists of just the static string component `"help"`. Therefore, the path you'll see in the hyperlink will be

```
/help
```

When someone clicks on the link, the `show_help` action of the `main` controller will be invoked.

The Question of Using *name*_path Versus *name*_url

When you create a named route, you're actually creating at least two route helper methods. In the preceding example, those two methods are `help_url` and `help_path`. The difference is that the `_url` method generates an entire URL, including protocol and domain, whereas the `_path` method generates just the path part (sometimes referred to as a *relative* path).

According to the HTTP spec, redirects should specify a URI, which can be interpreted (by some people) to mean a fully-qualified URL[2]. Therefore, if you want to be pedantic about it, you probably *should* always use the `_url` version when you use a named route as an argument to `redirect_to` in your controller code.

The `redirect_to` method seems to work perfectly with the relative paths generated by `_path` helpers, which makes arguments about the matter kind of pointless. In fact, other than redirects, permalinks, and a handful of other edge cases, it's the *Rails way* to use `_path` instead of `_url`. It produces a shorter string and the user agent (browser or otherwise) should be able to infer the fully qualified URL whenever it needs to do so, based on the HTTP headers of the request, a base element in the document, or the URL of the request.

As you read this book, and as you examine other code and other examples, the main thing to remember is that `help_url` and `help_path` are basically doing the same thing. I tend to use the `_url` style in general discussions about named route techniques, but to use `_path` in examples that occur inside view templates (for example, with `link_to` and `form_for`). It's mostly a writing-style thing, based on the theory that the URL version is more general and the path version more specialized. In

any case, it's good to get used to seeing both and getting your brain to view them as very closely connected.

Considerations

Named routes save you some effort when you need a URL generated. A named route zeros in directly on the route you need, bypassing the matching process. That means you don't have to provide as much detail as you otherwise would. You have to provide a value for any wildcard parameter in the route's pattern string, but you don't have to go down the laundry list of hard-coded, bound parameters. The only reason for doing that when you're trying to generate a URL is to steer the routing system to the correct route. But when you use a named route, the system already knows which rule you want to apply, and there is a (slight) corresponding performance boost.

What to Name Your Routes

The best way to figure out what named routes you'll need is to think top-down; that is, think about what you want to write in your application code, and then create the routes that will make it possible.

Take, for example, this call to `link_to`:

```
<%= link_to "Auction of #{h(auction.item.description)}",
    :controller => "auctions",
    :action     => "show",
    :id         => auction.id %>
```

The routing rule to match that path is this (generic type of route):

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

It seems a little heavy-handed to spell out all the routing parameters again, just so that the routing system can figure out which route we want. And it sure would be nice to shorten that `link_to` code. After all, the routing rule already specifies the controller and action.

This is a good candidate for a named route. We can improve the situation by introducing `auction_path`:

```
<%= link_to "Auction for #{h(auction.item.description)}",
      auction_path(:id => auction.id) %>
```

Giving the route a name is a shortcut; it takes us straight to that route, without a long search and without having to provide a thick description of the route's hard-coded parameters.

> **Courtenay Says...**
>
> Remember to escape your item descriptions!
>
> Links such as `#{auction.item.description}` should always be wrapped in an `h()` method to prevent *cross-site scripting hacks* (XSS). That is, unless you have some clever way of validating your input.

The named route will be the same as the plain route—except that we replace "connect" with the name we want to give the route:

```
map.auction "auctions/:id",
  :controller => "auctions",
  :action     => "show"
```

In the view, we can now use the more compact version of `link_to`; and we'll get (for auction 3, say) this URL in the hyperlink:

```
http://localhost:3000/auctions/show/3
```

Argument Sugar

In fact, we can make the argument to `auction_path` even shorter. If you need to supply an id number as an argument to a named route, you can just supply the number, without spelling out the `:id` key:

```
<%= link_to "Auction for #{h(auction.item.description)}",
      auction_path(auction.id) %>
```

And the syntactic sugar goes even further: You can just provide objects and Rails will grab the id automatically.

```
<%= link_to "Auction for #{h(auction.item.description)}",
      auction_path(auction) %>
```

This principle extends to other wildcards in the pattern string of the named route. For example, if you've got a route like this:

```
map.item 'auction/:auction_id/item/:id',
  :controller => "items",
  :action     => "show"
```

you'd be able to call it like this:

```
<%= link to item.description, item_path(@auction, item) %>
```

and you'd get something like this as your path (depending on the exact id numbers):

```
/auction/5/item/11
```

Here, we're letting Rails infer the ids of both an auction object and an item object. As long as you provide the arguments in the order in which their ids occur in the route's pattern string, the correct values will be dropped into place in the generated path.

