

Foreword

Rails is more than programming framework for creating web applications. It's also a framework for thinking about web applications. It ships not as a blank slate equally tolerant of every kind of expression. On the contrary, it trades that flexibility for the convenience of “what most people need most of the time to do most things.” It's a designer straightjacket that sets you free from focusing on the things that just don't matter and focuses your attention on the stuff that does.

To be able to accept that trade, you need to understand not just how to do something in Rails, but also why it's done like that. Only by understanding the why will you be able to consistently work with the framework instead of against it. It doesn't mean that you'll always have to agree with a certain choice, but you will need to agree to the overachieving principle of conventions. You have to learn to relax and let go of your attachment to personal idiosyncrasies when the productivity rewards are right.

This book can help you do just that. Not only does it serve as a guide in your exploration of the features in Rails, it also gives you a window into the mind and soul of Rails. Why we've chosen to do things the way we do them, why we frown on certain widespread approaches. It even goes so far as to include the discussions and stories of how we got there—straight from the community participants that helped shape them.

Learning how to do Hello World in Rails has always been easy to do on your own, but getting to know and appreciate the gestalt of Rails, less so. I applaud Obie for trying to help you on this journey. Enjoy it.

—David Heinemeier Hansson
Creator of Ruby on Rails

Introduction

In late 2004, I was consulting at one of the big American auto makers, alongside a good friend of mine, Aslak Hellesoy.¹ It was a challenging assignment, chock full of difficult political situations, technical frustration, and crushing deadlines. Not your ordinary deadlines either; they were the type of deadline where the client would get fined a million dollars a day if we were late. The pressure was on!

In a moment of questionable judgment, the team agreed to base our continuous integration system on a pet project of Aslak's named `DamageControl`. It was a Ruby-based version of the venerable `CruiseControl` server produced by our employer, ThoughtWorks.

The problem was that `DamageControl` wasn't quite what you'd call a finished product. And like many other Ruby-related things, it just didn't work very well on Windows. Yet for some reason I can't quite remember today, we had to deploy it on an old Windows 2000 server that also hosted the StarTeam source repository (yikes!).

Aslak needed help—over the course of several weeks we pair-programmed extensively on both the application code of `DamageControl` and C-based internals of the Win32 process libraries for Ruby. At the time I had eight years of serious enterprise Java programming experience under my belt and a deep love of the brilliant IntelliJ IDE. I really cannot convey how much I hated Ruby at that point in my career.

So what changed? Well, for starters I eventually made it out of that stressful assignment alive, and took on a relatively easy assignment overseas out of the London office of ThoughtWorks. Within a month or so, Ruby caught my attention again, this time via considerable blogosphere excitement about an up-and-coming web framework named Ruby on Rails. I decided to give Ruby another chance. Perhaps it wasn't so bad after all? I quickly built an innovative social networking system for internal use at ThoughtWorks.

That first Rails experience, over the course of a few weeks in February 2005, was life-altering. All of the best practices I had learned over the years about building web apps had been distilled into a single framework, written in some of the most elegant and concise code

that I had ever seen in my life. My interest in Java died a sudden death (although it took me almost another year to stop using IntelliJ). I began avidly blogging about Ruby and Rails and evangelizing it heavily both inside and out of ThoughtWorks. The rest, as they say, is history.

As I write this in 2007, the Rails business I pioneered at ThoughtWorks accounts for almost half of their global revenue, and they've established a large product division churning out Ruby-based commercial software. Among them is CruiseControl.rb, which I suspect is what Aslak wanted to build all along—it has the honor of being the official continuous integration server of the Ruby on Rails core team.

Ruby and Rails

Why do experienced enterprise folks like me fall in love with Ruby and Rails? Given a set of requirements to fulfill, the complexity of solutions created using Java and Microsoft technology is simply unacceptable. Excess complexity overwhelms individual understanding of the project and dramatically increases communications overhead for the team. The emphasis on following design patterns, as well as the obsession with performance, wears down the pure joy of application development with those platforms.

There's no peer pressure to do anything in the Rails community. DHH (David Heinemeier Hansson) picked a language that made him happy. Rails was born from code that he felt was beautiful. That kind of set the tone for the Rails community. So much about Rails is subjective. People either "get it" or they don't. But there's no malice from those who do towards those who don't, just gentle encouragement.

—Pat Maddox

Ruby is beautiful. Coding in Ruby is beautiful. Everyone I've known who makes the move into Ruby says they are happier than before. For this reason more than any other, Ruby and Rails are shaking up the status quo, especially in enterprise computing. Prior to getting involved with Rails, I was accustomed to working on projects based on fuzzy requirements bearing no relation to real-world needs. I was tired of mind-boggling arrays of competing frameworks to choose from and integrate, and I was tired of ugly code.

In contrast, Ruby is a beautiful, dynamic, high-level language. Ruby code is easier to read and write because it more closely maps to the problem domains we tackle, in a style that is closer to human language. The enhanced readability yields many benefits, both short-term and long-term, as code moves into production and must be understood by maintenance programmers.

My experience has shown me that programs written in Ruby have fewer lines of code than comparable programs in Java and C#. Smaller codebases are easier to maintain and long-term maintenance is widely cited as the biggest cost of successful software projects. Smaller codebases are also faster to debug when things go wrong, even without fancy debugging tools.

The Rise of Rails and Mainstream Acceptance

In ways similar to the Agile movement that helped birth it, Rails is all about catering to our needs as application developers—not as software engineers, and certainly not as computer scientists. By aggressively attacking unneeded complexity, Rails shines brightest in the people-oriented aspects of development that really matter to the ultimate success of our projects. We have fun when we're programming in Rails, and that makes us want to succeed!

The tools and technical infrastructure provided by Rails are comprehensive, encouraging us to focus on delivering business value. Ruby's Principle of Least Surprise is embodied in the simple and elegant design of the Rails. Best of all, Rails is completely free open-source software, which means that when all else fails, browsing the source code can yield answers to even the most difficult of problems.

David has occasionally mentioned that he is not particularly excited about Rails reaching mainstream acceptance, because the competitive edge enjoyed by early adopters would be diminished. Those early adopters have primarily been individuals and small groups of web designers and programmers, with legions of them coming out of the PHP world.

Enterprise Adoption

Call me an idealist if you like, but I believe that even enterprise developers at large and conservative corporations will act to become more effective and innovative at their jobs if they are given the tools and encouragement to do so. That's why it seems like they're jumping on the Rails bandwagon in ever-greater numbers with every year that passes.

Perhaps enterprise developers will ultimately be the most vocal and enthusiastic adopters of Ruby and Rails, because right now they are the ones who as a group stand to lose the most from the status quo. They're consistently the targets of mass layoffs and misguided outsourcing efforts, based on assumptions such as "specification is more important than implementation" and "implementation should be mechanical and trivial."

Is specification actually more important than the implementation? Not for most projects. Is implementation of all but the simplest projects trivial? Of course not! There are significant underlying reasons for the difficulties of software development, especially in enterprise environments:²

- Hard-to-understand legacy systems.
- Highly complex business domains such as investment banking.
- Stakeholders and business analysts who don't actually know what they want.
- Managers resistant to productivity because it shrinks their yearly budgets.
- End users who actively sabotage your project.
- Politics! Sticking your head out means worrying that it'll get chopped off.

As a consultant to Fortune 1000 companies, I lived and breathed those situations on an everyday basis for almost 10 years, eventually stumbling upon a powerful concept. There is a viable alternative to playing it safe, an alternative so powerful that it transcends politics and is guaranteed to bring you acclaim and open new doors of opportunity.

That alternative is being exceptional! It starts with productivity. I'm talking about becoming so obviously effective at your job that nobody will ever be able to scapegoat you, to the extent that it would be political suicide to try. I'm talking about cultivating practices that make your results stand out so brilliantly that they bring tears of joy to even the most cynical and hardened stakeholders of your projects. I mean regularly having time to polish your applications to a state of wonderfulness that consistently breeds passionate end users.

By simply being exceptional, you can be that individual (or team) that keeps clients happy and paying their invoices on time, or that survives layoffs year after year, because the decision-makers say: "Oh, there's no way we can afford to lose them."

Let me pause for a second. I wouldn't blame you for regarding my words with skepticism, but none of what I'm saying is idle hype. I'm describing my own life since moving to Ruby on Rails. This book is intended to help you make Ruby on Rails your secret (or not-so-secret) weapon for thriving in the treacherous world of software development.

Delivering Results

My contributors and I draw on our collective experience and industry knowledge to show you how to deliver practical results using Ruby on Rails on your projects, giving you the ammunition needed to justify your choice of technology and even defeat objections that will

undoubtedly come your way. Since we know there are never any silver bullets, we'll also warn you about situations where choosing Rails would be a mistake.

Along the way, we'll analyze each of the components of Rails in depth and discuss how to extend them when the need arises. Ruby is an extremely flexible language, which means there are myriad ways to customize the behavior of Rails yourself. As you will learn, the Ruby way is all about giving you the freedom to find the optimal solution to the problem at hand.

As a reference work, this book functions as a guide to the Rails API and the wealth of Ruby idioms, design approaches, libraries, and plugins useful to the Ruby on Rails enterprise developer.

About Opinionated Software

Before going on, I should mention that part of what makes Rails exceptional is that it is opinionated software, written by opinionated programmers. Likewise, this is an opinionated book, written by opinionated writers.

Here are some of the opinions about development that influence this book. You don't have to agree with all of them—just be aware of their influence:

- Developer motivation and productivity trump all other factors for project success.
- The best way to keep motivated and productive is to focus on delivering business value.
- Performance means “executing as fast as possible, on a given set of resources.”
- Scalability means “executing as fast as needed, on as many resources as needed.”
- Performance is irrelevant if you can't scale.
- If you can scale cheaply, milking every ounce of performance from your processors should never be your first priority.
- Linking scalability to choice of development tools is a pervasive mistake in the industry and most software does not have extreme scalability requirements.
- Performance *is* related to choice of language and tools because higher-level languages are easier to write and understand. There is wide consensus that the performance problems in most applications are caused by poorly written application code.
- Convention over configuration is a better way to write software. Huge XML configuration files must be eliminated!
- Code portability, the ability to take code and run it on a different hardware platform, is not particularly important.

- It's better to solve a problem *well* even if the solution only runs on one platform. Portability is irrelevant if your project fails.
- Database portability, the ability to run the same code on different relational database systems is rarely important and is almost never achieved.
- Presentation is very important, even for small projects. If your application looks bad, everyone will assume it is written badly.
- Allowing technology to dictate the approach to solving a business problem is usually a bad idea; however, that advice shouldn't be used as an excuse to stick with inferior technology.
- The benefits of generalized application components are dubious. Individual projects usually have very particular business needs and wildly different infrastructure requirements, making parameterized reuse very difficult to achieve in practice.

Phew, that's a lot of opinions. But don't worry, *The Rails Way* is primarily a reference work, and this list is the only one of its kind in the book. Speaking of which....

About This Book

This book is not a tutorial or basic introduction to Ruby or Rails. It is meant as a day-to-day reference for the full-time Rails developer. At times we delve deep into the Rails codebase to illustrate why Rails behaves the way that it does, and present snippets of actual Rails code. The more confident reader might be able to get started in Rails using just this book, extensive online resources, and his wits, but there are other publications that are more introductory in nature and might be a wee bit more appropriate for beginners.

I am a fulltime Rails application developer and so is every contributor to this book. We do not spend our days writing books or training other people, although that is certainly something that we enjoy doing on the side.

I started writing this book mostly for myself, because I hate having to use online documentation, especially API docs, which need to be consulted over and over again. Since the API documentation is liberally licensed (just like the rest of Rails), there are a few sections of the book that reproduce parts of the API documentation. In practically all cases, the API documentation has been expanded and/or corrected, supplemented with additional examples and commentary drawn from practical experience.

Hopefully you are like me—I really like books that I can keep next to my keyboard, scribble notes in, and fill with bookmarks and dog-ears. When I’m coding, I want to be able to quickly refer to both API documentation, in-depth explanations, and relevant examples.

Book Structure

I attempted to give the material a natural structure while meeting the goal of being the best-possible Rails reference book. To that end, careful attention has been given to presenting holistic explanations of each subsystem of Rails, including detailed API information where appropriate. Every chapter is slightly different in scope, and I suspect that Rails is now too big a topic to cover the whole thing in depth in just one book.

Believe me, it has not been easy coming up with a structure that makes perfect sense for everyone. Particularly, I have noted surprise in some readers when they notice that `ActiveRecord` is not covered first. Rails is foremost a web framework and at least to me, the controller and routing implementation is the most unique, powerful, and effective feature, with `ActiveRecord` following a close second.

Therefore, the flow of the book is as follows:

- The Rails environment, initialization, configuration, and logging
- The Rails dispatcher, controllers, rendering, and routing
- REST, Resources, and Rails
- `ActiveRecord` basics, associations, validation, and advanced techniques
- `ActionView` templating, caching, and helpers
- Ajax, Prototype, and Scriptaculous JavaScript libraries, and RJS
- Session management, login, and authentication
- XML and `ActiveResource`
- Background processing and `ActionMailer`
- Testing and specs (including coverage of RSpec on Rails and Selenium)
- Installing, managing, and writing your own plugins
- Rails production deployment, configurations, and Capistrano

Sample Code and Listings

The domains chosen for the code samples should be familiar to almost all professional developers. They include time and expense tracking, regional data management, and blogging applications. I don't spend pages explaining the subtler nuances of the business logic for the samples or justify design decisions that don't have a direct relationship to the topic at hand. Following in the footsteps of my series colleague Hal Fulton and *The Ruby Way*, most of the snippets are not full code listings—only the relevant code is shown. Ellipses (...) denote parts of the code that have been eliminated for clarity.

Whenever a code listing is large and significant, and I suspect that you might want to use it verbatim in your own code, I supply a listing heading. There are not too many of those. The whole set of code listings will not add up to a complete working system, nor are there 30 pages of sample application code in an appendix. The code listings should serve as inspiration for your production-ready work, but keep in mind that it often lacks touches necessary in real-world work. For example, examples of controller code are often missing pagination and access control logic, because it would detract from the point being expressed.

Plugins

Whenever you find yourself writing code that feels like *plumbing*, by which I mean completely unrelated to the business domain of your application, you're probably doing too much work. I hope that you have this book at your side when you encounter that feeling. There is almost always some new part of the Rails API or a third-party plugin for doing exactly what you are trying to do.

As a matter of fact, part of what sets this book apart is that I never hesitate in calling out the availability of third-party plugins, and I even document the ones that I feel are most crucial for effective Rails work. In cases where a plugin is better than the built-in Rails functionality, we don't cover the built-in Rails functionality (pagination is an example).

An average developer might see his productivity double with Rails, but I've seen serious Rails developers achieve gains that are much, much higher. That's because we follow the Don't Repeat Yourself (DRY) principle religiously, of which Don't Reinvent The Wheel (DRTW) is a close corollary. Reimplementing something when an existing implementation is *good enough* is an unnecessary waste of time that nevertheless can be very tempting, since it's such a joy to program in Ruby.

Ruby on Rails is actually a vast ecosystem of core code, official plugins, and third-party plugins. That ecosystem has been exploding rapidly and provides all the raw technology you need to build even the most complicated enterprise-class web applications. My goal is to

equip you with enough knowledge that you'll be able to avoid continuously reinventing the wheel.

Recommended Reading and Resources

Readers may find it useful to read this book while referring to some of the excellent reference titles listed in this section.

Most Ruby programmers always have their copy of the “Pickaxe” book nearby, *Programming Ruby* (ISBN: 0-9745140-5-5), because it is a good language reference. Readers interested in really understanding all of the nuances of Ruby programming should acquire *The Ruby Way, Second Edition* (ISBN: 0-6723288-4-4).

I highly recommend *Peepcode Screencasts*, in-depth video presentations on a variety of Rails subjects by the inimitable Geoffrey Grosenbach, available at <http://peepcode.com>.

Regarding David Heinemeier Hansson a.k.a. DHH

I had the pleasure of establishing a friendship with David Heinemeier Hansson, creator of Rails, in early 2005, before Rails hit the mainstream and he became an *International Web 2.0 Superstar*. My friendship with David is a big factor in why I'm writing this book today. David's opinions and public statements shape the Rails world, which means he gets quoted a lot when we discuss the nature of Rails and how to use it effectively.