A Little More Sugar with Your Sugar?

Furthermore, it doesn't have to be the id value that the route generator inserts into the URL. You can override that value by defining a `to_param` method in your model.

Let's say you want the description of an item to appear in the URL for the auction on that item. In the `item.rb` model file, you would override `to_params`; here, we'll override it so that it provides a "munged" (stripped of punctuation and joined with hyphens) version of the description:

```
def to_param
 description.gsub(/\s/, "-").gsub([^\W-], '').downcase
end
```

Subsequently, the method call `item_path(@item)` will produce something like this:

```
/auction/3/item/cello-bow
```

Of course, if you're putting things like "cello-bow" in a path field called `:id`, you will need to make provisions to dig the object out again. Blog applications that use this technique to create "slugs" for use in permanent links often have a separate database column to store the "munged" version of the title that serves as part of the path. That way, it's possible to do something like

```
Item.find_by_munged_description(params[:id])
```

to unearth the right item. (And yes, you can call it something other than `:id` in the route to make it clearer!)

> **Courtenay Says...**
>
> Why shouldn't you use numeric IDs in your URLs?
>
> First , your competitors can see just how many auctions you create. Numeric consecutive IDs also allow people to write automated spiders to steal your content. It's a window into your database. And finally, words in URLs just look better.

The Special Scope Method `with_options`

Sometimes you might want to create a bundle of named routes, all of which pertain to the same controller. You can achieve this kind of batch creation of named routes via the `with_options` mapping method.

Let's say you've got the following named routes:

```
map.help '/help', :controller => "main", :action => "help"
map.contact '/contact', :controller => "main", :action => "contact"
map.about '/about', :controller => "main", :action => "about"
```

You can consolidate these three named routes like this:

```
map.with_options :controller => "main" do |main|
  main.help '/help', :action => "help"
  main.contact '/contact', :action => "contact"
  main.about '/about', :action => "about"
end
```

The three inner calls create named routes that are scoped—constrained—to use `"main"` as the value for the `:controller` parameter, so you don't have to write it three times.

Note that those inner calls use `main`, not `map`, as their receiver. After the scope is set, `map` calls upon the nested mapper object, `main`, to do the heavy lifting.

Courtenay Says...

The advanced Rails programmer, when benchmarking an application under load, will notice that routing, route recognition, and the `url_for`, `link_to` and related helpers are often the slowest part of the request cycle. (Note: This doesn't become an issue until you are at least into the thousands of pageviews per hour, so you can stop prematurely optimizing now.)

Route recognition is slow because everything stops while a route is calculated. The more routes you have, the slower it will be. Some projects have hundreds of custom routes.

Generating URLs is slow because there are often many occurances of `link_to` in a page, and it all adds up.

What does this mean for the developer? One of the first things to do when your application starts creaking and groaning under heavy loads (lucky you!) is to cache those generated URLs or replace them with text. It's only milliseconds, but it all adds up.

Conclusion

The first half of the chapter helped you to fully understand generic routing based on `map.connect` rules and how the routing system has two purposes:

- Recognizing incoming requests and mapping them to a corresponding controller action, along with any additional variable receptors
- Recognizing URL parameters in methods such as `link_to` and matching them up to a corresponding route so that proper HTML links can be generated

We built on our knowledge of generic routing by covering some advanced techniques such as using regular expressions and globbing in our route definitions.

Finally, before moving on, you should make sure that you understand how named routes work and why they make your life easier as a developer by allowing you to write more concise view code. As you'll see in the next chapter, when we start defining batches of related named routes, we're on the cusp of delving into REST.

References

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

2. Zed Shaw, author of the Mongrel web server and expert in all matters HTTP-related, was not able to give me a conclusive answer, which should tell you something. (About the looseness of HTTP that is, not Zed.)