David has told me on a couple of occasions that he hates the “DHH” moniker that people tend to use instead of his long and difficult-to-spell full name. For that reason, in this book I try to always refer to him as “David” instead of the ever-tempting “DHH.” When you encounter references to “David” without further qualification, I'm referring to the one-and-only David Heinemeier Hansson.

Rails is by and large still a small community, and in some cases I reference core team members and Rails celebrities by name. A perfect example is the prodigious core-team member, Rick Olson, whose many useful plugins had me mentioning him over and over again throughout the text.

Goals

As stated, I hope to make this your primary working reference for Ruby on Rails. I don't really expect too many people to read it through end to end unless they're expanding their basic knowledge of the Rails framework. Whatever the case may be, over time I hope this book gives you as an application developer/programmer greater confidence in making design and implementation decisions while working on your day-to-day tasks. After spending time with

this book, your understanding of the fundamental concepts of Rails coupled with hands-on experience should leave you feeling comfortable working on real-world Rails projects, with real-world demands.

If you are in an architectural or development lead role, this book is not targeted to you, but should make you feel more comfortable discussing the pros and cons of Ruby on Rails adoption and ways to extend Rails to meet the particular needs of the project under your direction.

Finally, if you are a development manager, you should find the practical perspective of the book and our coverage of testing and tools especially interesting, and hopefully get some insight into why your developers are so excited about Ruby and Rails.

Prerequisites

The reader is assumed to have the following knowledge:

- Basic Ruby syntax and language constructs such as blocks
- Solid grasp of object-oriented principles and design patterns
- Basic understanding of relational databases and SQL
- Familiarity with how Rails applications are laid out and function
- Basic understanding of network protocols such as HTTP and SMTP
- Basic understanding of XML documents and web services
- Familiarity with transactional concepts such as ACID properties

As noted in the section “Book Structure,” this book does not progress from easy material in the front to harder material in the back. Some chapters do start out with fundamental, almost introductory material, and push on to more advanced coverage. There are definitely sections of the text that experienced Rails developer will gloss over. However, I believe that there is new knowledge and inspiration in every chapter, for all skill levels.

Required Technology

A late-model Apple *MacBookPro* with 4GB RAM, running OSX 10.4. Just kidding, of course. Linux is pretty good for Rails development also. Microsoft Windows—well, let me just put it this way—your mileage may vary. I’m being nice and diplomatic in saying that. We specifically *do not* discuss Rails development on Microsoft platforms in this book.³ To my knowledge, most working Rails professionals develop and deploy on non-Microsoft platforms.

References

1. Aslak is a well-known guy in Java open-source circles, primarily for writing XDoclet.
2. I’m not saying startups are much easier, but they usually have less dramatic problems.
3. For that information, try the Softies on Rails blog at <http://softiesonrails.com>.

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 labyrinthine 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 <code>id</code> |
|-------------------------------|-----------------------|------------------------------|--------------------------------------|
| | Controller | Action | |
| <code>/auctions/show/3</code> | <code>auctions</code> | <code>show</code> | <code>3</code> |
| <code>/auctions/index</code> | <code>auctions</code> | <code>index</code> | <code>nil</code> |
| <code>/auctions</code> | <code>auctions</code> | <code>index (default)</code> | <code>nil</code> |
| <code>/auctions/show</code> | <code>auctions</code> | <code>show</code> | <code>nil</code> —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.

Globbering 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 URL². 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 occurrences 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.)

Index

Symbols & Numerics

(pound sign) delimiter, 310
- (minus sign) delimiter, 310

307 redirects, 42

A

abstract base model classes, 291-292
acceptance tests, 588-589
AccountController class, Acts_as_Authenticated
 method, 496-498
Action Cache plugin, 337-338
action caching, 328-330
ActionMailer framework, setting up, 531
actions (Selenium), 589
ActionView, 308
Active Record framework, 140-142, 158
 attributes
 default values, 160-162
 serialized, 162
 configuring, 196-197
 CRUD, 163-178
 database connections, 189-196
 database locking, 179
 optimistic locking, 180-182
 pessimistic locking, 182

 database schema, naming conventions, 159
 finders
 conditions, 183-185
 options, 187-189
 parameters, 186
 results, ordering, 186
 macro-style methods, 155
 convention over configuration, 156
 pluralization, 157-158
 relationship declarations, 155-156
 Migration API, 146-148
 migrations, 142-145
 columns, defining, 149-155
 query cache, 172-174
ActiveRecord
 abstract base model classes, 291-292
 callback registration, 272-273
 before after/callbacks, 274
 callbacks, 271
 after_find callback, 278-279
 after_initialize callback, 278
 classes, 279-282
 halting execution, 274
 usage examples, 275-278
 classes, modifying at runtime, 301-304
 common behavior, reusing, 296-301
 observers, 282
 naming conventions, 283
 registration, 283

- polymorphic has_many relationships, 292-296
- STI, 284-291
 - inheritance, mapping to database, 286-287
- ActiveRecord models, creating forms, 362-369
- ActiveRecord SessionStore, 473-474
- ActiveRecordHelper module, 342-345, 347-348
- ActiveResource, 519
 - Create method, 522-523
 - customizing, 527-528
 - Delete method, 524-525
 - Find methods, 519-521
 - headers, 526
 - Update method, 524
- ActiveSupport library, 726-739, 759
 - Array class, 724
 - Class object, 729-730
 - default log formatter, 764-765
 - dependencies, 740
 - module attributes, 740-741
 - public instance methods, 742-743
 - Deprecation module, 744-745
 - Enumerable module, 747-748
 - Exception module, 748
 - FalseClass module, 749
 - File module, 749
 - Hash module, 750-757
 - Inflections class, 757-759
 - Module class, 765-769
 - NilClass, 772-773
 - Numeric class, 773-776
 - Object class, 776-777, 779
 - Proc class, 781-782
 - Range class, 782
 - String class, 783-790
 - Symbol class, 790
 - Time class, 792-798
- active_record_defaults plugin, 807
- acts-as-authenticated plugin
 - AccountController class, 496-498
 - authenticate method, 494
 - before-save callback, 493
 - current user, 499-501
 - installing, 486-487
 - login from cookie, 498-499
 - remember token, 495
- User model, 487-492
 - validation rules, 492
- Adam, James, 650
- adding
 - classes to mocks folder, 548-549
 - load paths, 10
 - plugin sources, 630
- after_find callback, 278-279
- after_initialize callback, 278
- Ajax, Prototype, 420-421
 - Class object, 424
 - Enumerable object, 437-443
 - Firebug, 421
 - Hash class, 443
 - Object class, 425-426
 - Prototype object, 445
 - Responders object, 437
 - top-level functions, 422-424
- alert() method (RJS), 457
- aliases, 806
- analyzing log files, 22-23
- ante-default route, 69-70
 - respond to method, 70
- APIs
 - integration test API, 585-586
 - Migration, 146-148
 - Prototype, 421
 - Class object, 424
 - Object class, 425-426
 - top-level functions, 422-424
 - RSelenium, 592
 - partial test cases, 592-593
- application console
 - route objects, 129-131
 - routes
 - dumping, 128-129
 - manual recognition and generation, 132-134
 - named routes, executing, 134-135
- application.rhtml layout template, 312-313
- arbitrary predicates, 599-601
- arguments
 - for named routes, 84-85
 - hashes, syntax, 136
- around filters, 48
- Array class

- ActiveSupport library, 724
- JavaScript extensions, 426-428
- to_xml method, 510-513
- assertions, 553, 562-566
 - assert_block, 563
 - for functional tests, 572-576
 - one-assertion guideline, 566-567
- assert_block assertion, 563
- AssetTagHelper module, 348-352
- assigning instance variables to view specs, 622
- associated objects, validating presence of, 261
- AssociationProxy class, 249-250
- associations, 199
 - belongs_to, 207-208
 - options, 209-215
 - class hierarchy, 199-201
 - extensions, 247-248
 - has_many, 215
 - options, 216-224
 - proxy methods, 224-225
 - many-to-many relationships, 225
 - has_and_belongs_to_many method, 225-232
 - through association, 233-241
 - objects, adding to collection, 203-207
 - one-to-many relationships, 201-203
 - one-to-one relationships, 241
 - has_one relationship, 241-246
 - unsaved, 246
- Astels, Dave, 566
- attribute-based finders, 170
- attributes
 - default values, 160-162
 - serialized, 162
- attributes method, 168
- authenticate method, acts as authenticated, 494
- AuthenticatedTestHelper module, 501-502
- auto-loading classes and modules, 8
- autocompleters, 466-467
- automatic class reloading, 15
- Autotest project, 624
- auto_discovery_link_tag method, 348
- auto_link method, 393

B

- BackgounDRb, adding background processing
 - to applications, 713-718
- background processing
 - with BackgounDRb, 713-718
 - with daemons, 719-722
 - with DRb, 710-712
 - with script/runner, 708-710
- Bates, Ryan, 810
- BDD (Behavior-Driven Development), 611
- before save callback, Acts as Authenticated, 493
- before/after callbacks, 274
- behaviors, 598, 603-604
 - shared behaviors, 604-607
- belongs_to association, 207-208
 - options, 209-215
- benchmark method, 353
- BenchmarkHelper module, 353
- bidirectional relationships, 227-228
- Bilkovich, Wilson, 249
- BNL (Business Natural Languages), 303
- boot script, initializer.rb file, 6
- bootstrapping, 3
 - auto-loading classes and modules, 8
 - initializer.rb file, 6
 - default load paths, 6-7
 - mode override, 2
 - Rails gem version, 2
 - RubyGems, 5
- bound parameters for routes, 60-61
- breadcrumbs help method, writing, 409-410
- Buck, Jamis, 108, 131, 808
- Builder API, 513-515
- building production stack, required components, 655
 - application tier, 656
 - database tier, 656
 - monitoring tools, 657
 - web tier, 656
- builtins, 8
- button_to method, 400

- Monit, 667-669
 - init script, 673-675
 - Nginx, 663-667
 - init script, 670-672
 - SMTP servers, 543
 - constants, JSON, 760
 - content_tag_for method, 390
 - controller-only resources, 113-115
 - controllers, 28-31
 - filters, 43-44
 - around filters, 48
 - conditions, 50
 - external filter classes, 46
 - filter chain halting, 50
 - filter chain ordering, 47-48
 - filter chain skipping, 49
 - halting the filter chain, 46
 - inheritance, 44-46
 - inline filter method, 47
 - functional testing, 570-571
 - assertions, 572-576
 - methods, 571
 - instance variables, 42-43
 - namespaced controllers, 107
 - post-backs, 112
 - requests, redirecting, 39-42
 - REST actions, 96-98
 - explicitly specifying, 105
 - session class method, 471
 - specs, 617-619
 - streaming, 51
 - send data() method, 51-52
 - send file() method, 52-55
 - templates, rendering, 33-36
 - view templates, 31
 - convention over configuration, 32
 - converting
 - numeric data to formatted strings,
 - NumberHelper module, 383-385
 - vendor libraries to Piston, 638
 - XML to Ruby hashes, 515-516
 - cookies, 481
 - reading, 482
 - writing, 482
 - CookieStore session storage, 476-478
 - create actions, 123-124
 - Create method (ActiveResource), 522-523
 - creating
 - mailer model, 532-533
 - migrations, 143-144
 - named routes, 81-83
 - route helper methods, 82
 - via with options mapping method, 86-87
 - sortable lists, 465-466
 - CRUD (Create Read Update Delete),
 - 92, 168-172
 - creating, 163
 - deleting, 178
 - reading, 164-168
 - updating, 174-178
 - CSV fixtures, 555
 - current user, Acts as Authenticated, 499-501
 - current user id, storing in session hash, 470
 - current_page? method, 401
 - custom actions (REST), syntax, 112-113
 - custom expectation matchers, 601-603
 - custom parameters, receptors, 73-74
 - custom routes, writing, 72
 - custom SQL queries, 171-172
 - custom validation, 268-269
 - customizing
 - ActiveResource, 527-528
 - environments, 18
 - SCM System (Capistrano), 692
 - to_xml method output, 505-506
 - validation error behavior, 347-348
 - cycle method, 394
- ## D
- daemons, adding background processing to
 - applications, 719-722
 - database locking, 179
 - optimistic locking, 180-182
 - pessimistic locking, 182
 - database schema
 - Active Record framework, 140-142, 158
 - attributes, 160-162
 - configuring, 196-197
 - connections, 189-196

- CRUD, 163-178
- database locking, 179-182
- finders, 183-189
- macro-style methods, 155-158
- Migration API, 146-148
- migrations, 142-145, 149-155
- naming conventions, 159
- query cache, 172-174
- database.yml, storing, 693-695
- DateHelper module, 355-361
- Davis, Ryan, 624
- DebugHelper module, 361
- debug_view_helper plugin, 808
- deep nesting, 108-109
- default load paths (initializer.rb file), 6-7
- default log formatter, 764-765
- default routes (routes.rb file), 65-66
 - ante-default route, 69-70
 - resond to method, 70
- generating, 67-68
- id field, 66, 77-78
- modifying, 68
- delay() method (RJS), 458
- Delete method (ActiveResource), 524-525
- DELETE requests, handling with REST, 98
- delimiters, 308, 310
 - blank lines, removing, 310
- deploying
 - Capistrano, 687-688, 691
 - multiserver deployments, 702-703
 - via :copy, 692
 - production environment
 - Capistrano installation procedures, 661
 - Mongrel installation procedures, 659
 - Monit installation procedures, 661
 - MySQL installation procedures, 660
 - Nginx installation procedures, 659
 - predeployment concerns, 676-677
 - Ruby installation procedures, 658
 - RubyGems installation procedures, 658
 - Subversion installation procedures, 660
- deployment machine (Capistrano), setting up, 689-690
- destroy actions, 121-122
- destroy method, 179

- development mode, 14
 - automatic class reloading, 15
 - caching, 328
 - class loader, 15-16
- development mode:fixtures, 557
- disabling sessions for robots, 472
- discover command, 631
- dispatcher, 29-31
- distance_of_time_in_words method, 360
- div_for method, 391
- dom_class method, 389
- dom_id method, 389
- draggable() method (RJS), 458
- dragging and dropping, 463-464
- DRb (Distributed Ruby)
 - adding background processing to applications, 710-712
 - session storage, 475
- drop_receiving() method (RJS), 458
- DSL (Domain-Specific Language), 155
- dumping
 - routes, 128-129
 - schema, 11
- dynamic fixture content, 556-557

E

- edge rails, 5, 805-806
 - applications, freezing/unfreezing, 4
- edit actions, 124-125
- edit/update operations (REST), 99-100
- email
 - file attachments
 - receiving, 543
 - sending, 539-540
 - receiving, 541-543
 - sending, 540
 - SMTP servers, configuring, 543
 - TMail methods, 542
- empty routes, 71
- enabling sessions, 473
- end_form_tag method, 378
- enforcing uniqueness of join models, 262
- Enumerable object (prototype), 437-443

- environment.rb file
 - log-level, overriding, 10
 - overriding, 9-11
 - production mode, 2-4
 - TZ environment variable, 12-13
- environments, customizing, 18
- ERb (Embedded Ruby), 308
 - delimiters, 310
- erb command, 308
- errors, 266
 - conditional validation, 264
 - when to use, 265-266
 - custom validation, 268-269
 - Errors collection, manipulating, 267
 - finding, 254
 - skipping validation, 269-270
 - validation methods
 - allow_nil option, 263
 - message option, 263
 - on option, 264
 - RecordInvalid, 263
 - validates_acceptance_of, 254
 - validates_associated, 255-256
 - validates_confirmation_of, 256
 - validates_each, 256
 - validates_exclusion_of, 257
 - validates_existence_of, 257
 - validates_format_of, 258-259
 - validates_inclusion_of, 257
 - validates_length_of, 259-260
 - validates_numericality_of, 260
 - validates_presence_of, 261
 - validates_uniqueness_of, 261-262
- error_messages_for method, 342-343
- error_message_on method, 342-343
- Event class (JavaScript), extensions, 428-430
- examples of callback usage, 275-278
- exception_logger plugin, 808
- exception_notification plugin, 808
- excerpt method, 395
- executing, named routes in application console, 134-135
- expectations, 599
 - custom expectation matchers, 601-603

- expiring cached content, 333-334
 - fragments, 334
 - Sweeper class, 335-336
- explicitly specifying RESTful controllers, 105
- extending classes, 644
- extensions to associations, 247-248
- external filter classes, 46
- externals, 635

F

- faux accessors, 369-370
- Fielding, Roy T., 89
- Fields, Jay, 566
- file attachments
 - receiving, 543
 - sending, 539-540
- file_field_tag method, 378
- filters, 43-44
 - around filters, 48
 - conditions, 50
 - external filter classes, 46
 - filter chain halting, 50
 - filter chain ordering, 47-48
 - filter chain skipping, 49
 - halting the filter chain, 46
 - inheritance, 44-46
 - inline filter method, 47
- Find methods (ActiveResource), 519-521
- finders
 - attribute-based, 170
 - conditions, 183-185
 - options, 187-189
 - parameters, 186
 - results, ordering, 186
- finding errors, 254
- FireBug, 421
- fixtures, 554
 - accessing from tests, 556
 - CSV fixtures, 555
 - disadvantages of, 560-561
 - dynamic content, 556-557
 - generating from development data, 558-559
 - in development mode, 557
 - options, 559

- forcing plugin reinstallation, 633
- form method, 344-345
- form tags (HTML), generating, 378-381
- formatted named routes, 117
- FormHelper module, 362-370
 - faux accessors, 370
- FormOptionsHelper module, 371-377
- FormTagHelper module, 378-381
- form_tag method, 379
- Fowler, Martin, 139
- fragment caching, 330
 - global fragments, 332-333
 - named fragments, 331-332
 - storage, 338-340
- frameworks, skipping, 9
- freezing/unfreezing applications, 4
- Function class (JavaScript), extensions, 430-431
- functional tests, 570-571, 576-580
 - assertions, 572-576
 - equality parameter, 580-581
 - methods, 571
 - RJS behavior, testing, 581-582
 - routing rules, testing, 582-584
 - selection methods, 582
- functions (prototype), 422-424

G

- gateway capability (Capistrano), 705
- gems
 - color gem, 807
 - Rails versions, 2
- generating
 - fixtures from development data, 558-559
 - HTML tags, 378-381
 - TagHelper module, 391-392
 - routes, 132-134
- generating default route, 67-68
- generators, 614
- globs
 - key value pairs, globbing, 80-81
 - route globbing, 79-80
- green bars, 553
- Grosenbach, Geoffrey, 806

H

- h method, 319
- halting
 - callback execution, 274
 - filter chain, 46
- Hash class (prototype), 443
- hashes, syntax, 136
- has_and_belongs_to_many method, 225-228
 - columns, adding to join tables, 232
 - custom SQL options, 229-231
- has_finder plugin, 808
- has_many association, 215
 - options, 216-224
 - proxy methods, 224-225
- has_many_polymorphs plugin, 809
- has_one relationship, 241-244
 - options, 244-246
- HEAD element, yielding additional content, 315
- headers (ActiveResource), setting, 526
- Helmkamp, Bryan, 808
- helper methods, 311
 - breadcrumbs, writing, 409-410
 - photo_for, writing, 408
 - stubbing, 623
 - tiles, writing, 410-416
 - title helper, writing, 407-408
- helper modules
 - ActiveRecordHelper, 342-348
 - AssetTagHelper, 348-352
 - BenchmarkHelper, 353
 - CaptureHelper, 354
 - DateHelper, 355-361
 - DebugHelper, 361
 - FormHelper, 362-369
 - faux accessors, 370
 - FormOptionsHelper, 371-377
 - FormTagHelper, 378-381
 - JavaScriptHelper, 381-383
 - NumberHelper, 383, 385
 - PaginationHelper, 386-388
 - PrototypeHelper
 - link_to_remote method, 445-449
 - observe_field method, 451-453

- observe_form method, 453
 - periodically_call_remote method, 451
 - remote_form_for method, 449-451
 - RecordIdentificationHelper, 388-389
 - RecordTagHelper, 390-391
 - TagHelper, 391-392
 - TextHelper, 393-400
 - UrlHelper, 400-406
 - writing, 407
 - helper specs, 623
 - hide() method (RJS), 458
 - highlight method, 395
 - Hodel, Eric, 624
 - hook code, 641
 - HTML
 - adding content to HEAD, 348
 - form tags, generating, 378-381
 - input tags, generating, 378-381
 - messages, sending, 536-537
 - select helper methods, 371-373
 - select tags, creating for calendar data, 355-359
 - TagHelper module, 391-392
 - tags, generating, 391-392
 - HTTP requests, processing using REST, 98
 - HTTP status codes, 37
 - for redirection, 41-42
 - HTTPS, securing sessions, 473
 - Hyett, PJ, 807
- I**
- image_path method, 350
 - image_tag method, 350
 - implicit multipart messages, 539
 - importing vendor libraries into Piston, 637
 - in-place editors, 467
 - index actions, 118-121
 - inflections, 157-158
 - Inflections class (ActiveSupport Library), 757-759
 - Inflector class, 511
 - inheritance, 44-46
 - STI, 284-291
 - mapping to the database, 286-287
 - init scripts, configuring
 - Mongrel, 672-673
 - Monit, 673-675
 - Nginx, 670-672
 - init.rb file, 640
 - initializer.rb file, 6
 - default load paths, 6-7
 - injection attacks, preventing, 319
 - inline filter method, 47
 - inline rendering, 35
 - input method, 346
 - input tags (HTML), generating, 378-381
 - insert_html() method (RJS), 458
 - install command, 632
 - install.rb file, 645-647
 - installing
 - Capistrano, 661, 684
 - Mongrel, 659
 - Mongrel Cluster, 659
 - Monit, 661
 - MySQL, 660
 - Nginx, 659
 - Piston, 636
 - plugins, 629
 - Acts as Authenticated, 486-487
 - Routing Navigator, 136
 - RSpec, 613
 - Ruby, 658
 - RubyGems, 658
 - Selenium on Rails, 591
 - Subversion, 660
 - instance variables, 42-43, 315-319
 - assigning to view specs, 622
 - integration mode (RSpec on Rails), 619
 - integration tests, 550-551, 584-586
 - sessions, 586
 - invalid?() method, 267
 - isolation mode (RSpec on Rails), 619
- J-K**
- JavaScript
 - Array class extensions, 426-428
 - Event class extensions, 428-430
 - Function class extensions, 430-431
 - JSON, 461-462

Number class extensions, 432-433
 RJS, 453-455
 alert() method, 457
 call() method, 457
 delay() method, 458
 draggable() method, 458
 drop_receiving() method, 458
 hide() method, 458
 insert_html() method, 458
 literal() method, 459
 redirect_to() method, 459
 remove() method, 459
 replace() method, 460
 replace_html() method, 460
 select() method, 460
 show() method, 460
 sortable() method, 460
 templates, 455-456
 toggle() method, 461
 visual_effect() method, 461
 String class extensions, 433-436
 JavaScriptHelper module, 381-383
 javascript_include_tag method, 351
 javascript_path method, 352
 join models, enforcing uniqueness, 262
 JSON (JavaScript Object Notation),
 420, 461-462, 760
 class methods, 761-762
 constants, 760
 rendering, 36
 Kernel class
 public instance methods, 762-764
 key-value pairs, globbing, 80-81
 keywords, yield, 313
 Koziarski, Michael, 158

L

lambdas, 415
 layouts, application.rhtml layout template,
 312-313
 legacy naming schemes, 159
 link_to method, 401-402
 link_to_remote method, 445-449

list command, 629-630
 listing available plugins, 629-630
 literal() method (RJS), 459
 load paths, 15
 adding, 10
 class loader, 15-16
 locators (Selenium), 590
 locking, 179
 optimistic locking, 180-182
 pessimistic locking, 182
 locking down plugin versions, 636
 locking Piston revisions, 639
 log files, 20-22
 analyzing, 22-23
 log level, overriding, 10
 logging, 18
 log files, 20-22
 analyzing, 22-23
 severity levels, 19
 Syslog, 24
 logging partials, 327
 login from cookie, Acts as Authenticated,
 498-499

M

macro-style methods, 43, 155
 convention over configuration, 156
 pluralization, 157-158
 relationship declarations, 155-156
 mailer methods, 533
 email
 receiving, 541-543
 sending, 540
 file attachments, sending, 539-540
 HTML messages, sending, 536-537
 multipart messages, sending, 537-539
 options, 534-536
 mailer models, 531
 creating, 532-533
 mail_to method, 404-405
 manipulating Errors collection, 267
 manual route recognition and generation,
 132-134

- many-to-many relationships, 225
 - has_and_belongs_to_many method, 225-228
 - columns, adding to join tables, 232
 - custom SQL options, 229-231
 - through association, 233-238
 - options, 238-241
- markdown method, 396
- Marklund, Peter, 798
- memcache session storage, 475-476
- messages
 - error messages, 266
 - validating, 267
- metaprogramming, 155
- methods
 - assertion methods, 562-565
 - assert_block, 563
 - attributes, 168
 - auto_discovery_link_tag method, 348
 - auto_link, 393
 - benchmark, 353
 - button_to, 400
 - checkbox_tag, 378
 - concat, 393
 - content_tag_for, 390
 - Create, 522-523
 - current_page?, 401
 - cycle, 394
 - delete, 524-525
 - destroy, 179
 - distance_of_time_in_words, 360
 - div_for, 391
 - dom_class, 389
 - dom_id, 389
 - end_form_tag, 378
 - error_messages_for, 342-343
 - error_message_on, 342-343
 - excerpt, 395
 - file_field_tag, 378
 - Find, 519-521
 - for functional tests, 571
 - form, 344-345
 - form_tag, 379
 - h, 319
 - has_and_belongs_to_many method, 225-228
 - helper methods, 311
 - writing, 407-416
 - highlight, 395
 - image_path, 350
 - image_tag, 350
 - inline filter method, 47
 - input, 346
 - javascript_include_tag, 351
 - javascript_path, 352
 - lambdas, 415
 - link_to, 401-402
 - macro-style, 43, 155
 - convention over configuration, 156
 - pluralization, 157-158
 - relationship declarations, 155-156
 - mailer methods, 533
 - email, receiving, 541-543
 - email, sending, 540
 - file attachments, sending, 539-540
 - HTML messages, sending, 536-537
 - multipart messages, sending, 537-539
 - options, 534-536
 - mail_to, 404-405
 - markdown, 396
 - mock_model, 617
 - partial_path, 389
 - pluralize, 396
 - public instance methods, 726
 - reload, 169
 - render
 - :content type option, 37
 - :layout option, 37
 - :status option, 37-39
 - reset_cycle, 397
 - respond_to, 69-70
 - resource representations, 116
 - route helper methods, 82
 - sanitize, 397
 - select helper methods, 371-373
 - send data(), 51-52
 - send file(), 52-55
 - setup methods, 553
 - simple_format, 398
 - strip_links, 398
 - strip_tags, 398
 - stylesheet_link_tag, 352
 - textilize, 399

- textilize_without_paragraph, 399
 - TMail, 542
 - to_xml, 503-504
 - include parameter, 507-508
 - methods parameter, 508-509
 - output, customizing, 505-506
 - overriding, 510
 - procs parameter, 509-510
 - truncate, 399
 - update, 524
 - url_for, 76-77, 405-406
 - validation errors
 - skipping validations, 269-270
 - validation methods
 - :allow_nil option, 263
 - :message option, 263
 - :on option, 264
 - conditional validation, 264-266
 - custom validation, 268-269
 - RecordInvalid, 263
 - validates_acceptance_of, 254
 - validates_associated, 255-256
 - validates_confirmation_of, 256
 - validates_each, 256
 - validates_exclusion_of, 257
 - validates_existence_of, 257
 - validates_format_of, 258-259
 - validates_inclusion_of, 257
 - validates_length_of, 259-260
 - validates_numericality_of, 260
 - validates_presence_of, 261
 - validates_uniqueness_of, 261-262
 - word_wrap, 400
 - methods:invalid?(), 267
 - methods:on(), 267
 - Migration API, 146-148
 - migrations, 142
 - caveats, 145
 - columns, creating, 149-155
 - creating, 143-144
 - naming, 144
 - MIT-LICENSE files, 644
 - Mocha library, 549
 - stubbing, 549-550
 - mocking
 - mock objects, 607-608
 - partial mocking, 609-610
 - mocks folder, adding classes, 548-549
 - mock_model method, 617
 - model classes, associations, 199
 - belongs_to, 207-215
 - class hierarchy, 199-201
 - extensions, 247-248
 - has_many, 215-225
 - many_to_many relationships, 225-232
 - objects, adding to collection, 203-207
 - one_to_many relationships, 201-203
 - one_to_one relationships, 241-246
 - model specs, 614-616
 - models, unit testing, 568-569
 - modifying ActiveRecord classes at runtime, 301-304
 - modifying default route, 68
 - Module class, 765-769
 - modules
 - AuthenticatedTestHelper, 501-502
 - auto-loading code, 8
 - Mongrel, 652
 - init script, configuring, 672-673
 - installing, 659
 - Mongrel Cluster
 - configuring, 662
 - installing, 659
 - Monit
 - configuring, 667, 669
 - init script, configuring, 673-675
 - installing, 661
 - multipart messages
 - implicit multipart messages, 539
 - sending, 537-539
 - multiserver deployments (Capistrano), 702-703
 - MySQL, installing, 660
- ## N
- named routes, 81, 94-95
 - arguments, 84-85
 - creating, 81-87
 - route helper methods, 82
 - executing in application console, 134-135
 - namespaced controllers, 107

- naming conventions
 - for database schema, 159
 - for observers, 283
 - for migrations, 144
 - for routes, 83-84
- nested resources, 101-103, 106-108
 - :name_prefix option, 103-105
 - :path_prefix option, 103
 - deep nesting, 108-109
- new actions, 123-124
- new/create operations (REST), 99-100
- Nginx
 - configuring, 663-667
 - init script, configuring, 670-672
 - installing, 659
- Nilclass, 772-773
- null objects, 608
- Number class (JavaScript), extensions, 432-433
- NumberHelper module, 383-385
- Numeric class (ActiveSupport library), 773-776

O

- Object class (ActiveSupport library), 776-779
- Object class (Prototype), 425-426
- objects
 - adding to collection, 203-207
 - errors, finding, 254
 - storing in session hash, 470
- observers, 11, 282
 - naming conventions, 283
 - registration, 283
- observe_field method, 451-453
- observe_form method, 453
- Olson, Rick, 136, 486
- on() method, 267
- one-to-many relationships, 201-203
- one-to-one relationships, 241
 - has_one relationship, 241-244
 - options, 244-246
- optimistic locking, 180-182
- options
 - for belongs_to associations, 209-215
 - for has_many associations, 216-224

- for has_one relationships, 244-246
- for mailer methods, 534-536
- for through associations, 238-241
- ORM frameworks, ActiveRecord
 - abstract base model classes, 291-292
 - before/after callbacks, 274
 - callbacks, 271, 275-282
 - halting execution, 274
 - registration, 272-273
 - classes, modifying at runtime, 301-304
 - common behavior, reusing, 296-301
 - observers, 282-283
 - polymorphic has_many relationships, 292-296
 - STI, 284-291
- overriding
 - environment.rb file, 9-11
 - TZ code, 12-13
 - to_xml method, 510

P

- page caching, 328
- PaginationHelper module, 386-388
- parameters, 60-61
 - receptors, 73-74
 - wildcard parameters for routes, 61-62
- parsing XML, 515-518
 - XmlSimple library, 516-517
- partial mocking, 609-610
- partial stubbing, 609-610
- partial templates
 - inline rendering, 35
 - rendering, 35
- partial test cases, 592-593
- partials, 320-322
 - collections, rendering, 325-327
 - logging, 327
 - passing variables to, 324-325
 - reusing, 322
 - shared, 323-324
- partial_path method, 389
- passing variables to partials, 324-325
- patterns (Selenium), 590
- PeepCode screencast, 810

- performance of applications, analyzing, 22
- periodically_call_remote method, 451
- pessimistic locking, 182
- photo_for method, writing, 408
- Piston
 - installing, 636
 - properties, 639
 - revisions, locking and unlocking, 639
 - vendor libraries,
 - converting, 638
 - importing, 637
 - updating, 638
- piston lock command, 639
- plugins, 628
 - Action Cache plugin, 337-338
 - active_record_defaults, 807
 - Acts as Authenticated
 - AccountController class, 496-498
 - authenticate method, 494
 - before save callback, 493
 - current user, 499-501
 - installing, 486-487
 - login from cookie, 498-499
 - remember token, 495
 - User model, 487-492
 - validation rules, 492
 - checking out via Subversion, 634
 - debug_view_helper, 808
 - exception_logger, 808
 - exception_notification, 808
 - has_finder, 808
 - has_many_polymorphs, 809
 - installing, 629-632
 - listing, 629-630
 - locking down, 636
 - Piston, properties, 639
 - query_trace, 809
 - Rake tasks, 647
 - Rakefiles, 648-649
 - reinstalling, 633
 - removing, 633
 - Routing Navigator, 137
 - installing, 136
 - RSpec on Rails, 613
 - generators, 614
 - model specs, 614-616
 - session timeout, 478-479
 - sources, adding, 630
 - sources, removing, 631
 - Subversion externals, 635
 - testing, 649
 - updating, 634
 - will_paginate, 386
 - writing, 640, 644-647
 - Rake tasks, 647
 - Rakefiles, 649
- plural REST routes, 98-99
- pluralization, 157-158
- pluralize method, 396
- polymorphic associations, 293-296
- polymorphic has_many relationships, 292-296
- post-backs, 112
- predicates, 599-601
- preinitializing Capistrano, 698-700
- prerequisites for production environment,
 - 652-654
- Proc class (ActiveSupport library), 781-782
- processing order of routes, 75-76
- production environment, 652
 - caching, 676
 - components of, 655
 - application tier, 656
 - database tier, 656
 - monitoring tools, 657
 - web tier, 656
 - performance issues, 676-677
 - predeployment concerns, 676-677
 - prerequisites, 652-654
 - redundancy, 676
 - scalability issues, 676-677
- production mode, 2-4, 17
- Prototype, 420-421
 - Ajax object, 436
 - Class object, 424
 - Enumerable object, 437-443
 - Firebug, 421
 - Hash class, 443
 - Object class, 425-426
 - Prototype object, 445
 - Responders object, 437
 - top-level functions, 422-424

Prototype object, 445
 PrototypeHelper module
 observe_field method, 451-453
 observe_form method, 453
 link_to_remote method, 445-449
 periodically_call_remote method, 451
 remote_form_for method, 449-451
 proxy collection objects, 202
 proxy methods for has_many associations,
 224-225
 PStore files, 474
 public instance methods, 726
 PUT requests, handling with REST, 98

Q-R

query cache, 172-174
 query_trace plugin, 809

Railcasts, 810
 Rake tasks, 647
 freezing/unfreezing, 4
 Rakefiles, 648-649
 rake tests, 587
 for SVN, 811-812
 Rakefiles, 648-649
 Range class (ActiveSupport library), 782
 RCov, 624
 reading cookies, 482
 README files, 644
 receiving email, 541-543
 file attachments, 543
 TMail methods, 542
 receptors, 61-62, 73-74
 recipes (Capistrano), managing Mongrel clusters,
 701-702
 RecordIdentificationHelper module, 388-389
 RecordInvalid method, 263
 RecordTagHelper module, 390-391
 red bars, 553
 redirect to command, 41
 redirecting
 HTTP status codes, 41-42
 requests, 39-42
 redirect_to() method (RJS), 459

Redpath, Luke, 478
 registration of observers, 283
 regular expressions in routes, 76
 reinstalling plugins, 633
 reload method, 169
 remember token, acts as authenticated, 495
 remote user accounts (Capistrano), 691
 remote_form_for method, 449-451
 remove command, 633
 remove() method (RJS), 459
 removing
 blank lines from output, 310
 plugin sources, 631
 plugins, 633
 render command, 33
 render method. *See also* rendering
 :content type option, 37
 :layout option, 37
 :status option, 37-39
 rendering
 collections, 325-327
 templates, 33-34
 inline rendering, 35
 partial templates, 35
 structured data, 36
 text rendering, 35
 view templates, 31
 replace() method (RJS), 460
 replace_html() method (RJS), 460
 representations of resources, 115
 request handling, 28
 requests
 dispatcher, 29-31
 redirecting, 39-42
 requirements for using Capistrano, 682-684
 reset_cycle method, 397
 resources (REST), 93
 controller-only, 113-115
 nested, 101-108
 deep nesting, 108-109
 name_prefix option, 103-105
 path_prefix option, 103
 representations of, 115
 respond to method, 69-70
 Responders object (Prototype), 437

- respond_to method, resource representations, 116
- REST, 91
 - controller actions, 96-98
 - controllers
 - explicitly specifying, 105
 - post-backs, 112
 - create action, 123-124
 - custom actions, syntax, 112-113
 - destroy action, 121-122
 - edit action, 124-125
 - edit/update operations, 99-100
 - formatted named routes, 117
 - HTTP requests, 96-98
 - index action, 118-121
 - named routes, 95
 - new action, 123-124
 - new/create operations, 99-100
 - plural routes, 98-99
 - resources, 93
 - controller-only, 113-115
 - nested, 101-109
 - representations of, 115
 - route customizations
 - extra collection routes, 111
 - extra member routes, 110-111
 - show action, 121
 - singular resource routes, 100-101
 - singular routes, 98-99
 - syntactic vinegar, 113
 - update action, 124-125
- REST (Representational State Transfer), 57, 89
- reusing
 - common behavior, 296-301
 - code, 628
 - partials, 322
- REXML, 515
- RJS (Ruby JavaScript), 453-456
 - alert() method, 457
 - call() method, 457
 - delay() method, 458
 - draggable() method, 458
 - drop_receiving() method, 458
 - hide() method, 458
 - insert_html() method, 458
 - literal() method, 459
 - redirect_to() method, 459
 - remove() method, 459
 - replace() method, 460
 - replace_html() method, 460
 - select() method, 460
 - show() method, 460
 - sortable() method, 460
 - templates, 455-456
 - toggle() method, 461
 - visual_effect() method, 461
- RJS behavior, testing, 581-582
- robots, disabling sessions, 472
- route customizations (REST)
 - extra collection routes, 111
 - extra member routes, 110-111
- route globbing, 79-80
 - globbing key-value pairs, 80-81
- route objects, 129-131
- routes
 - bound parameters, 59-61
 - custom routes, writing, 72
 - default routes, :id value, 77-78
 - dumping, 128-129
 - empty routes, 71
 - literal URLs, 79
 - manual recognition and generation in application console, 132-134
 - named routes, 81
 - arguments, 84-85
 - creating, 81-83
 - creating via with options mapping method, 86-87
 - executing in application console, 134-135
 - naming, 83-84
 - processing order, 75-76
 - regular expressions, 76
 - routes.rb file, 63-64
 - ante-default route, 69-70
 - default route, 65-68
 - static strings, 62-63, 72-73
 - syntax, 59
 - testing, 135
 - url for method, 76-77
 - wildcard parameters, 61-62

- routes.rb file, 63-64
 - ante-default route, 69-70
 - respond to method, 70
 - custom routes, writing, 72
 - default route, 65-66
 - :id field, 66
 - generating, 67-68
 - modifying, 68
 - empty routes, 71
 - routing, 59
 - receptors, 73-74
 - Routing Navigator plugin, 137
 - installing, 136
 - routing rules, testing, 582-584
 - RSelenese, 592
 - partial test cases, 592-593
 - RSpec, 546
 - arbitrary predicates, 599-601
 - Autotest project, 624
 - behaviors, 603-604
 - shared behaviors, 604-607
 - controller specs, 617-619
 - expectations, custom expectation matchers, 601-603
 - installing, 613
 - mock objects, 607-608
 - null objects, 608
 - partial mocking, 609-610
 - partial stubbing, 609-610
 - RCov, 624
 - running specs, 610
 - scaffolding, 623
 - specdoc format, 611-612
 - specs, helper specs, 623
 - stub objects, 608
 - view specs, 621
 - helper methods, stubbing, 623
 - instance variables, assigning, 622
 - RSpec on Rails, 613
 - errors, specifying, 620
 - generators, 614
 - model specs, 614-616
 - integration mode, 619
 - isolation mode, 619
 - routes, specifying, 620
 - RSpec scripts, 597
 - expectations, 599
 - Ruby, installing, 658
 - RubyGems, 5
 - installing, 658
 - running edge rails, 4
 - running specs, 610
- ## S
- sanitize method, 397
 - scaffolding, 623
 - schema dumping, 11
 - SCM System (Capistrano), customizing, 692
 - scoping Capistrano variables, 696-698
 - screencasts, 810
 - script/plugin command, 629
 - script/runner, adding background processing to
 - applications, 708-710
 - Scriptaculous
 - dragging and dropping, 463-464
 - in-place editors, 467
 - sortable lists, creating, 465-466
 - scripts
 - RSpec, 597
 - expectation matchers, 599
 - spawner script, 688
 - SpiderTester, 809
 - securing sessions, 480
 - select helper methods, 371-373
 - select tags (HTML), creating for calendar data, 355-359
 - select() method (RJS), 460
 - Selenium, 589
 - actions, 589
 - assertions, 590
 - locators, 590
 - patterns, 590
 - RSelenese, partial test cases, 592-593
 - Selenium on Rails, 591
 - self-referential relationships, 226
 - send data() method, 51-52
 - send file() method, 52-55
 - sending
 - email, 540
 - file attachments, 539-540
 - HTML messages, 536-537
 - multipart messages, 537-539
 - serialized attributes, 162

- session hash
 - current user id, storing, 470
 - objects, storing, 470
- session id, 469
- sessions, 469, 586
 - cleaning up, 481
 - enabling, 473
 - for robots, disabling, 472
 - options, 471
 - securing, 480
 - securing with HTTPS, 473
 - storing
 - DRb session storage, 475
 - in ActiveRecord, 473-474
 - in CookieStore, 476-478
 - in memcache, 475-476
 - in PStore files, 474
 - timing out, 478-479
 - tracking, 479-480
- setting up ActionMailer framework, 531
- setup methods, 553
- shared behaviors, 604-607
- shared partials, 323-324
- shared secrets, 495
- Shaw, Zed, 652
- show actions, 121
- show() method (RJS), 460
- simple_format method, 398
- single responsibility principle, 282
- singular resource routes, 100-101
- singular REST routes, 98-99
- skipping
 - error validation, 269-270
 - frameworks, 9
- SMTP server, configuring, 543
- sortable lists, creating, 465-466
- sortable() method (RJS), 460
- sources command, 630
- spec method, specdoc format, 611-612
- specdoc format (spec method), 611-612
- specs, 597
 - controller specs, 617-619
 - expectations, 599
 - helper specs, 623
 - model specs, 614-616
 - running specs, 610
 - view specs, 621
 - helper methods, stubbing, 623
 - instance variables, assigning, 622
- SpiderTester script, 809
- SQL
 - custom options, 229-231
 - custom queries, 171-172
- static strings, 62-63, 72-73
- Stephenson, Sam, 420
- STI (single-table inheritance), 284-291
 - inheritance, mapping to the database, 286-287
- storing
 - database.yml, 693-695
 - sessions
 - ActiveRecord, 473-474
 - CookieStore, 476-478
 - current user id, 470
 - DRb session storage, 475
 - memcache, 475-476
 - objects, 470
 - PStore files, 474
- streaming, 51
 - send data() method, 51-52
 - send file() method, 52-55
- String class (ActiveSupport library), 783-790
- String class (JavaScript), extensions, 433-436
- strings, typecasting, 518
- strip_links method, 398
- strip_tags method, 398
- structured date, rendering, 36
- stub objects, 608
 - partial stubbing, 609-610
- stubbing, 549-550
- stylesheet_link_tag method, 352
- stylesheet_path method, 352
- Subversion, 810-811
 - externals, 635
 - installing, 660
 - plugins, updating, 634
 - rake tests, 811-812
 - website, 633
- suffixes for templates, 312
- Susser, Josh, 112, 233, 369
- SVN (Subversion). *See* Subversion
- Sweeper class, 335-336
- Symbol class (ActiveSupport library), 790

syntactic sugar, 113
 syntactic vinegar, 113
 syntax
 for custom REST actions, 112-113
 for hashes in arguments, 136
 Syslog, 24

T

TagHelper module, 391-392
 templates
 application.rhtml layout template, 312-313
 delimiters, 308-310
 partials, 320-322
 collections, rendering, 325-327
 logging, 327
 passing variables to, 324-325
 reusing, 322
 shared partials, 323-324
 rendering, 33-35
 RJS, 455-456
 structured data, rendering, 36
 suffixes, 312
 text, rendering, 35
 variables, 315
 instance variables, 315-319
 test mode, 17
 test suites, 553
 testing, 553
 acceptance tests, 588-589
 assertions, 562-566
 assert, 563
 one-assertion guideline, 566-567
 errors, 553
 failures, 553
 fixtures, 554-556
 CSV fixtures, 555
 disadvantages of, 560-561
 dynamic content, 556-557
 generating from development data, 558-559
 in development mode, 557
 options, 559
 functional tests, 570-571, 576-580
 assertions, 572-576
 equality parameter, 580-581
 methods, 571
 routing rules, testing, 582-584
 selection methods, 582
 testing RJS behavior, 581-582
 integration tests, 550-551, 584-586
 plugins, 649
 Selenium, 589
 actions, 589
 assertions, 590
 locators, 590
 patterns, 590
 routes, 135
 unit tests, 568-569
 with Rake, 587
 XUnit framework, 552
 text, rendering, 35
 TextHelper module, 393-400
 textilize method, 399
 textilize_without_paragraph method, 399
 tiles help method, writing, 410-416
 time and date information, creating HTML
 select tags, 355-359
 Time class (ActiveSupport library), 792-798
 time zones, 798-802
 timing out sessions, 478-479
 TMail, 540
 file attachments, receiving, 543
 methods, 542
 toggle() method (RJS), 461
 to_xml method, 503-504
 Array class, 510-513
 :include parameter, 507-508
 :methods parameter, 508-509
 :procs parameter, 509-510
 output, customizing, 505-506
 overriding, 510
 tracking sessions, 479-480
 transactional fixtures, 559
 transactions (Capistrano), 703-705
 truncate method, 399
 typecasting strings, 518
 TZ (time zone) code, overriding in environ-
 ment.rb file, 12-13

U

- unfreezing edge rails applications, 4
- uninstall.rb file, 645-647
- uniqueness of join models, enforcing, 262
- unit tests, 568-569
- unlocking Piston revisions, 639
- unsaved associations, 246
- unsource command, 631
- update actions, 124-125
- Update method (ActiveResource), 524
- updating plugins, 634
- url for method, 76-77
- UrlHelper module, 400-406
- URLs, generating, 133-134
- url_for method, 405-406
- User model, Acts as Authenticated, 487-492
- user-submitted content, preventing injection attacks, 319

V

- validates_acceptance_of method, 254
- validates_associated method, 255-256
- validates_confirmation_of method, 256
- validates_each method, 256
- validates_exclusion_of method, 257
- validates_existence_of method, 257
- validates_format_of method, 258-259
- validates_inclusion_of method, 257
- validates_length_of method, 259-260
- validates_numericality_of method, 260
- validates_presence_of method, 261
- validates_uniqueness_of method, 261-262
- validation errors
 - reporting, 342-343
 - skipping validation, 269-270
- validation methods
 - allow_nil option, 263
 - conditional validation, 264
 - when to use, 265-266
 - custom validation, 268-269

- message option, 263
 - on option, 264
- validation rules, Acts as Authenticated, 492
- variables, 315
 - instance variables, 315-319
 - passing to partials, 324-325
- variables (Capistrano), 696-698
- vendor libraries
 - converting to Piston, 638
 - importing into Piston, 637
- view specs, 621
 - helper methods, stubbing, 623
 - instance variables, assigning, 622
- view templates, rendering, 31
- viewing generated routes, 132
- views
 - functional testing, 576-580
 - equality parameter, 580-581
 - routing rules, testing, 582-584
 - selection methods, 582
 - instance variables, 42-43
- visual_effect() method (RJS), 461

W

- Wanstrath, Chris, 807
- websites, Subversion project, 633
- whiny nil, 772
- wildcard parameters for routes, 61-62
- Williams, Bruce, 387
- will_paginate plugin, 386
- word_wrap method, 400
- writing
 - cookies, 482
 - custom routes, 72
 - helper methods, 407-408
 - breadcrumbs, 409-410
 - photo_for, 408
 - tiles, 410-416
 - helper modules, 407
- writing plugins, 640, 644-647
 - Rake tasks, 647
 - Rakefiles, 649

X-Y-Z

XML

- Builder API, 513-515
- parsing, 516-518
- parsing into Ruby hashes, 515-517
- rendering, 36
- to_xml method, 503-504
 - include parameter, 507-508
 - methods parameter, 508-509
 - output, customizing, 505-506
 - procs parameter, 509-510
- XmlSimple library, 516-517
- XUnit framework, 552

- YAML (Yet Another Markup Language), 554
- yield keyword, 313
- yielding additional HEAD element content, 315